

La fase PRE en PYST-PUB. Versión 2.1

De \documentclass a la activación de una clase KOMA

F. Javier Payán Somet

31 de agosto de 2026

Versión 15.0 del manual

Índice

I	La fase PRE de un documento del sistema PYST-PUB	2
1	Presentación de la fase PRE del sistema PYST-PUB	2
1.1	Los cuatro ficheros	3
2	Los catálogos de configuración	6
2.1	\DeclareDefaultTypeOptions	6
2.2	\DeclareDocumentType	8
2.3	\DeclareDepartment	9
2.4	Las variables del tipo secuencias de los elementos registrados	11
3	Definición de los espacios de claves	12
3.1	Anatomía del comando keys_define:nn	13
3.2	El namespace pyst	14
3.2.1	Las claves que consume el pre-escaneo	14
3.2.2	Las claves que configuran el documento	14
3.2.3	Las claves de activación	17
3.2.4	Claves de configuración estructural aditivas	19

3.3	Las claves del namespace <code>pyst/dept</code>	24
3.4	Las claves que no se reconocen	25
3.5	Tabla resumen de las claves de los namespace	25
4	Sobre las claves reservadas	26
4.1	Por qué unas variables se declaran reservadas y otras no	27
4.2	Codificación de la reserva de claves	27
5	Cómo se leen los valores del usuario en el sistema PYST-PUB	30
5.1	El Invariante: Jerarquía y Reglas de Precedencia	30
5.2	Lectura de los catálogos de configuración	32
5.3	Prevalidación del tipo de documento y el departamento	32
5.4	Bloque 4. Aplicación de los valores por defecto y del tipo de documento	36
5.5	Bloque 5. Aplicación del departamento	37
5.6	Bloque 6. Aplicación de los valores declarados por el usuario	39
5.7	Bloque 7. Filtrar y no interceptar	40
5.8	Bloque 8. El conflicto de <i>layout</i>	42
5.9	La función <code>\pyst_validate_keyval:nnn</code>	43
6	La entrega a KOMA	45
6.1	Bloque 1. Ensamblado de la lista de opciones	46
6.2	El conflicto de <i>layout</i> , otra vez	47
6.3	Las dos constantes públicas	48
6.4	Bloque 3. La proyección a condicionales <code>\newif</code>	48
6.5	Bloques 2 y 4. Los hechos que se deducen	49
6.6	Bloque 5. Las comprobaciones defensivas	50
6.7	Bloques 6 y 7. Geometría y babel, como cadenas	51
6.8	Bloque 8. Los accesorios públicos	51
6.9	Cargar, configurar, recalcular	52
6.10	Qué queda al final de la fase PRE	54
7	Resumen	54
7.1	En qué consiste la fase PRE	55
7.2	Los valores vienen de cuatro fuentes diferentes	55
7.3	Primer paso. Guardar sin entender	56
7.4	Segundo paso. Declarar un espacio de nombres	57
7.5	Tercer paso. Volcar unas claves sobre las otras	57
7.6	Un tercer espacio de nombre que no es un espacio de nombre	58
7.7	Y lo que queda después	58
II	Aspectos técnicos en profundidad	60
8	Análisis del comando <code>\pyst_coverstyle_normalize:n</code>	60
8.1	Firma del Comando y Normas de Nomenclatura (<code>expl3</code>)	60

8.2	Análisis Detallado Línea por Línea	61
8.2.1	Paso 1: Limpieza e Higienización Inicial	61
8.2.2	Paso 2: Evaluación de Casos Especiales (“none” o Vacío)	61
8.2.3	Paso 3: Validación de Seguridad y Filtrado de Caracteres Prohibidos	62
8.2.4	Paso 4: Normalización de la Extensión (.def)	62
9	Análisis de la Función del núcleo de expl3 \keys_set:nn	64
9.1	Sintaxis y Firma de la Función	64
9.2	Ciclo de Vida y Flujo Interno de Procesamiento	64
9.3	Variantes de Expansión	65
9.4	Variaciones de Tolerancia y Filtrado	66
9.4.1	Ignorar Claves Desconocidas (\keys_set_known:nn)	66
9.4.2	Capturar Claves Desconocidas (\keys_set_known:nnN)	66
9.4.3	Filtrado por Grupos (\keys_set_filter:nnn)	66
10	El vocabulario de expl3, reunido	66
10.1	Cómo se lee un nombre	66
10.2	Los prefijos de las variables	67
10.3	Las acciones que se repiten	67
10.4	Los sufijos de la firma	68
10.5	Dos idiomas que conviene reconocer	68

Parte I

La fase PRE de un documento del sistema PYST-PUB

ESTE DOCUMENTO cumple con dos objetivos. Por un lado, analizar la fase PRE del sistema PYST-PUB y por otro, aprovechar este análisis para estudiar la sintaxis del nuevo lenguaje `expl3` en el que está escrita. Este doble análisis nos permitirá presentar no solo qué se hace sino cómo se hace dentro del nuevo régimen en el que se están escribiendo los nuevos desarrollos de $\text{\LaTeX}$, en lo que se ha dado por llamar $\text{\LaTeX}3$. 1234567890

1 Presentación de la fase PRE del sistema PYST-PUB

El sistema PYST-PUB está construido como una envoltura de la clase KOMA adaptada a la publicación de documentos en un entorno académico. Se puede dividir en dos fases PRE y POST con propósitos y métodos bien diferenciados.

Mientras que la fase POST recoge la experiencia previa del sistema de publicaciones de la Escuela Técnica Superior de Ingeniería de la Universidad de Sevilla, la fase PRE, como se señala en el subtítulo de este documento tiene un único propósito: convertir lo que el usuario escribió en `\documentclass[...]{pyst-pub}`, combinado con lo que declaran unos ficheros de configuración o catálogos, en una llamada correcta a `\LoadClass` sobre una de las tres clases de KOMA-Script, `scrbook`, `scrreprt` o `scrartcl` con exactamente las opciones que corresponde pasarle. Nada más. Cuando esa llamada termina, la fase PRE ha acabado su trabajo y KOMA, junto con los paquetes elegidos por el sistema PYST-PUB (la fase POST), se hace cargo de generar el documento.

Ese propósito, tan sencillo de enunciar, tiene una dificultad concreta debida a la existencia de *cuatro* fuentes de opciones que pueden contradecirse entre sí. Por un lado, nos encontramos con el fichero de configuración de los tipos de documentos, en el cual se recoge la existencia de un conjunto de valores por defecto. Por otro lado, en el mismo fichero de configuración, se definen cada uno de los tipos de documento, con valores que pueden contradecir/modificar/ampliar los declarados por defecto. Por una vía paralela se pueden declarar valores institucionales de diferentes departamentos; valores que se declaran en el fichero `pyst-depts.cfg`.

Sobre estas dos vías de independientes de introducción de valores, una que hace referencia a los documentos y otra a una posible declaración de un departamento y sus datos institucionales, el usuario también puede realizar cambios a través de parámetros declarados en

`\documentclass` y/o el preámbulo del documento. Gráficamente, las cuatro fuentes de valores que permitirán construir el departamento se muestran en el siguiente esquema.

$$\left. \begin{array}{l} 1.- \text{ Defaults} < 2.- \text{ Tipo} \\ 3.- \text{ Departamento} \end{array} \right\} < 4.- \text{ Usuario}$$

Esquema que también señala una precedencia de cómo considerar los valores. Los del tipo de documento están por encima de los valores por defecto; en paralelo, y de forma independiente, se encuentran los valores institucionales de un posible departamento y, por encima de todos ellos, los valores declarados por el usuario. Un objetivo de la fase PRE es traducir este esquema mediante una codificación adecuada.

Y antes de entrar en ningún mecanismo, conviene dejar escrita otra regla de diseño de la que se derivan decisiones importantes.

No reinventar lo que KOMA-Script ya hace.

El sistema PYST-PUB no calcula márgenes, no decide tamaños de papel, no valida nombres de opciones de `typearea` ni de `geometry`, y no mantiene su propia noción de qué es una opción de clase válida más allá de las suyas propias. Todo eso ya lo resuelve KOMA, y resolverlo dos veces no añade seguridad, añadiría una segunda fuente de comandos, opciones y definiciones que habría que mantener sincronizadas con KOMA y, por lo tanto, generaría una fuente de posibles errores¹. Por ello, el papel de PYST-PUB se limita a *reunir* las opciones correctas, en el orden correcto, y *entregárselas a KOMA por los cauces adecuados* para que KOMA actúe.

Esta regla reaparece varias veces en esta nota, siempre en el mismo sentido: cada vez que hay una tentación de que PYST-PUB resuelva algo por su cuenta, la pregunta correcta es pensar si KOMA ya ofrece la manera de hacerlo. En caso afirmativo, pedirselo.

1.1 Los cuatro ficheros

La codificación de la fase PRE está repartida en los cuatro ficheros definidos en la [Tabla 1](#), que `pyst-pub.cls` carga en ese orden, antes de llamar a la clase base.

En el primero de los ficheros, `pyst-core.ltx`, se realiza en primer lugar una validación del motor que se está utilizando para la compilación del documento. En el caso de que no sea L^AT_EX, se emite un mensaje de aviso y se aborta la compilación.

Pasado este primer control, en `pyst-core.ltx` se registran los mensajes que van a utilizarse en el sistema. Este registro de mensajes centralizados permite y facilita su posterior mantenimiento. Con este fin, se utiliza el comando `\msg_new:` con una firma que dependerá del número de parámetros con el que está definido. Por ejemplo, el siguiente código muestra el mensaje que aparecería si el *tipo de documento* asignado a la variable `doc-type` no estuviese reconocido:

```
\msg_new:nnnn { pyst } { unknown-type }
{
  Unknown~type~'#1'.'.~
```

¹En versiones de desarrollos de PYST-PUB se cayó varias veces en ese error.

Tabla 1: Ficheros componentes de la fase PRE.

Fichero	Papel
<code>pyst-core.ltx</code>	Registro: valida el motor, declara todas las variables globales y todos los mensajes del sistema, y aporta los auxiliares de depuración. Añade además la comprobación de coherencia de versiones de la propia fase PRE.
<code>pyst-configuration.ltx</code>	Definiciones: define los comandos con los que se escriben los catálogos de configuración (<code>\DeclareDocumentType</code> , <code>\DeclareDepartment...</code>), los <i>dos</i> espacios de claves, el guardián de claves reservadas, los gestores de ruta y las funciones de normalización. Y qué significa cada clave declarada en el sistema PYST-PUB.
<code>pyst-options.ltx</code>	Coordinación: leer los catálogos, validar y aplicar las cuatro fuentes en el orden correcto contra los dos espacios de nombres. Resolver conflictos en las declaraciones.
<code>pyst-build.ltx</code>	Exportación: traducir el resultado a lo que <code>\LoadClass</code> y <code>\KOMAOPTIONS</code> necesitan, y proyectar los interruptores internos a condicionales <code>\newif</code> para el resto del sistema.

```

Available:~\seq_use:Nn \g_pyst_available_types_seq { ,~ }
}
{
  Check~the~value~of~doc~type~for~tipos~(case-sensitive:~'libro'~
and~'Libro'~are~different~strings~to~this~check,~for~example).~The~
full~list~
of~registered~types~are:~\seq_use:Nn \g_pyst_available_types_seq { ,~ }
.
}

```

Como tercer bloque en el fichero `pyst-core.ltx` se realiza un registro de las variables globales internas del sistema PYST-PUB, haciendo uso de diversos comandos de `expl3` en función del tipo de variable. Así, encontramos declaración de variables del módulo *property list*, mediante el comando `\prop_new:N`, como variables del módulo *token list*, haciendo uso de `\tl_new:N`, del módulo *clist*, utilizando `\clist_new:N` y variables del tipo *booleano*, `\bool_new:N`.

Como puede observarse de su firma, todas estas funciones reciben como parámetro el nombre de una variable, como por ejemplo, `\g_pyst_koma_layout_bool`. En la definición del nombre de las variables se ha intentado mantener las especificaciones del régimen `expl3`.

Para finalizar, en `pyst-core.ltx` se definen variables y funciones que permitirán establecer un registro de depuración, y un último bloque que comprueba coherencia de versiones de la propia fase PRE.

La razón de este bloque la encontramos en la manera en la que se instala/actualiza el sistema PYST-PUB. Recordemos que cada usuario copia un fichero de distribución en su propio árbol

TEXMFLOCAL. En este caso, un posible fallo no es la manipulación² sino la posibilidad de que alguien sustituya un fichero corregido de la fase PRE y deje los otros tres de una versión anterior. Nada en el sistema se enteraría, y el síntoma de un posible error podría aparecer mucho después como un comportamiento que no corresponde a ninguna versión publicada. Por ello, se han definido dos funciones y una secuencia.

```
\cs_new_protected:Npn \pyst_load_pre_file:n #1
{
  \file_input:n { #1 }
  \seq_gput_right:Nn \g_pyst_pre_files_seq { #1 }
}

\seq_gput_right:Nn \g_pyst_pre_files_seq { core/pre/pyst-core.ltx }
```

`\pyst_load_pre_file:n` no hace más que cargar el fichero y anotar su nombre. La lista de ficheros de la fase PRE **no se declara en ninguna parte**, la construye este envoltorio conforme `pyst-pub.cls` los va cargando, de modo que las líneas de carga de la clase *son* la lista. La asimetría de la última línea es deliberada y conviene no “corregirla”. `pyst-core.ltx` es también un fichero de la fase PRE, pero se carga antes de que exista el envoltorio que se ha definido precisamente en `pyst-core.ltx`, así que es necesario que su nombre se inscribe a mano.

La versión de cada fichero no se declara tampoco, ya que la declara cada uno en su propia cabecera con `\ProvidesExplFile`, que deja **FECHA** **vVERSIÓN** **DESCRIPCIÓN** en `\ver@<clave>`. La comprobación se limita a extraer el segundo campo de cada uno y compararlos entre sí. Además escribe siempre en el `.log` qué ficheros se cargaron y con qué versión. Algo parecido a lo siguiente.

```
PYST PRE: core/pre/pyst-core.ltx v2.1
PYST PRE: core/pre/pyst-configuration.ltx v2.1
PYST PRE: core/pre/pyst-options.ltx v2.1
PYST PRE: core/pre/pyst-build.ltx v2.1
```

Ese registro es incondicional, no depende de `debug=true`, porque el `.log` que llega adjunto a un informe de problemas suele ser de una compilación normal. Responde a la primera pregunta que necesita cualquier diagnóstico *emph* ¿qué se estaba ejecutando? y es la más difícil de contestar.

Conviene establecer una clara distinción entre la función realizada por los ficheros que se cargan a continuación `pyst-configuration.ltx` y `pyst-options.ltx`. Hemos utilizados dos ficheros diferentes para acentuar el papel que hace cada uno. En `pyst-configuration.ltx` se definen comandos y espacios de nombres, lo que nos lleva a definir qué significa cada clave que se utiliza en PYST-PUB. Veremos que para conseguir el diseño deseado hemos establecido que determinadas claves van a ser reservadas. Aquí diremos cuales. En `pyst-configuration.ltx` hay declaraciones y definiciones. Si la fase acabara aquí, no se generaría ningún documento.

En el fichero `pyst-options.ltx` ocurre algo muy diferente. Su papel fundamental es saber qué ha dicho el usuario. Y utilizando qué ha dicho, es decir, qué ha declarado en `\documentclass`,

²Contra la que ningún mecanismo de **TEX** puede hacer nada, porque cualquier macro se puede volver a redefinir en el preámbulo.

y conociendo los ficheros de configuración, organizar todos estos datos para que el sistema construya un documento de acuerdo con nuestras reglas de diseño.

En el último de los cuatro ficheros de esta fase PRE, `pyst-build.ltx` se realiza una exportación de todos los valores declarados al resto del sistema, comenzando por la carga de la clase de KOMA que se va a utilizar. Se activan una serie de interruptores que en la siguiente fase permitirán que se carguen (o no) determinados paquetes, ficheros, tipografías, módulos etc. Y se devuelve el control al fichero `pyst-pub.cls`. Lo que se realiza a continuación está fuera de esta nota que está circunscrita a la fase PRE, que ya ha acabado.

2 Los catálogos de configuración

En los catálogos de configuración se recogen tres de las posibles fuentes de entradas de datos en el programa. Puesto que se supone que estas entradas deben poder utilizarse por un usuario habitual, los catálogos de configuración son ficheros de textos plano, en los que resulta muy fácil conocer cómo declarar un nuevo tipo de documento o un nuevo departamento. En el Manual de Usuario está descrito este procedimiento,

Desde un punto de vista técnico, la declaración se consigue utilizando tres comandos `\DeclareDefaultTypeOptions`, `\DeclareDocumentType` y `\DeclareDepartment` que vamos a describir en este Apartado. Estos tres comandos están definidos en el fichero `pyst-configuration.ltx`. Y, como hemos dicho anteriormente, la definición de los comandos no significa que se ejecuten en este fichero. Es decir, aquí vamos a ver qué hacen los tres comandos y, posteriormente, cuando analicemos el fichero `pyst-options.ltx` veremos el resultado de su ejecución.

Los tres comandos hacen en esencia lo mismo. Dado un tipo de documento o el acrónimo de un departamento, definidos a través de una cadena de `clave=valor`, separadas por comas, conocidas en la terminología `expl3` como *keyval*, los tres comandos guardan el *keyval*³ tal cual, sin interpretarlo, en un conjunto de variables globales.

Para finalizar el registro de los tipos de documentos que va a utilizar el sistema PYST-PUB veremos que es necesario crear una variable global que permita su conocimiento por el resto del sistema. Lo mismo ocurre para los departamentos. Es necesario crear una variable global que nos permita saber qué departamentos han sido declarados, utilizando para ello el acrónimo de los mismos.

En el resto de este Apartado se analizan por separado los comandos de declaración y el procedimiento seguido para registrar los tipos de documentos y departamentos que componen nuestros elementos de configuración.

2.1 `\DeclareDefaultTypeOptions`

El siguiente código muestra la definición del comando `\DeclareDefaultTypeOptions`.

```
\NewDocumentCommand \DeclareDefaultTypeOptions { m }
{ \tl_gset:Nn \g_pyst_type_defaults_tl { #1 } }
```

³Aunque no es una palabra reconocida en nuestro idioma, al tener un significado técnico tan concreto vamos a utilizarla a partir de ahora como tal, en itálica, como *keyval*. Debemos tener en cuenta, además, que a veces el valor se omite: `math` y `math=true` tienen exactamente el mismo significado.

Mediante `\NewDocumentCommand`, el mecanismo moderno de L^AT_EX2_ε (paquete `xparse`) para definir comandos nuevos (sustituyendo el antiguo `\newcommand`), se declara un comando de nombre `\DeclareDefaultTypeOptions` que admite un único parámetro *obligatorio* `{ m }`. El cuerpo del comando está formado por una única línea de manera que todo lo que estaba escrito entre las llaves de `\DeclareDefaultTypeOptions{...}` llega aquí como `#1`.

`\tl_gset:Nn` es la primera función significativa de `expl3` que vemos en este documento y aparecerá constantemente, así que merece la pena descomponer su sintaxis, común al resto de funciones de `expl3`. El prefijo de la función, `tl`, hace referencia al módulo en el que se encuentra agrupada, en este caso, el que gestiona los denominados *token lists*, es decir, *cadenas de texto con nombre*, lo más parecido a una variable de texto de cualquier otro lenguaje; `gset` significa realiza una asignación *global* del contenido de una variable, de forma que ese valor persiste durante *toda* la compilación y generación del documento, no solo dentro del grupo actual.

El sufijo, o *firma* del comando, separado del nombre del comando por `:`, `Nn` en este caso, indica cómo se leen los dos argumentos de la función `\tl_gset`. El primero, `N`, indica que espera recibir una macro o comando sin llaves. En este caso, recibe la variable de destino: `\g_pyst_type_defaults_tl`, que ha sido creada previamente en `pyst-core.ltx`. El segundo, `n`, indica que recibirá un texto normal o un código envuelto entre llaves `{ ... }`. En este caso, recibe `#1` (que representa el texto que el usuario introdujo en el comando).

Así que `\DeclareDefaultTypeOptions` no hace nada más que copiar, literalmente, todo lo que está escrito entre llaves a una variable de texto global llamada `\g_pyst_type_defaults_tl`. No interpreta ninguna de las claves que contiene (por ejemplo, `language`, `babel`, `math`...); no comprueba que tengan sentido; no hace nada con ellas todavía. Es, literalmente, un almacén de texto que se queda a la espera de futuras acciones. Podremos observar que en el fichero `pyst-types.cfg` el comando se invoca *una sola vez*, al principio del fichero, antes del primer `\DeclareDocumentType`.

Si en `pyst-types.cfg` aparece por ejemplo

```
\DeclareDefaultTypeOptions{
  language      = spanish,
  babel         = {spanish, es-ucroman, es-noshorthands,
                  es-nosectiondot, es-noindentfirst,
                  es-nolists, shorthands=off},
  graphics      = true,
  math          = true,
  doc-options   = {twoside=true, headings=normal,
                  headsepline=true,numbers=noendperiod}
}
```

una vez leído el fichero `pyst-types.cfg` la variable `\g_pyst_type_defaults_tl` tendrá el valor:

```
language=spanish, babel={spanish, es-ucroman, es-noshorthands, es-
nosectiondot, es-noindentfirst, es-nolists, shorthands=off}, graphics=
true, math=true, doc-options={twoside=true, headings=normal,}
```

Las seis líneas, con sus llaves, han quedado guardada tal cual en `\g_pyst_type_defaults_tl`.

2.2 \DeclareDocumentType

La función del comando `\DeclareDocumentType` es declarar un *tipo de documento* y, como podemos ver en el fichero `doc-types.cfg` se realiza, por ejemplo en la forma

```
\DeclareDocumentType{libro}{
  base-class      = scrbook,
  doc-style       = book,
  coverstyle      = libro,
  koma-layout     = {DIV=12, BCOR=20mm},
  advanced-math   = true,
  modules         = {pyst-book-common, pyst-book-edition}
}
```

La definición del comando `\DeclareDocumentType` se recoge en las siguientes líneas de código.

```
\NewDocumentCommand \DeclareDocumentType { m m }
{
  \prop_gput:Nnn \g_pyst_types_prop { #1 } { #2 }
  \msg_info:nnn { pyst } { type-registered } { #1 }
}
```

Nos encontramos en este caso con dos parámetros obligatorios (`{ m m }`). `#1` es el nombre del tipo (`libro` en el ejemplo), `#2` es el *keyval* que podemos ver entre las llaves siguientes.

En la definición se utiliza una nueva función de `expl3`, `\prop_gput:Nnn`. De acuerdo con su sintaxis, pertenece al *módulo prop* (*property list*) del lenguaje `expl3` y probablemente es la función más importante de este Apartado. Podemos definirla como una función de asignación global del módulo `prop` con firma o especificador de argumentos `Nnn`; es decir, espera recibir

- N** Un único token (No rodeado por llaves), que es el nombre de la variable destino, `\g_pyst_types_prop`.
- n** Un argumento normal (rodeado por llaves), (`#1`).
- n** Otro argumento normal (rodeado por llaves), (`#2`)

La función opera sobre un tipo de dato o variable `prop`. Una *property list* (`prop`), una variable del tipo `prop`, es lo que en otros lenguajes de programación se denomina *diccionario* o *mapa*. Puede visualizarse como una tabla de dos columnas, de manera que a cada elemento de la primera columna le corresponde un elemento de la segunda. En ocasiones, a estas dos columnas se le asignan los nombres de *clave* y *valor*; valor que, en este caso es un *keyval*. En adelante, utilizaremos el término *lista de propiedades* para referirnos a este tipo de variables.

La función `\prop_gput:Nnn` coloca en la primera columna de la variable *global* del tipo `prop`, `\g_pyst_types_prop`, el *nombre del tipo de documento*. En la segunda columna, el conjunto de pares `clave=valor` que lo definen; es decir, esta segunda columna no contiene un dato simple, sino el *keyval* propio del tipo. El comando `\prop_gput:Nnn` añade, o *sobrescribe*, si ya existiera, dentro de la variable `\g_pyst_types_prop` el par `libro → valor asignado`.

Después de terminar de leerse el fichero `pyst-types.cfg` el contenido de la variable global `\g_pyst_types_prop` será el mostrado en la [Tabla 2](#).

Tabla 2: `\g_pyst_types_prop`, tras cargar `pyst-types.cfg` completo.

Clave	Valor asignado al tipo (como texto)
libro	<code>base-class=scrbook, doc-style=book, coverstyle=libro, koma-layout={DIV=12,BCOR=20mm}, advanced-math=true, modules={...}</code>
examen	<code>base-class=scrartcl, coverstyle=examen, geometry-layout={...}, advanced-math=true, modules={...}</code>
minimo	<code>base-class=scrartcl, coverstyle=minimo, ...</code>
...	(siete tipos más)

Por último, la segunda línea de la función `\DeclareDocumentType`, `\msg\info:nnn` nos presenta una nueva función del módulo `msg`. Esta función escribe una línea informativa en el `.log` (“Type “libro” registered”), visible solo si se inspecciona el registro de compilación; no interrumpe nada y no se muestra en pantalla por defecto. Es un mensaje informativo sin condición alguna.

2.3 \DeclareDepartment

Un departamento se declara con el mismo patrón utilizado en la declaración de un tipo de documento. En este caso, se crea la lista de propiedades (`\g_pyst_depts_config_prop`), con la misma indiferencia hacia su contenido. El fichero `pyst-depts.cfg` presenta el aspecto de una ficha institucional. Por ejemplo,

```
\DeclareDepartment{dtsc}{
  name      = {Teoría de la Señal y Comunicaciones},
  university = {Universidad de Sevilla},
  school     = {Escuela Técnica Superior de Ingeniería},
  address    = {Camino de los Descubrimientos s/n \ 41092 Sevilla SPAIN
},
  phone      = {+34 954 487 331},
  email      = {teseyco@us.es},
  logo       = {logoTSC.pdf},
  def        = {defs/dept/pyst-dept-dtsc.def}
}
```

Aquí podemos observar que los ocho valores están escritos entre llaves, a diferencia de los ejemplos presentes en `pyst-types.cfg`. Esto no es un capricho tipográfico sino que se debe a la manera en la que se procesará posteriormente un *keyval*, en la que se supone que los pares `clave=valor` están separados por comas. Una dirección postal real, por ejemplo, tiene todas las papeletas de contener una coma en su valor: “Calle X, 41092 Sevilla”. Así, sin llaves, esa coma se interpretaría como el separador entre `address` y el siguiente campo, y el valor se cortaría exactamente ahí. Por ello, envolver cada valor entre llaves es lo que protege el *keyval* de cualquier coma que el contenido pueda llevar dentro, la lleve o no en un caso concreto. La función que se utiliza en la declaración del departamento se define en el código siguiente.

```
\NewDocumentCommand{\DeclareDepartment}{ m m }
{
  \prop_gput:Nnn \g_pyst_depts_config_prop { #1 } { #2 }
  \msg_info:nnn { pyst } { dept-registered } { #1 }
}
```

Es importante detenerse en un detalle de la lectura posterior del fichero `pyst-depts.cfg`. La mayor parte de la fase PRE está escrita utilizando un régimen $\text{\LaTeX}3$, como puede observarse en la declaración inicial de cada fichero de esta fase. Por ejemplo,

```
\ProvidesExplFile{core/pre/pyst-configuration.ltx}{2026/08/15}{v2.1}{PYST~
document~types~and~departments~system}
```

Esta misma declaración muestra este segundo detalle que estamos describiendo. Puede observarse el uso de la tilde (~) como elemento separador de las palabras utilizadas en la descripción del fichero. Este es un tema relacionado con los catcodes o *Categorías de códigos*. En régimen $\text{\LaTeX}2\epsilon$ (realmente, en $\text{\TeX}$) y en régimen `expl3` casi todos los códigos de caracteres comparten las mismas categorías con algunas notables e importantes excepciones. Por ejemplo, el espacio tiene categoría 10 en $\text{\TeX}$ y categoría 9 en `expl3`, lo que significa que se ignoraría. Así, si no hubiésemos escrito

```
{PYST~document~types~and~departments~system}
```

al cargarse el fichero, en el `.log` aparecería en realidad

```
{PYSTdocumenttypesanddepartmentssystem}
```

Y lo mismo ocurriría, por ejemplo, en el nombre del departamento en su declaración. *Teoría de la Señal* se leería como *TeoriadelaSeñal*, sin separación alguna. La solución habitual que consiste en escribir con tilde (~) *Teoría~de~la~Señal* en lugar de cada espacio, funciona, pero es incómoda y fácil de olvidar en un fichero que no es un fichero de código, sino una lista de nombres, direcciones y demás.

La solución que hemos adoptado para resolver esta incomodidad consiste en leer el fichero `pyst-depts.cfg` en un régimen $\text{\LaTeX}2\epsilon$ dentro de `pyst-options.ltx` donde *un espacio* vuelve a ser catcode 10; esto es, *un espacio*. Así, hemos definido un comando propio `\pyst_dept_input_twoc:n`, que mostramos en el siguiente código.

```
\cs_new_protected:Npn \pyst_dept_input_twoc:n #1
{ \ExplSyntaxOff \input { #1 } \ExplSyntaxOn }
```

En esta definición, se hace uso del comando `\cs_new_protected:Npn`, del módulo `cs` (*control sequence*), que se analizará más adelante. Por ahora, nos basta entender que equivale al `\NewDocumentCommand` pero para funciones internas del sistema en lugar de comandos de usuario. En este caso, el mecanismo de la función evita el régimen $\text{\LaTeX}3$, utilizando `\ExplSyntaxOff`, abre el fichero que se le pasa como parámetro, `\input { #1 }` y restaura el régimen $\text{\LaTeX}3$, `\ExplSyntaxOn`.

Dentro de `pyst-options.ltx` la lectura se realiza mediante el código mostrado a continuación.

```
\file_if_exist:nTF { pyst-depts.cfg }
{ \pyst_dept_input_twoe:n { pyst-depts.cfg } }
{ \msg_warning:nn { pyst } { depts-cfg-missing } }
```

De nuevo, una función propia de `expl3`, `\file_if_exist:nTF` con un significado que no necesita explicación. Su firma `nTF` revela que espera un nombre entre llaves, `n`, en este caso, el nombre de un fichero. Si existe (`T`), se ejecuta el primer bloque del código y si no existe (`F`) el segundo, en este caso, un mensaje, indicando que el fichero correspondiente no se ha encontrado.

2.4 Las variables del tipo secuencias de los elementos registrados

Una vez que se han creado las dos variables globales del tipo *property list* obtenidas tras la lectura de los catálogos de configuración, es importante obtener dos *listas*, otro tipo de variable de `expl3` que permitan conocer que tipos de documentos existen y qué departamentos se han configurado.

Estas dos listas, denominadas respectivamente

- `\g_pyst_available_types_seq`,
- `\g_pyst_available_depts_seq`,

existen únicamente para poder *conocer* lo que hay disponible. No se rellenan dentro de las funciones `\DeclareDocumentType`/`\DeclareDepartment` y se construyen aparte, después de leer los ficheros completos, mediante el siguiente bloque de código:

```
\seq_gclear:N \g_pyst_available_types_seq
\prop_map_inline:Nn \g_pyst_types_prop
{ \seq_gput_right:Nn \g_pyst_available_types_seq {#1} }

\seq_gclear:N \g_pyst_available_depts_seq
\prop_map_inline:Nn \g_pyst_depts_config_prop
{ \seq_gput_right:Nn \g_pyst_available_depts_seq {#1} }
```

Podría pensarse que estas listas deberían haberse formado en el momento de creación de la *listas de propiedades* correspondientes, leyendo directamente los ficheros. Sin embargo, hay dos razones para hacerlo una vez que las listas ya existen utilizando estas listas de propiedades.

La primera y más obvia es que una lista derivada siempre estará sincronizada con el catálogo real, porque *es* el catálogo, visto de otra forma.

La segunda es más sutil y es la que de verdad obliga a hacerlo así. Debemos evitar que aparezcan *duplicados* en los tipos o departamentos disponibles. Las claves de una lista de propiedades son únicas, así que si `pyst-types.cfg` o `pyst-depts.cfg` declarase el mismo tipo/departamento dos veces, la segunda declaración pisaría a la primera y la lista de propiedades correspondiente conservaría una sola entrada. Sin embargo, si la secuencia se rellena dentro de las propias funciones `\DeclareDocumentType`/`\DeclareDepartment`, ese tipo/departamento

aparecería *dos veces* en las listas de tipos y departamentos disponibles, aunque la lista de propiedades solo tuviese una entrada para cada clave.

Como contrapartida `\prop_map_inline:Nn` recorre la lista de propiedades en un orden que `expl3` deja explícitamente sin definir, así que estas listas no conservan el orden de declaración del fichero `.cfg` correspondiente. Este problema es puramente cosmético puesto que solo afecta al orden en el que se listan los tipos en un mensaje, por ejemplo, pero conviene conocer este hecho para no apoyar ninguna lógica en el orden en el que aparecen las claves en la lista de propiedades y en la secuencia correspondiente.

3 Definición de los espacios de claves

Hemos visto en el Apartado anterior tres de las posibles vías en la que declaran claves y valores para poder construir los documentos en el sistema PYST-PUB. Falta la cuarta posible fuente que sería la asignación que el usuario hace en `\documentclass`. Las cuatro vías comparten, como es habitual en `LATEX` el mecanismo común de `[clave=valor]`. Sin embargo, aún no sabemos como utilizar esos valores asignados a esas claves dentro de nuestro sistema.

En el régimen `expl3` esa utilización se realiza mediante un procedimiento que tiene lugar en varias fases. En la primera, que es la que vamos a analizar en este apartado, se construyen los denominados `namespaces` o espacios de claves o espacios de nombres,

Como dijimos en la introducción, la fase PRE se construye con la lectura secuencial de los cuatro ficheros que la forman, el segundo de los cuales es `pyst-configuration.ltx`. Por lo tanto, siguiendo con la lectura de su contenido, veamos a continuación cómo se construye el núcleo en el que se basa el sistema PYST-PUB. Y este núcleo, todo lo que PYST-PUB sabe interpretar, la configuración del documento, los campos de un departamento, etc, se construye alrededor de un conjunto de claves y procedimientos asociados que se definen en dos `namespaces`⁴ o espacios de claves o espacios de nombres, declarados con el comando `\keys_define:nn`, con los nombres `pyst` y `pyst/dept`.

Estos espacios de nombres es lo que permite, entre otras cosas, que utilizar los datos que definen un departamento se puede realizar con una sola clave o que el usuario pueda sobrescribir un campo de ese departamento cuando desee, sin que el sistema tenga que tener en cuenta dónde se realiza esa acción.

Antes de entrar de lleno en el análisis de los `namespaces`, conviene revisar brevemente cómo se ha llegado a este diseño. Inicialmente, el sistema PYST-PUB utilizaba seis `namespaces` diferentes, cada uno con un contenido específico, utilizado para establecer las diferentes funciones que se le asignaban a cada una de las claves que se definían en el sistema. Sin embargo, después de la versión 1.1c se consideró que el sistema era enormemente complejo, con continuos saltos adelante y atrás; se construían estructuras que sólo eran utilizadas mucho más tarde en el flujo del programa. Por eso, se planteó reconstruir nuevamente el sistema de claves del sistema de una manera mucho más lineal.

En las primeras revisiones se planteó crear un único espacio de claves. Sin embargo, pronto se vio que este único espacio generaba un problema para el que era difícil encontrar solución,

⁴En el contexto de `expl3`, un `namespace` es un prefijo de texto que agrupa un conjunto de claves relacionadas y acciones asociadas a las mismas. Sirve para aislar esos conjuntos de claves entre sí, permitiendo que una misma palabra (clave) (por ejemplo, `doc-type`) ejecute instrucciones completamente distintas dependiendo del `namespace` desde el cual el sistema la esté consultando, evitando colisiones.

ya que un usuario podía declarar en un tipo de documento claves correspondientes a los datos institucionales de un departamento y/o en la definición de un departamento podían escribirse claves que definían un tipo de documento y ese procedimiento “malévolo” *no era detectable*.

Este problema nos ha hecho llegar a esta solución de la versión 2.0 que consideramos estable y a salvo de ese problema indetectable de un procedimiento inadecuado por parte de un usuario. Adicionalmente, hemos evitado una fragilidad del sistema protegiéndonos ante posibles cambios del núcleo de $\text{\LaTeX}$; fragilidad detectada y resuelta a costa de modificar la forma en la que la fase PRE del sistema PYST-PUB se comunicaba con el sistema KOMA, manteniendo la estructura envolvente del primero.

Aclarado este punto, analicemos en primero lugar la manera en la que `expl3` permite definir el sistema de claves utilizados en los desarrollos modernos de $\text{\LaTeX}$.

3.1 Anatomía del comando `keys_define:nn`

Antes de recorrer las claves utilizadas en PYST-PUB, conviene detenerse en la función que las crea, porque sobre ella se basa toda esta sección. `\keys_define:nn` pertenece al módulo `keys`, el sistema de claves de `expl3`, conocido también como `l3keys`, y su firma `nn` indica que espera dos argumentos normales, ambos entre llaves:

```
\keys_define:nn {<nombre del espacio>} {<declaración de claves>}
```

El primero es el nombre del *namespace* o espacio de claves, `pyst` y `pyst / dept`, en nuestro caso, que actúa como un pronombre: dos claves llamadas igual en espacios distintos son claves distintas y no se estorban. El segundo es la lista de claves propiamente dicha, escrita también como un conjunto de `clave=valor`, con una peculiaridad: lo que aparece a la izquierda del signo igual no es el nombre de una clave sino el nombre *seguido de un punto y una propiedad*. Lo que hay a la derecha del signo igual dependerá de la propiedad. Por ejemplo, en el namespace `pyst` nos encontramos con la siguiente declaración:

```
doc-style .code:n = { ...qué hacer... }
```

La clave será `doc-style`. Ese `.code:n` es la *propiedad* de la clave. `l3keys` ofrece muchas propiedades aunque en PYST-PUB solo aparecen tres y conviene distinguirlas bien porque dos de ellas se confunden con facilidad:

- `.code:n` La más general y la que predomina en el fichero `pyst-options.ltx`. Su significado es *cuando alguien le asigna un valor a esta clave, ejecuta el código que hay a la derecha del signo igual*. El valor que el usuario haya dado a la clave llega al código como `#1`, exactamente igual que en un `\NewDocumentCommand`. Es, en la práctica, definir un comando en miniatura cuyo nombre es la clave.
- `.default:n` El valor que la clave toma si el usuario la escribe *sin* valor. Que `math` lleve `.default:n = true` es lo que permite escribir `math` a secas en `\documentclass[...]{pyst-pub}` en lugar del más incómodo `math=true`. Si nadie menciona la clave, no ocurre nada.
- `.initial:n` El valor que se aplica *en el momento mismo de definir la clave*, aunque nadie la escriba jamás. No es un valor «de reserva» que espere a ser necesitado: se ejecuta ahí y entonces, durante la lectura del propio `pyst-options.ltx`.

La diferencia entre las dos últimas es la fuente de malentendidos más habitual, y merece fijarla con un ejemplo del propio fichero:

```
font .code:n = { \tl_gset:Nn \g_pyst_font_tl { #1 } } ,
font .initial:n = stixtwo ,
math .code:n = { \pyst_set_bool:Nnn \g_pyst_module_math_bool
                { math } { #1 } } ,
math .default:n = true ,
```

Si el usuario no escribe nada en absoluto, `\g_pyst_font_tl` ya contiene `stixtwo`, porque `.initial:n` se disparó al leerse el fichero, mientras que el booleano de `math` conserva el valor que le diera el tipo de documento, porque `.default:n` no llegó a entrar en juego. En una frase: `.initial:n` es el *valor de fábrica de la variable*; `.default:n`, el *valor implícito de la clave*.

Obsérvese, además, que una misma clave puede recibir varias propiedades en líneas sucesivas, como hacen `font` y `math` arriba. No son dos claves distintas: son dos declaraciones sobre la misma.

Una última observación sobre `\keys_define:nn` que conviene tener presente al leer el fichero: es *acumulativa*. Cada llamada añade claves al espacio indicado sin borrar las que ya hubiera; declarar dos veces la misma clave, en cambio, sobrescribe la anterior en silencio. PYST-PUB aprovecha lo primero para declarar los espacios de una sola vez, en un único bloque, pero nada impediría repartirlo en varios.

3.2 El namespace pyst

A continuación vamos a analizar las claves definidas en este espacio de claves agrupadas por lo que hacen, no por el orden en que aparecen en el fichero.

3.2.1 Las claves que consume el pre-escaneo

Estudiemos en primer lugar dos claves que determinan que *tipo* de documento se desea construir y si se va a realizar la declaración o no de un departamento. El papel central de estas claves aconseja realizar con ellas un tratamiento específico y previo, como se verá en el [Apartado 5.3](#). Por ello, en la definición del espacio de nombre `pyst`

```
doc-type .code:n = { } ,
dept-name .code:n = { } ,
```

tienen un cuerpo vacío (`no-op` en la terminología `expl3`). No es un descuido: su valor real ya se ha leído y aplicado *antes* de que el flujo normal llegue a este espacio de claves. Cuando el sistema procesa de forma global el espacio `pyst` no hay nada más que hacer.

3.2.2 Las claves que configuran el documento

A continuación nos encontramos un conjunto de claves que configuran las características más notables del documento,

```
base-class .code:n =
```

```

{ \pyst_guard_reserved:nn { base-class }
{
  \str_case:nnF { #1 }
  {
    { scrbook } { \tl_gset:Nn \g_pyst_base_class_tl { #1 } }
    { scrreprt } { \tl_gset:Nn \g_pyst_base_class_tl { #1 } }
    { scrartcl } { \tl_gset:Nn \g_pyst_base_class_tl { #1 } }
  }
  {
    \msg_warning:nnnnnn { pyst } { invalid-choice }
    { base-class } { #1 } { scrbook,~scrreprt,~scrartcl } { scrartcl
  }
  \tl_gset:Nn \g_pyst_base_class_tl { scrartcl }
}
} } ,

```

La envoltura `\pyst_guard_reserved:nn` que rodea el comando `\str_case:nnF` se explica más adelante (Apartado 4); por ahora basta con saber que decide *cuándo* se ejecuta este código, no *qué* hace.

La clave `base-class` es la única clave del namespace que valida contra una lista cerrada escrita en línea, sin delegar en un ayudante externo. Solo tres valores son válidos, las tres clases que KOMA-Script ofrece para libros y artículos.

La función que hace posible esta validación es `\str_case:nnF`, del *módulo* `str`, que gestiona *cadenas de caracteres*. Conviene no confundir `str` con `tl`: ambos guardan texto, pero una `tl` conserva los *tokens* tal como se escribieron, con sus catcodes mientras que una `str` los reduce a caracteres corrientes. Para comparar dos palabras, que es lo que aquí interesa, la comparación de cadenas es la robusta: no le afecta cómo se escribiera el valor.

Su firma `nnF` descompone así:

- n** La cadena a examinar, entre llaves: aquí `#1`, el valor que el usuario haya escrito.
- n** Una lista de *casos*, también entre llaves, formada por pares consecutivos: primero un valor posible, y a continuación el código a ejecutar si coincide. En el ejemplo hay tres pares, `{ scrbook } / asignar, { scrreprt } / asignar, { scrartcl } / asignar`. En cuanto uno coincide, se ejecuta su código y los demás se ignoran.
- F** El sufijo `F` añade una rama *false* final: el código que se ejecuta si **ninguno** de los casos anteriores encajó.

Ese sufijo pertenece a una convención general de `expl3` que reaparecerá muchas veces en esta nota, así que conviene fijarla ahora. Las funciones que hacen una pregunta admiten tres terminaciones:

- TF** Espera dos bloques de código: el primero se ejecuta si la respuesta es cierta; el segundo, si es falsa.
- T** Espera uno solo, que se ejecuta *únicamente* si la respuesta es cierta. Si es falsa, no ocurre nada.
- F** El simétrico: un solo bloque, que se ejecuta *únicamente* si la respuesta es falsa.

Elegir una u otra no es cuestión de estilo: es declarar qué casos interesan. `\str_case:nnF` usa `F` porque el valor “cierto” (coincide) ya está cubierto por la lista de casos, y lo único que falta es describir qué hacer cuando no existe coincidencia.

La rama `F` de la comparación contiene dos instrucciones. La primera es `\msg_warning:nnnnnn`, del módulo `msg` que ya vimos con `\msg_info:nnn`: seis argumentos `n` que se corresponden, en orden, con el *módulo* que emite el aviso (`pyst`), el *nombre* del mensaje (`invalid-choice`) y los cuatro datos que ese mensaje necesita para redactarse, la clave afectada, el valor inválido, la lista de valores admitidos y el valor de repliegue. Nótese la técnica: el texto del aviso no está aquí, sino declarado aparte en `pyst-core.ltx`; aquí solo se nombra y se le suministran los huecos que ha de rellenar.

La segunda instrucción de la rama es la que importa para el resultado, `\tl_gset:Nn`, que asigna `scrartcl`. Es decir, ante un valor que no encaja el sistema PYST-PUB no aborta, avisa y *repliega* hacia un estado conocido, en lugar de detener la compilación por una errata o, peor aún, de continuar con una variable vacía. Puesto que esta variable hace referencia a la clase que KOMA va a utilizar, esto daría lugar a un error posterior. En este caso, se ha preferido realizar un aviso en la consola de compilación y no detener ese proceso. Si el resultado (la clase) no es la deseada, el usuario actuará en consecuencia.

El resto de las claves de este bloque está formado por las siguientes:

```
doc-style .code:n = { \tl_gset:Nn \g_pyst_doc_style_tl { #1 } } ,
doc-style .initial:n = ,
coverstyle .code:n = { \pyst_coverstyle_normalize:n { #1 } } ,
coverstyle .initial:n = ,
language .code:n =
{
  \str_case:nnF { #1 }
  {
    { spanish } { \tl_gset:Nn \g_pyst_language_tl { spanish } }
    { english } { \tl_gset:Nn \g_pyst_language_tl { english } }
  }
  {
    \msg_warning:nnnnnn { pyst } { invalid-choice }
    { language } { #1 } { spanish,~english } { spanish }
    \tl_gset:Nn \g_pyst_language_tl { spanish }
  }
} ,
font .code:n = { \tl_gset:Nn \g_pyst_font_tl { #1 } } ,
font .initial:n = stixtwo ,
babel .code:n = { \pyst_guard_reserved:nn { babel }
                 { \clist_gset:Nn \g_pyst_babel_clist { #1 } } } ,
```

`doc-style` y `font` son claves de asignación directa: toman el valor `#1` y lo guardan en las correspondientes variables globales. guardarlo, sin más. La propiedad `\initial:n` de `doc-style` y `font` ya ha sido explicada anteriormente.

La clave `coverstyle` delega en un *comando de validación* propio (Ver [Apartado 8](#)) en vez de asignar sin más su valor, puesto que este valor necesita normalizarse. La clave `coverstyle` acepta tres formas equivalentes (un nombre, *minimo*, el nombre del fichero en el que se define,

`minimo.def`, y `none` o vacío para “sin portada”) y rechaza cualquier valor que contenga una ruta; las portadas se encuentran en una única carpeta del sistema.

Podemos observar que la clave `language` recibe un tratamiento similar al efectuado para la clave `base-class` aunque, en este caso no existe una reserva de asignación. Se permiten únicamente los valores `spanish` y `english`, con el valor `spanish` en caso de no ser ninguno de los dos.

La clave `babel` es de asignación directa, como `doc-style`, pero con la salvedad de estar también envuelta en `\pyst_guard_reserved:nn` y que su valor no se guarda en una variable de texto global sino en una variable del tipo `clist`, `\g_pyst_babel_clist`.

3.2.3 Las claves de activación

A continuación se declaran seis claves que vamos a catalogar como *claves de activación*, ya que sólo pueden tomar valores verdadero o falso.

```
math          .code:n      = { \pyst_set_bool:Nnn
                        \g_pyst_module_math_bool { math } { #1 } } ,
math          .default:n = true ,
graphics      .code:n      = { \pyst_set_bool:Nnn
                        \g_pyst_module_graphics_bool { graphics } { #1 } } ,
graphics      .default:n = true ,
advanced-math .code:n      = { \pyst_set_bool:Nnn
                        \g_pyst_module_advanced_math_bool { advanced-math } { #1 } }
,
advanced-math .default:n = true ,
debug         .code:n      = { \pyst_set_bool:Nnn
                        \g_pyst_debug_bool { debug } { #1 } } ,
debug         .default:n = true ,
compat        .code:n      = { \pyst_set_bool:Nnn
                        \g_pyst_compat_bool { compat } { #1 } } ,
compat        .default:n = true ,
copyright     .code:n      = { \pyst_set_bool:Nnn
                        \g_pyst_copyright_bool { copyright } { #1 } } ,
copyright     .default:n = true ,
```

Las seis claves comparten un único comando de validación, `\pyst\set\bool:Nnn`, definido en `pyst-core.ltx`. Su firma `Nnn` anuncia sus tres argumentos: la variable booleana de destino (`N`, sin llaves), el *nombre* de la clave, que solo sirve para poder citarlo si hay que avisar de un error, y el valor que escribió el usuario. Merece la pena abrirlo, porque introduce varias expresiones e ideas nuevas:

```
\cs_new:Npn \pyst_set_bool:Nnn #1 #2 #3
{
  \exp_args:Ne \str_case:nnF { \str_lowercase:n { #3 } }
  {
    { true } { \bool_gset_true:N #1 }
    { false } { \bool_gset_false:N #1 }
    { 1 } { \bool_gset_true:N #1 }
    { 0 } { \bool_gset_false:N #1 }
```

```

    { yes }    { \bool_gset_true:N #1 }
    { si }     { \bool_gset_true:N #1 }
    { sí }     { \bool_gset_true:N #1 }
    { no }     { \bool_gset_false:N #1 }
    { on }     { \bool_gset_true:N #1 }
    { off }    { \bool_gset_false:N #1 }
  }
  {
    \msg_error:nnnn { pyst } { invalid-bool } { #2 } { #3 }
    \bool_gset_false:N #1
  }
}

```

La primera es el uso del comando `\cs\new:Npn`, del módulo `cs` (*control sequence*), que es la forma de `expl3` de *definir una función nueva*, el equivalente al `\NewDocumentCommand`, pero para funciones internas del sistema en lugar de comandos de usuario. Su firma `Npn` se lee así: `N`, el nombre de la función que se está creando; `p` (de *parameter text*), el texto de parámetros, aquí `#1 #2 #3`, porque la función toma tres parámetros y `n`, el cuerpo entre llaves.

La firma `p` merece una reflexión adicional. Sirve principalmente para dos cosas:

Declaración explícita Avisa al compilador de que debe esperar un patrón como `#1 #2 #3` inmediatamente antes del bloque `{ ... }`.

Sintaxis avanzada con delimitadores Permite definir parámetros con separadores personalizados estilo TeX tradicional (por ejemplo, `#1,#2 \q_stop`), algo que la versión simplificada sin `p`, `\cs_new:Nn`, no permite porque asume únicamente argumentos estándar.

Si no se incluye la firma `p` usando `\cs_new:Nn`, se prescinde de esa declaración manual y se deja que `expl3` inspeccione automáticamente el sufijo de la función `Nnn` para saber qué argumentos esperar.

En segundo lugar podemos ver en este código la aparición de funciones del módulo `bool`, que gestiona variables *booleanas*, las que solo pueden valer cierto o falso. `\bool\gset\true:N` y `\bool\gset\false:N` fijan ese valor, y su firma es la más simple posible: una sola `N`, el nombre de la variable a modificar. El `g` de `gset` vuelve a significar *global*, igual que en `\tl\gset:Nn`: el valor sobrevive al grupo en el que se fijó.

Por último, un hecho que vemos por primera vez, el uso del comando `\exp\args:Ne`, y es de las construcciones que más desconciertan cuando se ven por primera vez. Aprovechemos este ejemplo para entender su uso.

Ya hemos descrito el comando `\str\case:nnF`, que compara su primer argumento *tal como se lo dan, entre llaves, sin evaluarlo, con el segundo, entregado también entre llaves*. En esta ocasión, queremos comparar la conversión a minúsculas de un parámetro con un conjunto de cadenas de caracteres.

En este caso sin embargo, si le entregásemos directamente a `\str\case:nnF` la cadena `\str\lowercase:n{ #3 }` como primer parámetro, compararía esa llamada, literalmente, esa cadena de caracteres (tokens), con `true`, `false` y el resto de valores. Evidentemente, no coincidiría jamás. Por lo tanto, hace falta que la conversión a minúsculas que constituye el primer parámetro de la función `\str\case:nnF` ocurra *antes* de realizar esa comparación.

Eso es lo que hace la familia `\exp\args:N...`. No aporta lógica propia, sino que *prepara* los argumentos de otra función antes de pasárselos. Su firma se lee de izquierda a derecha, una letra por argumento de la función envuelta:

- N** El primer argumento, aquí, la propia función `str\case:nnF`, se pasa sin tocar.
- e** El segundo (que a su vez es el primer argumento de la función `str\case:nnF`) se *expande por completo* antes de entregarlo. **e** viene de *exhaustive*: se evalúa hasta el final, de modo que `\str\lowercase:n{TRUE}`, por ejemplo, llega convertido ya en `true`.

Los especificadores de expansión son un vocabulario pequeño que conviene tener a mano, porque reaparecen constantemente:

- n, N** Literal: se pasa tal cual, sin evaluar nada.
- V** El *valor* de la variable que se le da, no su nombre.
- o** Una única expansión, un solo paso.
- e, x** Expansión completa, hasta el final.

La consecuencia práctica de todo ello es que `pyst\set\_bool:Nnn` acepta diez formas para dos valores — `true/false`, `1/0`, `yes/no`, `si/sí`, `on/off` y en cualquier combinación de mayúsculas y minúsculas, porque todo se normaliza antes de comparar. Si aun así nada encaja, la rama **F** avisa con `invalid-bool` y deja el booleano en `false`: fallar hacia un estado conocido, otra vez, en vez de dejar la variable como estuviera.

La propiedad `\default:n` en las seis claves es lo que permite escribir `math` a secas en vez del más incómodo `math=true`, por ejemplo, o `copyright`, con el significado equivalente `copyright=true`.

3.2.4 Claves de configuración estructural aditivas

Cuatro claves del namespace `pyst` no asignan su valor a una variable fija: lo *encauzan*, a través de tres funciones propias, hacia una de dos listas posibles⁵ según en qué fase de lecturas de las claves nos encontremos, distinguiendo, fundamentalmente, la correspondiente a la carga de los valores definidos por el usuario a través del comando `\documentclass[...]{pyst-pub}`. Por ello, antes de mirar las cuatro claves, conviene entender el mecanismo que decide esa “fase”, porque es compartido por todas ellas.

El conmutador de fase: `\g_pyst_phase_user_bool`

`\g_pyst_phase_user_bool` es un booleano global, declarado en `pyst-core.ltx` con `\bool_new:N` y, por tanto, en falso desde su creación, que es el valor por defecto de toda variable booleana recién declarada. Se levanta una única vez, con `\bool_gset_true:N`, en un punto muy concreto del fichero `pyst-options.ltx`:

```
% --- Bloque 6: opciones propias del usuario -----
\bool_gset_true:N \g_pyst_phase_user_bool
\cs_if_exist:NTF \@raw@classoptionslist
```

⁵De manera aditiva, de ahí el nombre que hemos utilizado para agruparlas.

```

{
  \keys_set:nV { pyst } \@raw@classoptionslist
  \keys_set:nV { pyst / dept } \@raw@classoptionslist
}
{
  \keys_set:nV { pyst } \@classoptionslist
  \keys_set:nV { pyst / dept } \@classoptionslist
}

```

Es decir: justo antes de procesar la lista cruda de opciones que el usuario escribió en `\documentclass`. Hasta ese instante, la bandera permanece baja mientras se aplican, por este orden, los valores por defecto, el tipo de documento y, si se asignó uno, el departamento. Y una vez levantada, *no se vuelve a bajar*: no hay ningún `\bool_gset_false:N` posterior sobre ella en todo el fichero.

Ese diseño, levantar una sola vez, sin retorno, es lo que convierte a la variable en un simple indicador de *fase*: mientras está en falso, quien escribe es el sistema (opciones por defecto, tipo de documento o el departamento, si es que se ha declarado uno); en cuanto pasa a verdadero, quien escribe es el usuario (si no tenemos en cuenta que el usuario es también quien ha podido declarar un valor a la clave `dept-name`), y así se mantiene durante el resto de la lectura de claves. Cualquier función que necesite distinguir «¿esto lo trajo el tipo o lo trajo el usuario?» solo tiene que consultarla.

Las cuatro claves

Con el conmutador ya entendido, el namespace declara así las cuatro claves que lo consultan dentro de los comandos ejecutados como propiedad de las claves:

```

modules      .code:n = { \pyst_modules_route:n { #1 } } ,
doc-options   .code:n = { \pyst_koma_route:n { #1 } } ,
koma-layout   .code:n =
{
  \bool_gset_true:N \g_pyst_koma_layout_bool
  \pyst_koma_route:n { #1 }
} ,
geometry-layout .code:n = { \pyst_geometry_route:n { #1 } } ,

```

Tres comandos, `\pyst_koma_route:n`, `\pyst_geometry_route:n` y `\pyst_modules_route:n`, atienden a las cuatro claves. Dos de las claves comparten la misma función; las otras dos tienen la suya propia. Analicemos en primer lugar las que comparten el comando.

Las claves `doc-options`, `koma-layout` y `geometry-layout`

Las dos primeras claves utilizan el comando `\pyst\koma\route:n` como parte de la propiedad de las mismas. Su código se muestra a continuación.

```

\cs_new_protected:Npn \pyst_koma_route:n #1
{
  \bool_if:NTF \g_pyst_phase_user_bool

```

```

    { \clist_gput_right:Nn \g_pyst_user_opts_clist { #1 } }
    { \clist_gput_right:Nn \g_pyst_type_opts_clist { #1 } }
  }

```

Vemos que el comando `\bool\if:N` consulta el conmutador, de acuerdo con su firma `N`, y el sufijo `TF` anuncia las dos ramas, cierta y falsa y, según su valor, añade el valor de la clave, `#1` a una de dos listas. Ambas son del módulo `clist` (*comma list*), el tipo natural para acumular opciones de KOMA, porque es exactamente el formato que la clase base espera recibir.

La incorporación a las listas se realiza con el comando `\clist\gput\right:Nn`, `g` de asignación global, `put_right` de *añadir por la derecha*, es decir, al final de la lista, preservando lo que ya hubiera es decir, una operación de acumular, no la de sustituir; de ahí que estos claves realicen una configuración estructural *aditiva*.

Debemos observar que la propiedad asociada a la clave `koma-layout` antes de encauzar su valor asigna a la variable booleana `\g_pyst_koma_layout_bool` el valor `true`. Esta variable la consulta `pyst-build.ltx` para detectar que estrategia de configuración del layout, la propuesta armónica, a cargo de `typearea` de la clase KOMA, o una estrategia basada en la optimización de la superficie, si el usuario ha elegido un layout gestionado por `geometry`. El criterio establecido en el sistema PYST-PUB en caso de conflicto (el usuario ha asignado valores a las dos claves de layout, `koma-layout` y `geometry-layout` *se resuelve siempre a favor del layout gestionado por el paquete geometry*). Esa misma propiedad asigna, además, el valor `true` a una segunda variable booleana `\g_pyst_koma_layout_user_bool` pero solo cuando `\g_pyst_phase_user_bool` está activo.

```

koma-layout      .code:n =
{
  \bool_gset_true:N \g_pyst_koma_layout_bool
  \bool_if:NT \g_pyst_phase_user_bool
    { \bool_gset_true:N \g_pyst_koma_layout_user_bool }
  \pyst_koma_route:n { #1 }
} ,

```

La razón de que exista esta segunda variable es una consecuencia directa del canal compartido. `koma-layout` y `doc-options` escriben en las mismas dos listas, de modo que, mirando esas listas, es imposible saber si un `DIV=12` vino de una u otra clave, ni si lo puso el tipo o el usuario. `geometry-layout` no tiene ese problema porque conserva su propia lista de usuario separada. El valor de esta variable `\g_pyst_koma_layout_user_bool` es lo que le faltaba asignar a `koma-layout` para que el informe de depuración pueda contestar la pregunta *¿el layout activo viene del tipo o lo redefinió el usuario?* sin reabrir el canal compartido.

Adviértase dónde se levanta: en el `.code:n` de la clave, no dentro de `\pyst_koma_route:n`. Es deliberado. El enrutador lo comparten `koma-layout` y `doc-options`, así que levantarla ahí la contaminaría: un `doc-options=` cualquiera del usuario haría creer al sistema que hubo una redefinición de layout que nunca ocurrió. Solo la clave sabe cuál de las dos se está procesando.

En el caso de `geometry-layout`, el comando que recibe el valor es `\pyst_geometry_route:n`, que sigue exactamente el mismo patrón que `\pyst\koma\route:n`, como se muestra a continuación.

```

\cs_new_protected:Npn \pyst_geometry_route:n #1

```

```
{
  \bool_if:NTF \g_pyst_phase_user_bool
  { \clist_gput_right:Nn \g_pyst_geometry_layout_user_clist { #1 } }
  { \clist_gput_right:Nn \g_pyst_geometry_clist { #1 } }
}
```

El único cambio efectuado es que opera sobre su propio par de listas: `\g_pyst_geometry_clist` para el tipo, `\g_pyst_geometry_layout_user_clist` para el usuario.

La clave `modules`

En el sistema PYST-PUB la clave `modules` se utiliza para gestionar la incorporación de módulos especiales utilizados solo en algunos tipos de documentos. En la declaración del namespace `pyst` el valor de la clave se asigna al comando `\pyst\modules\route:n`, definido en el siguiente bloque.

```
\cs_new_protected:Npn \pyst_modules_route:n #1
{
  \bool_if:NTF \g_pyst_phase_user_bool
  { \seq_gset_split:Nnn \g_pyst_user_extra_modules_seq { , } { #1 } }
  { \seq_gset_split:Nnn \g_pyst_user_modules_seq { , } { #1 } }
}
```

Una `seq` es una lista ordenada de elementos, más parecida a un vector que a una cadena de texto. `\seq_gset_split:Nnn` toma una cadena, la parte por un separador — aquí la coma — y deposita los trozos como elementos independientes de `\g_pyst_user_extra_modules_seq` o `\g_pyst_user_modules_seq`, según la misma consulta al conmutador. Nótese que es `gset`, no `gput`: cada llamada *sustituye* el contenido de la secuencia en lugar de añadirse a él, porque cada canal se declara de una sola vez.

Por qué dos listas y no una

¿Por qué mantener dos listas por canal, en vez de una sola donde todo se acumula según llega? Porque más adelante, en `pyst-build.ltx`, hace falta fundirlas *en un orden concreto*: primero todo lo que aportó el tipo, después todo lo que escribió el usuario. KOMA resuelve una clave repetida quedándose con la *última* ocurrencia, así que colocar al usuario en segundo lugar es, literalmente, lo que le da la última palabra. Si todo se mezclara en una única lista sin distinguir el origen, esa garantía de orden desaparecería.

Los módulos son la única excepción a esa fusión: sus dos secuencias no se funden, sino que se cargan *ambas*, la del tipo y la del usuario, porque declarar un módulo adicional es una operación aditiva, no una sustitución.

Listas y secuencias. ¿Porqué la asimetría de los comandos anteriores?

A las cuatro claves de este bloque se les asigna en cualquier lugar del sistema PYST-PUB un mismo tipo de valor, una cadena de texto que puede contener llaves y elementos separados por comas. Sin embargo los valores de la clave `modules` se acumulan en una variable de secuencia

(seq) mientras que el resto de las claves, `doc-options`, `koma-layout` y `geometry-layout` acumulan sus valores en una `clist`. Esa asimetría no es un descuido de estilo, refleja que las tres claves resuelven *problemas distintos* y conviene verlo hasta el final hasta el proceso que consume esos valores y no quedarnos solo en cómo se definen, porque es ahí donde se justifica la elección.

Las listas de `koma` y `geometry` terminan, en el fichero `pyst-build.ltx`, convertidas en una *única cadena de texto* mediante `\clist_use:Nn`, entregada literalmente a un solo comando externo:

```
\clist_if_empty:NF \g_pyst_type_opts_clist
  { \clist_put_right:NV \g_pyst_final_opts_clist \g_pyst_type_opts_clist }
\clist_put_right:No \g_pyst_final_opts_clist { \g_pyst_user_opts_clist }
...
\tl_const:Nx \PystClassOptions { \clist_use:Nn \g_pyst_final_opts_clist { ,
  } }
...
\clist_new:N \l_pyst_geometry_merged_clist
\clist_concat:NNN \l_pyst_geometry_merged_clist
  \g_pyst_geometry_clist \g_pyst_geometry_layout_user_clist
\tl_const:Nx \PystGeoConfig { \clist_use:Nn
  \l_pyst_geometry_merged_clist { , } }
```

`\clist_use:Nn` une los elementos con el separador que se le indique, aquí la coma, y produce exactamente el *keyval* que `\KOMAOPTIONS{...}` y `\geometry{...}` esperan recibir.

Los módulos resuelven un problema distinto. En `pyst-modules.ltx`, ninguna de las dos secuencias se convierte jamás en cadena, ambas se *recorren*, elemento a elemento, para ejecutar una acción por cada uno a través del comando siguiente.

```
\cs_new_protected:Npn \pyst_load_one_module:n #1
{
  \file_if_exist:nTF { modules/#1.ltx }
  { \file_input:n { modules/#1.ltx } ... }
  { \msg_warning:nnn { pyst } { file-not-found } { modules/#1.ltx } }
}

\seq_map_function:NN \g_pyst_user_modules_seq
  \pyst_load_one_module:n
\seq_map_function:NN \g_pyst_user_extra_modules_seq
  \pyst_load_one_module:n
```

`\seq_map_function:NN` aplica una función (aquí `\pyst_load_one_module:n`) a cada elemento de la secuencia por separado. No hay, en ningún punto de `pyst-modules.ltx`, una necesidad de tener todos los módulos, unidos por comas, como una sola cadena. Lo que hace falta es la acción, repetida una vez por cada uno de los módulos, de comprobar que `modules/<nombre>.ltx` existe, cargarlo, y registrarlo y eso es precisamente lo que una *seq* recorrida con `\seq_map_function:NN` ofrece de forma directa.

Dicho de otro modo, se han generado variables de distintos tipos porque para cada una de ellas se realizan operaciones diferentes. Si el destino es *un argumento, una vez*, se utiliza una

variable del tipo `clist`. Si el destino es *una acción, por cada elemento*, la variable elegida es del tipo `seq`. La asimetría existe porque *ahorra un trabajo posterior*.

3.3 Las claves del namespace `pyst/dept`

En el namespace `pyst` se declaran un conjunto de claves con nombres iguales a los campos institucionales que hemos utilizado para caracterizar a los departamentos, como se muestra en el siguiente listado.

```
name      .code:n = { \pyst_guard_reserved:nn { name }
                  { \tl_gset:Nn \g_pyst_meta_department_tl { #1 } } }
,
university .code:n = { \pyst_guard_reserved:nn { university }
                  { \tl_gset:Nn \g_pyst_meta_university_tl { #1 } } }
,
school     .code:n = { \pyst_guard_reserved:nn { school }
                  { \tl_gset:Nn \g_pyst_meta_school_tl { #1 } } } ,
address    .code:n = { \pyst_guard_reserved:nn { address }
                  { \tl_gset:Nn \g_pyst_meta_address_tl { #1 } } } ,
phone      .code:n = { \pyst_guard_reserved:nn { phone }
                  { \tl_gset:Nn \g_pyst_meta_phone_tl { #1 } } } ,
email      .code:n = { \pyst_guard_reserved:nn { email }
                  { \tl_gset:Nn \g_pyst_meta_email_tl { #1 } } } ,
def         .code:n = { \pyst_guard_reserved:nn { def }
                  { \tl_gset:Nn \g_pyst_dept_def_tl { #1 } } } ,
logo       .code:n =
{
  \tl_set:Nn \l_tmpa_tl { #1 }
  \tl_trim_spaces:N \l_tmpa_tl
  \tl_gset:NV \g_pyst_dept_logo_tl \l_tmpa_tl
  \bool_gset_true:N \g_pyst_user_has_logo_bool
} ,
unknown    .code:n = { } ,
```

Ocho claves, indistinguibles en su forma de las claves de asignación directa que ya se han visto, salvo que siete de los ocho están envueltas en `\pyst_guard_reserved:nn`, la misma salvaguarda que protege a `babel` y `base-classy` que impide que el usuario reescriba, sin querer, la dirección de un departamento desde `\documentclass`. Se explica la razón de esta salvaguarda en detalle en la [Apartado 4](#).

La clave `logo` sin embargo no está protegida por la salvaguarda y, por tanto, se le puede asignar un valor, el nombre de un fichero que contiene un logo que se quiere utilizar para caracterizar al departamento. No se realiza en este momento una comprobación de que el fichero existe y el código de su propiedad permite, por ejemplo, que el usuario escriba `logo={}`, que permitiría evitar que el documento tuviese un logo del departamento (a pesar que se hubiera declarado un departamento) manteniéndose el resto de los valores institucionales.

Adicionalmente, la propiedad de la clave `logo` en el namespace asigna un valor verdadero a la bandera `\g_pyst_user_has_logo_bool`, la bandera que distingue si el logo lo puso el

usuario en `\documentclass` o lo puso el departamento en el momento de creación en el fichero `pyst-depts.cfg`. Veremos que la fase POST necesita saber esto para decidir a cuál de los dos logotipos recurrir.

3.4 Las claves que no se reconocen

Existe en la declaración de los namespaces `pyst` y `pyst/dept` una clave que, en realidad, no es una clave, la declarada mediante la siguiente línea.

```
unknown .code:n = { } ,
```

Podemos definir el código asociado a la clave `unknown` (en realidad, una pseudo-clave) como “no haga nada” y es la aplicación más directa de la regla fundamental de esta nota. Una clave que `pyst` no reconoce es, por definición, una opción de KOMA-Script, `DIV`, `paper`, `BCOR`, cualquiera de los cientos que `typearea` y las clases base (`scrbook`, `scrreport` y `scrartcl`) entienden. Decidir si es válida o no es trabajo de KOMA, no de PYST-PUB. Este *handler* no hace nada porque no le corresponde hacer nada; de recogerla se ocupa, más tarde y por otra vía, el paso 5 (Apartado 5.7).

3.5 Tabla resumen de las claves de los namespace

Tabla 3: Las 19 claves del namespace `pyst`.

#	Clave	Patrón	Destino / comportamiento
1	<code>doc-type</code>	pre-escaneo	no-op aquí — la consume el bloque 3
2	<code>dept-name</code>	pre-escaneo	no-op aquí — la consume el bloque 3
3	<code>base-class</code>	reservada	<code>\g_pyst_base_class_tl</code> , validada con <code>\str_case:nnF</code>
4	<code>doc-style</code>	asignación	<code>\g_pyst_doc_style_tl</code>
5	<code>coverstyle</code>	delegada	<code>\pyst_coverstyle_normalize:n</code>
6	<code>language</code>	validada	en línea (<code>\str_case:nnF</code>), igual que <code>base-class</code>
7	<code>font</code>	asignación	<code>\g_pyst_font_tl</code> (<code>\.initial:n = stixtwo</code>)
8	<code>babel</code>	reservada	<code>\g_pyst_babel_clist</code>
9	<code>math</code>	booleano	<code>\g_pyst_module_math_bool</code>
10	<code>graphics</code>	booleano	<code>\g_pyst_module_graphics_bool</code>
11	<code>advanced-math</code>	booleano	<code>\g_pyst_module_advanced_math_bool</code>
12	<code>debug</code>	booleano	<code>\g_pyst_debug_bool</code>
13	<code>compat</code>	booleano	<code>\g_pyst_compat_bool</code>
14	<code>copyright</code>	booleano	<code>\g_pyst_copyright_bool</code>
15	<code>modules</code>	canal aditivo	<code>seq</code> — <code>\pyst_modules_route:n</code>
16	<code>doc-options</code>	canal aditivo	<code>clist</code> — <code>\pyst_koma_route:n</code>

(continúa)

Tabla 3: Las 19 claves del namespace `pyst` (continuación).

#	Clave	Patrón	Destino / comportamiento
17	<code>koma-layout</code>	canal aditivo	<code>clist</code> — ídem, y levanta <code>\g_pyst_koma_layout_bool</code> ; en fase de usuario, además <code>\g_pyst_koma_layout_user_bool</code>
18	<code>geometry-layout</code>	canal aditivo	<code>clist</code> — <code>\pyst_geometry_route:n</code>
19	<code>unknown</code>	gestor	Filtra hacia KOMA. No es clave de usuario.

Tabla 4: Las 9 claves del namespace `pyst/dept`.

#	Clave	Patrón	Destino / comportamiento
1	<code>name</code>	reservada	<code>\g_pyst_meta_department_tl</code>
2	<code>university</code>	reservada	<code>\g_pyst_meta_university_tl</code>
3	<code>school</code>	reservada	<code>\g_pyst_meta_school_tl</code>
4	<code>address</code>	reservada	<code>\g_pyst_meta_address_tl</code>
5	<code>phone</code>	reservada	<code>\g_pyst_meta_phone_tl</code>
6	<code>email</code>	reservada	<code>\g_pyst_meta_email_tl</code>
7	<code>def</code>	reservada	<code>\g_pyst_dept_def_tl</code>
8	<code>logo</code>	asignación	<code>\g_pyst_dept_logo_tl</code> + <code>\g_pyst_user_has_logo_bool</code> — no reservada, a propósito
9	<code>unknown</code>	gestor	Filtra hacia KOMA. No es clave de usuario.

Nueve claves están *reservadas* y solo se le puede asignar valor mediante su incorporación a los valores por defecto y declaración de tipos de documentos en el catálogo de tipos (`pyst-types.cfg`) o en el catálogo de los departamentos, `pyst-depts.cfg`. Se rechazarán con un aviso si el usuario las declara en `\documentclass[...]` y están incorporadas en la `clist` `\c_pyst_reserved_keys_clist`. Son `base-class`, `babel`, y los siete campos institucionales salvo `logo`, excluida a propósito porque un departamento puede tener más de un logotipo según el tipo de documento.

4 Sobre las claves reservadas

La separación en dos namespaces ([Apartado 3](#)) resuelve el problema que se había detectado que el catálogo de tipos de documentos o el de departamentos pudieran declarar variables institucionales o conformadoras de un documento de manera cruzada, pero no resuelve otro problema, la posibilidad que el usuario declarase algunos de esos valores reservados en la declaración `\documentclass[...]`. Debemos evitar que las nueve claves reservadas sean alcanzables⁶ de manera que, por ejemplo, si un usuario escribe `\documentclass[phone=9999,`

⁶En el sentido de que se les asigne un valor.

...] esa declaración no provoque ningún efecto en el documento o si otro usuario establece `\documentclass[doc-type=libro, base-class=scrartcl,...]`, el documento debe seguir componiéndose como un *libro*, lo declarado en el catálogo de tipos de documentos. y se ignore por completo el deseo del usuario.

Es decir, si decíamos al principio que las decisiones del usuario están por encima de todo lo demás, es el momento de matizar esa regla de diseño. Sigue siendo cierto pero siempre y cuando su decisión no provoque resultados erróneos o indeseados.

En este Apartado presentamos la razón que nos ha llevado a declarar algunas variables como reservadas y no a otras y como se han previsto la codificación precisa para lograr estos resultados.

4.1 Por qué unas variables se declaran reservadas y otras no

La declaración de `base-class` como clave reservada tiene una razón estructural clara puesto que decide si el tipo de documento elegido *tiene o no capítulos*. Las posibles segmentación del documento mediante los comandos `\frontmatter`, `\mainmatter` y `\backmatter`, la asignación de estilos de página mediante la clave `doc-style` que, aunque su desarrollo pertenezca a la fase POST, es fácil ver que están pensados únicamente para documentos con capítulos. Cambiar la clase desde fuera del catálogo de tipos de documentos produciría un documento incoherente que, en muchos casos, no sería ni siquiera compilable.

Los siete campos institucionales, `name`, `university`, `school`, `address`, `phone`, `email`, `def`, son datos que se declaran en el catálogo de departamentos, `pyst-depts.cfg`. Para una modificación puntual ya se ha previsto una vía en el preámbulo del propio documento, donde es posible escribir, por ejemplo, `\department{Mi otro departamento}`, o cualquier otra modificación de estos metadatos institucionales. Por ello, en el diseño del sistema PYST-PUB se ha pensado que permitir además la vía de `\documentclass[...]{pyst-pub}` sería una segunda opción sin ninguna ventaja sobre la primera. *No existe ninguna otra razón para declarar reservadas esas claves.*

De hecho, se ha permitido una excepción entre esos metadatos institucionales de manera deliberada con la clave `logo` y no por descuido sino por diseño. Un departamento puede tener legítimamente más de un logotipo y usar uno distinto según el tipo de documento que se esté generando, un membrete para un certificado, otro para una tesis así que elegirlo desde `\documentclass[...]{pyst-pub}` es exactamente el uso previsto, no un caso a impedir. De manera simétrica, su utilización en el preámbulo del documento *no está permitido*.

4.2 Codificación de la reserva de claves

El primer paso para realizar una reserva de un conjunto de claves consiste, evidentemente, en crear una lista con las claves que se desean reservar. Esto se consigue mediante la construcción de una variable del tipo `clist` realizada en el siguiente bloque.

```
\clist_const:Nn \c_pyst_reserved_keys_clist
{
  base-class , babel ,
  name , university , school , address , phone , email , def
}
```

Esta es una lista constante que se construye una sola vez y no vuelve a cambiar durante la compilación; es una única línea declarativa, separada del mecanismo que la hace cumplir. En esta creación nos encontramos con un comando que no se había visto anteriormente. `\clist_const:Nn` crea una lista constante, no modificable, en lugar de una variable, `\c_pyst_reserved_keys_clist`. De acuerdo con la convención de `expl3`, su prefijo es `c_`, el tercero que hemos utilizado. Los tres han sido:

- `g_` Variable global: sobrevive al grupo en que se fijó, y por tanto puede leerse desde cualquier punto posterior de la compilación.
- `l_` Variable local: su valor no se garantiza más allá del bloque en el que se usa.
- `c_` Constante: se fija una vez y no debe modificarse. No lleva marca de global o local porque la cuestión no se plantea.

Por lo tanto, la creación de la lista constante es una decisión de diseño, implícita en la programación y no es un dato que vaya a ser ajustado ni por el usuario, ni en tiempo de compilación ni en tiempo de ejecución.

Una vez que hemos creado una lista con las claves que queremos reservar, declarado determinadas claves como reservadas

Que la política de reserva sea una *constante* y no una variable global no es indiferente: declara, en el propio nombre, que esa lista es una decisión de diseño fija y no un dato que el sistema vaya ajustando sobre la marcha.

El guardián

```
\cs_new_protected:Npn \pyst_guard_reserved:nn #1 #2
{
  \bool_if:NTF \g_pyst_phase_user_bool
  {
    \clist_if_in:NnTF \c_pyst_reserved_keys_clist { #1 }
    { \msg_warning:nnn { pyst } { reserved-key } { #1 } }
    { #2 }
  }
  { #2 }
}
```

Dos argumentos: `#1` es el nombre de la clave, solo para poder citarlo en el aviso y `#2` es el código que de verdad hace el trabajo si se permite ejecutarlo. Que un argumento sea *código a ejecutar* y no un dato es un idioma frecuente en `expl3`, y es lo que convierte a esta función en un envoltorio: no sustituye a la acción de la clave, la envuelve.

`\clist_if_in:NnTF` es la función de consulta que faltaba por ver del módulo `clist`: pregunta si un elemento **pertenece** a una lista. `N` es la lista, `n` el elemento buscado, y el sufijo `TF` aporta las dos ramas.

El guardián reutiliza `\g_pyst_phase_user_bool`, el mismo booleano que gobierna el enrutado de los canales aditivos, para saber en qué momento está: si todavía no es la fase del usuario, deja pasar `#2` sin más comprobación, porque en los bloques 4 y 5 quien escribe es, precisamente,

el departamento o el tipo, y éstos tienen plena libertad sobre sus propios datos. Solo en el bloque 6, con la bandera levantada, se consulta la lista y se decide.

Conectado a cada clave, el patrón es siempre el mismo, se muestran dos ejemplos, uno de asignación simple y otro dentro de la validación en línea de `base-class`:

```
phone .code:n = { \pyst_guard_reserved:nn { phone }
                  { \tl_gset:Nn \g_pyst_meta_phone_tl { #1 } } } ,

base-class .code:n =
{ \pyst_guard_reserved:nn { base-class }
{
  \str_case:nnF { #1 } { ... }
} } ,
```

El rechazo es un *aviso*, no un error: el valor que ya tenía la variable, el del tipo, o el del departamento, permanece intacto y la compilación continúa con un documento válido. El mensaje, además, no se limita a decir que no:

«Opcion 'phone' reservada: se declara en los ficheros .cfg (o, para los datos institucionales, con los comandos de metadatos del preambulo). Ignorada aqui; se mantiene el valor vigente.»

indica dónde va cada cosa, porque un aviso que solo prohíbe obliga a ir a buscar la alternativa por otro lado.

Ampliar la lista

Añadir una clave reservada más exige *dos* ediciones, no una: incluir su nombre en la lista constante `\c_pyst_reserved_keys_clist`, definida en el fichero `pyst-configuration.ltx`, y envolver el cuerpo de su `.code:n` en `\pyst_guard_reserved:nn`, como hacen ya las nueve claves existentes. La lista, por sí sola, no hace nada: es el guardián, invocado desde el propio `.code:n` de la clave, quien la consulta.

Un nombre añadido a la lista sin la segunda adición produce una reserva ficticia: la clave no quedaría protegida. Ese fallo no se detectó en las primeras versiones del sistema. Una vez encontrado, se consideró la necesidad de incorporar un modo de comprobación que permitiese detectarlo y advertir del mismo.

El bloque 0: comprobar que la reserva se realiza

`\pyst_check_reserved_keys`: recorre la lista constante y, por cada nombre, hace algo aparentemente extraño: aplicar la clave en seco, con valor vacío, en el namespace donde exista, habiendo levantado antes la bandera de fase de usuario. Si el guardián está en su sitio, intercepta la escritura y lo hace constar; si no está, la clave ejecuta su cuerpo y la ausencia queda al descubierto. Es una prueba en miniatura, ejecutada en cada compilación.

De ahí sale su numeración. La comprobación tiene que ejecutarse **antes** de que ninguna fuente haya escrito nada real, porque siete de las nueve claves reservadas son campos de departamento: si se ejecutara después, una clave sin guardián no solo delataría su ausencia,

sino que además escribiría un valor vacío encima del metadato que el departamento acabara de fijar, y un diagnóstico limpio se convertiría en un diagnóstico más una pérdida de datos.

Llamarlo bloque o y ponerlo el primero convierte esa condición en algo estructural. Mientras la comprobación estuvo intercalada entre otros bloques, era correcta porque los que la precedían no escribían valores reales; es decir, dependía de una propiedad de sus vecinos que alguien podía romper sin darse cuenta al añadir código. Ejecutándose antes que todo lo demás, no hay vecinos de los que depender.

Esa restricción se comprobó experimentalmente: fabricando la avería, quitando el guardián de la clave `name` pero dejándola en la lista y compilando con la comprobación colocada antes y después del bloque del departamento. En ambos casos se emite el diagnóstico; solo en el segundo el nombre del departamento aparece vacío.

Desde la versión 2.1 este diagnóstico es un `\msg_error` y no un `\msg_warning`. La diferencia importa: un aviso puede filtrarlo un paquete como `silence` y pasa fácilmente inadvertido en un `.log` largo, mientras que un error detiene la compilación en modo interactivo y es imposible no verlo. No se eligió `\msg_fatal` porque hay un valor de reserva seguro (el documento compila) y porque un descuido de mantenimiento no debe impedir trabajar a quien solo quiere compilar su tesis.

5 Cómo se leen los valores del usuario en el sistema PYST-PUB

Hemos visto cómo se declaran los valores/datos que puede utilizar el usuario, en los catálogos de configuración y en el conjunto de opciones que se pueden definir en el momento de declaración del documento, mediante `\documentclass[...]`. Es decir, el sistema PYST-PUB tiene que gestionar la entrada, validación, resolución de prioridades y distribución de opciones procedentes de cuatro fuentes distintas:

1. Valores globales por defecto (*defaults*).
2. El catálogo de tipos de documento (`pyst-types.cfg`).
3. El catálogo institucional de departamentos (`pyst-depts.cfg`).
4. Las opciones explícitas introducidas por el usuario en `\documentclass[...]`.

5.1 El Invariante: Jerarquía y Reglas de Precedencia

El diseño del sistema PYST-PUB se fundamenta en un modelo estricto de sobrescritura mostrado en la regla siguiente.

$$\left. \begin{array}{l} 1.- \text{Defaults} < 2.- \text{Tipo} \\ 3.- \text{Departamento} \end{array} \right\} < 4.- \text{Usuario}$$

Expresado en palabras, todo gravita en el valor del *tipo de documento* que se desea generar. Este tipo de documento se elige de un *catálogo de tipos de documentos*, definidos en el fichero `pyst-types.cfg`. Todos los tipos comparten un conjunto de *parámetros por defecto*.

En el momento de definir un tipo de documento los valores por defecto se pueden modificar y añadir nuevas claves y valores. Por ejemplo, cada tipo de documento posee una clase, definida por la clave `base-class` que no tiene sentido que se declare entre los valores por defecto. Como

también que se asigne qué tipo de layout posee, si tiene “portada”⁷ o no y qué tipo en el caso que la tenga. En general, al crear un tipo de documento en el catálogo de tipos se pretende asignar características comunes que en la mayoría de los casos no necesitarán cambiarse.

De manera *paralela*, hemos supuesto que algunos tipos de documentos pueden generarse vinculados a un departamento⁸. Por ello, se ha creado un *catálogos de departamentos*, definidos en el fichero `pyst-depts.cfg` que declaran un conjunto de valores institucionales o *metadatos departamentales*.

Por encima de estas fuentes de valores, el usuario puede añadir o modificar determinadas claves en el momento de la escritura de `\documentclass[...]`. No tiene demasiado sentido que en la creación del tipo de documento se elija la tipografía, por ejemplo. No obstante, la existencia de un catálogo de tipos de documentos pretende simplificar la creación de documentos, evitando la asignación de valores a la mayor parte de las claves, para lograr una imagen corporativa y propia del centro.

Junto a todo lo anterior hemos decidido que los documentos generados en el sistema PYST-PUB cumplan tres criterios de diseño importantes.

1. No es posible realizar asignaciones cruzadas en los catálogos; es decir, en el catálogo de tipos de documentos no pueden aparecer claves y valores de los tipos de documentos y en el catálogo de los tipos de documentos no aparecen metadatos institucionales de los departamentos.
2. Los metadatos de los departamentos, con la excepción de la clave `logo`, no pueden modificarse en `\documentclass[...]`. El cambio, si se desea, debe realizarse en el preámbulo del documento. Por ejemplo, podemos cambiar el teléfono de un departamento escribiendo en el preámbulo

```
\phone{+34-954999888}
```

3. Y por encima de todo esto, una regla fundamental: *no inventar lo que KOMA-script ya hace*. Si un usuario decide que su documento va a tener un cuerpo de letra de 12 pt, y páginas apaisadas, KOMA establece que eso se logra escribiendo

```
fontsize=12pt, landscape.
```

Carece de sentido que el sistema PYST-PUB trate de no interpretar nada que no esté definido por él. Los anteriores claves y valores tienen que ser interpretados por KOMA.

Es decir, la confección de un documento en el sistema PYST-PUB está sujeto a un conjunto de reglas, invariantes y jerarquías que deben ser programados de manera clara y eficiente. *Tenemos que leer los valores de los dos catálogos y los valores de `\documentclass[...]`. Utilizar lo que es de PYST-PUB y todo lo demás dejarlo en manos de KOMA.*

En el resto de este Apartado describiremos de qué manera se ha articulado todo lo anterior mediante una programación adecuada. Los códigos presentados se encuentran en el tercer fichero de la etapa PRE, `pyst-options.ltx`.

⁷Recordemos que la clave `coverstyle` declara un fichero que en realidad supera el concepto de portada. Es más que una portada (y a veces no es una portada). Ver el Manual de Usuario para ampliar el concepto.

⁸Mientras que el *tipo de documento* es una declaración obligatoria, el departamento, a través de la clave `dept-name` puede declararse o no.

5.2 Lectura de los catálogos de configuración

Si en el [Apartado 2](#) se realizaba la declaración de los ficheros de configuración, el primer paso que debe darse es la lectura de estos ficheros y la creación de las variables de tipo `sec` que nos informan de los tipos de documentos declarados y los departamentos de los que disponemos datos institucionales. Es decir, el primer paso de esta parte de la fase PRE es la ejecución del bloque de código siguiente:

```
\file_if_exist:nTF { pyst-types.cfg }
{ \file_input:n { pyst-types.cfg } }
{ \msg_fatal:nn { pyst } { types-cfg-missing } }

\file_if_exist:nTF { pyst-depts.cfg }
{ \pyst_dept_input_twoe:n { pyst-depts.cfg } }
{ \msg_warning:nn { pyst } { depts-cfg-missing } }

\seq_gclear:N \g_pyst_available_types_seq
\prop_map_inline:Nn \g_pyst_types_prop
{
  \seq_gput_right:Nn \g_pyst_available_types_seq {#1}
}

\seq_gclear:N \g_pyst_available_depts_seq
\prop_map_inline:Nn \g_pyst_depts_config_prop
{
  \seq_gput_right:Nn \g_pyst_available_depts_seq {#1}
}
```

Observemos en primer lugar la asimetría de los mensajes que se generan en el caso de que el catálogo de tipos de documentos no exista o que no exista el catálogo de departamentos. La razón es clara. Si no existe un catálogo de tipos de documentos, el sistema PYST-PUB *no puede funcionar*. En ese caso, se genera un error fatal y se aborta la compilación.

La situación es diferente en el caso de falta de catálogos de departamentos. El sistema genera un aviso pero la compilación no se detiene. Incluso, los metadatos se pueden crear en el preámbulo y utilizarse en el resto documento sin ningún problema.

El resultado del anterior bloque de código, analizado en el [Apartado 2](#), es la creación de un conjunto de variables globales, `\g_pyst_type_defaults_tl`, `\g_pyst_types_prop`, `\g_pyst_available_types_seq`, `\g_pyst_depts_config_prop` y `\g_pyst_available_depts_seq` que recogen todo lo declarado en estos ficheros (catálogos) de configuración.

5.3 Prevalidación del tipo de documento y el departamento

Para garantizar las reglas de diseño definidas es imprescindible conocer en primer lugar qué tipo de documento se desea generar⁹ y, si se ha declarado un departamento, si este departamento existe; esto es, si el tipo de documento y el posible departamento están dentro de los tipos de documentos y departamentos declarados en los catálogos correspondientes.

⁹Es importante repetir que el *tipo de documento*, identificado por la clave `doc-type` es la única clave obligatoria que debemos suministrar al sistema mediante la declaración `\documentclass[...]{pyst-pub}`.

El siguiente código, presente en el Bloque 3 de `pyst.options.ltx` nos permite comenzar a efectuar esta tarea. Para facilitar su descripción, en esta ocasión lo hemos numerado. Veremos que el resultado del mismo es, si no se aborta la compilación, la asignación a una o dos variables globales de los valores declarados por el usuario del tipo de documento y nombre del departamento, en caso de realizarse una declaración de este último.

```

1  \tl_new:N \l_pyst_scan_type_tl
2  \tl_new:N \l_pyst_scan_dept_tl
3  \tl_new:N \l_pyst_scratch_tl
4
5  \keys_define:nn { pyst / prescan }
6  {
7    doc-type .code:n = { \tl_set:Nn \l_pyst_scan_type_tl { #1 } } ,
8    dept-name .code:n = { \tl_set:Nn \l_pyst_scan_dept_tl { #1 } } ,
9    unknown .code:n = { } ,
10 }
11
12 % \@raw@classoptionslist (kernel, LaTeX >= 2020/10/01) is used in
13 % preference to \@classoptionslist wherever either is read in this
14 % file: it preserves the braces around a grouped value such as
15 % doc-options={...}, where the older list does not.
16 \cs_if_exist:NTF \@raw@classoptionslist
17 { \keys_set:nV { pyst / prescan } \@raw@classoptionslist }
18 { \keys_set:nV { pyst / prescan } \@classoptionslist }
19
20 \tl_if_empty:NT \l_pyst_scan_type_tl
21 { \msg_fatal:nn { pyst } { doc-type-required } }
22 \prop_if_in:NVF \g_pyst_types_prop \l_pyst_scan_type_tl
23 { \exp_args:NnnV \msg_fatal:nnn { pyst } { unknown-type }
24   \l_pyst_scan_type_tl }
25 \tl_gset_eq:NN \g_pyst_current_type_tl \l_pyst_scan_type_tl
26
27 \tl_if_empty:NF \l_pyst_scan_dept_tl
28 {
29   \prop_if_in:NVTF \g_pyst_depts_config_prop \l_pyst_scan_dept_tl
30   { \tl_gset_eq:NN \g_pyst_current_dept_tl \l_pyst_scan_dept_tl }
31   { \exp_args:NnnV \msg_warning:nnn { pyst } { unknown-dept }
32     \l_pyst_scan_dept_tl }
33 }

```

En las líneas 1-3 se crean tres variables locales auxiliares. Entre las líneas 5-10 se define un espacio de nombres auxiliar, `pyst/prescan` y en las líneas 16-18 se realiza la lectura de todas las opciones que existen en `\documentclass[...]{pyst-pub}`, utilizando una variable interna, `\@raw@classoptionslist`, que el núcleo de $\text{\LaTeX}$ desde la versión de octubre de 2020 crea sin recortar los espacios de su interior. Es decir, en `\@raw@classoptionslist` se vuelca la lista completa de opciones, tal cual la escribe el usuario. En el caso de que esa variable no exista, se utiliza `\@classoptionslist`. Vamos a analizar este código con más detenimiento.

Conviene reparar en un detalle de nomenclatura que hasta ahora no había hecho falta. Al definir las dos claves del espacio `pyst/prescan`, se usa `\tl_set:Nn`, no `\tl_gset:Nn`: sin la

g de *global*. Y las variables se llaman `\l_pyst_scan_type_tl`, con *l_* inicial en lugar de *g_*. Ambas cosas van juntas y expresan lo mismo: son variables *locales*, de vida breve, que solo sirven para transportar un valor de aquí al paso siguiente. `expl3` impone esta correspondencia como convención estricta, una variable *l_* se asigna con `set`, una *g_* con `gset` y respetarla es lo que permite saber, con solo leer un nombre, cuánto vive un dato y quién puede haberlo tocado.

Las dos funciones de las líneas 16 y 17 son también nuevas. `\cs_if_exist:N\TF` pregunta si una secuencia de control *está definida*. La firma *N* muestra que espera el nombre de una variable y el ya conocido sufijo *TF* para las dos ramas verdadera y falsa de la comprobación. En el caso de existencia, lo que ocurre como ya hemos dicho en versiones del núcleo de $\text{\LaTeX}$ posteriores a Octubre de 2020, se ejecuta la línea 17. Como hemos dicho, en el caso de que no exista, se usa aquí como salvaguarda de compatibilidad, `\@classoptionslist` y se ejecuta la línea 18.

En cualquiera de los casos, nos encontramos con una función `\keys_set:nV` que es la contrapartida de `\keys_define:nn`. Si esta *declara* las claves de un espacio de nombres, `\keys_set:nV` realiza una función mucho más compleja. Por su importancia, la función `\keys_set:nV` se explica con todo detenimiento en el [Apartado 9](#). Por ahora, nos basta con conocer que, de acuerdo con su firma, (*n*), espera recibir un espacio de nombres a procesar y su segundo argumento es el nombre de una variable que contiene el *keyval* que queremos procesar. *No se procesa el nombre de la variable sino su contenido*, su valor, como se indica por el sufijo de su firma *V*.

En este caso, se compara las claves del espacio de nombre con las claves presentes en el *keyval*. Aquellas que coinciden se procesan (ahora veremos cómo) y las que no coinciden, se ignoran, de acuerdo con la definición del espacio de nombres. Para las que coinciden, el *valor* se convierte en el *#1* de la propiedad de la clave correspondiente; propiedad que se ejecuta.

Aunque se procesa la lista que el usuario introdujo como opciones en `\documentclass[...]`, todo lo que no sea *doc-type* o *dept-name*, las únicas claves declaradas en el espacio de nombres *pyst/prescan*, cae en un *unknown* igualmente vacío, puesto que solo hay que capturar esos dos posibles valores. Como resultado de este proceso, las variables `\l_pyst_scan_type_tl` y `\l_pyst_scan_dept_tl` asumen los valores que las claves *doc-type* y *dept-name*, respectivamente, tuvieran en `\documentclass[...]{pyst-pub}`.

Analicemos las líneas 20, 21, 22, 23, en las que aparecen tres nuevas funciones y sigamos viendo qué está ocurriendo.

- `\tl_if_empty:NT` pregunta si una variable de texto está vacía. Ya conocemos las dos mitades de su nombre: opera sobre el módulo *tl*, recibe el nombre de la variable (*N*) y usa el sufijo *T*, con una sola rama, porque el único caso que interesa aquí es que *sí* esté vacía; si contiene algo, se continúa. Si está vacía, emite un mensaje `\msg_fatal:nn { pyst } { doc-type-required }`; es decir, la compilación se detiene porque no se ha encontrado una asignación *doc-type=xxxx* y el tipo de documento es la única clave obligatoria.
- `\prop_if_in:N\VF` interroga a una lista de propiedades (una variable del tipo *property list*. Pregunta si una clave dada existe en la lista. Su firma reúne tres cosas ya vistas, *N* es la lista de propiedades, *V* indica que la clave a buscar viene dada como el *valor* de una variable, no como texto literal y *F* señala que solo interesa el caso negativo: si el tipo *no* está en el catálogo, abortar la compilación mediante un nuevo mensaje `\msg_fatal:nnn`. Para construir este mensaje, tenemos la tercera función.

- `\exp_args:NnnV`, `\msg_fatal:nnn` espera tres argumentos normales, módulo, nombre del mensaje y un dato pero el dato que queremos darle no es un texto literal, sino el *contenido* de `\l_pyst_scan_type_tl`. Y no existe una variante `\msg_fatal:nnV` a la que recurrir.

La firma de `\exp_args:NnnV` se lee, letra a letra, como instrucciones sobre los cuatro argumentos que la siguen:

- N** Pasa el primero sin tocarlo: la propia función `\msg_fatal:nnn`.
- n** Pasa el segundo tal cual: `{ pyst }`.
- n** Y el tercero también literal: `{ unknown-type }`.
- V** Del cuarto, en cambio, toma su *valor*: sustituye `\l_pyst_scan_type_tl` por el nombre del tipo que el usuario escribió. Un nombre de tipo de documento inventado, si hemos llegado a este punto.

Hecho ese trabajo previo, `\exp_args` desaparece y lo que queda ejecutándose es, sencillamente, `\msg_fatal:nnn { pyst } { unknown-type } { inventado }`. Dicho de otro modo: `\exp_args:N...` permite construir, sobre la marcha, la variante de firma que no existe en la biblioteca.

En ambos casos se usa `\msg_fatal`, que detiene la compilación sin excepción, frente a `\msg_error`, que avisa y continúa con lo que haya. Un tipo de documento inválido o ausente no es un defecto recuperable: sin él, nada de lo que sigue tiene sentido.

Si la compilación no se ha abortado porque existe la clave `doc-type` y el valor de la misma es uno de los que se encuentra en el catálogo de tipos de documentos, la línea 24 asigna el valor de esa clave a la variable global `\g_pyst_current_type_tl`. Si el usuario hubiese escrito `\documentclass[doc-type=libro, ...]{pyst-pub}`, el resultado de todo este bloque sería que el contenido de la variable global `\g_pyst_current_type_tl` es `libro`.

En el último bloque del anterior código, el definido por las líneas 26-32, se realiza un procesamiento similar para el departamento. En este caso, sin embargo, el comportamiento es muy diferente. En primer lugar, podemos ver que si no se ha declarado ningún departamento, no ocurre nada. Sin embargo, si se ha declarado un departamento y este (su acrónimo) no se encuentra dentro de la lista de propiedades `\g_pyst_depts_config_prop`, la que `\DeclareDepartment` fue construyendo al leerse el catálogo, se genera un mensaje que en este caso solo es un aviso. Se menciona que el departamento declarado por el usuario se desconoce y la compilación continúa. Esta asimetría del departamento respecto al tipo obedece, como es claro, a la diferencia de importancia en los errores en uno y otro caso. Sin departamento se puede generar un documento pero no es posible generar ninguno si se desconoce el tipo de documento que se desea obtener.

Y si el departamento sí consta en el catálogo, la línea 29 hace con él lo mismo que la línea 24 había hecho con el tipo, copiar su acrónimo a una variable global, `\g_pyst_current_dept_tl`. Conviene no confundir las dos variables que intervienen aquí, porque sus nombres se parecen y sus papeles son opuestos: `\g_pyst_depts_config_prop` es la lista de propiedades *donde se busca*, y `\g_pyst_current_dept_tl` es la variable de texto *donde se anota el resultado* cuando la búsqueda tiene éxito.

Esa distinción no es un detalle de nomenclatura: es la que gobierna todo el Bloque 5. Al terminar el Bloque 3, `\g_pyst_current_dept_tl` contiene un acrónimo si y solo si se declaró

un departamento y además consta en el catálogo. En cualquier otro caso sigue vacía: vacía porque nadie la escribió, si el usuario no declaró ninguno; vacía porque se tomó la rama del aviso, si declaró uno inexistente. Una sola variable resume, por tanto, el resultado de las dos comprobaciones.

Si la compilación no se ha abortado, el resultado de este bloque es que la variable global `\g_pyst_current_type_tl` tiene el valor declarado por el usuario en la clave `doc-type`. Además, si el usuario ha declarado un departamento, y ese departamento está dentro del catálogo de departamentos, la variable global `\g_pyst_current_dept_tl` toma el valor de la clave `dept-name`. En caso contrario, esta variable está vacía.

5.4 Bloque 4. Aplicación de los valores por defecto y del tipo de documento

En este bloque de código se aplican los dos primeros orígenes de valores que configuran el documento propiamente dicho y lo hace en el orden que habíamos descrito, primero los valores por defecto en el caso que existan y después los valores del tipo de documento seleccionado, que se superponen sobre los primeros por escribirse en segundo lugar.

```
\tl_if_empty:NF \g_pyst_type_defaults_tl
{
  \exp_args:NV \pyst_validate_keyval:nnn \g_pyst_type_defaults_tl
  { pyst } { DeclareDefaultTypeOptions }
  \keys_set:nV { pyst } \g_pyst_type_defaults_tl
}

\prop_get:NVN \g_pyst_types_prop \g_pyst_current_type_tl
\l_pyst_scratch_tl
\exp_args:NV \pyst_validate_keyval:nnn \l_pyst_scratch_tl
{ pyst } { pyst-types.cfg }
\keys_set:nV { pyst } \l_pyst_scratch_tl
```

En el código se comprueba en primer lugar si existe el *keyval* definido por los valores por defecto que se aplican a todos los tipos de documentos. Este *keyval* se ha obtenido anteriormente y guardado, si existe, en la variable global `\g_pyst_type_defaults_tl`. Si esta variable *no está vacía*, que es lo que comprueba el comando condicional `\tl_if_empty:NF`, se ejecuta el bloque. En este aparece por primera vez `\pyst_validate_keyval:nnn`. Por su complejidad, hemos preferido estudiarla en el [Apartado 5.9](#) y lo que más nos interesa en este momento es señalar su función. Valida un *keyval* (primer argumento) respecto a un espacio de nombres, segundo argumento. Su tercer argumento declara la procedencia del *keyval* y solo tiene una función informativa.

Como resultado de la aplicación de la función, si hubiera valores por defecto y alguna clave del mismo no perteneciera al espacio de nombres `pyst`, se emitiría un aviso en el fichero `.log` como el mostrado en las siguiente líneas

```
Package pyst Warning: La clave 'DIV' en DeclareDefaultTypeOptions no es una
(pyst)                   clave de PYST. Será ignorada.
```

generadas al haber escrito `DIV=12` dentro de las opciones por defecto. Posteriormente, con independencia de que algunas de las claves del *keyval* no se hubieran validados, se ejecutaría la siguiente línea del código

```
\keys_set:nV { pyst } \g_pyst_type_defaults_tl
```

Y aquí, por fin, ocurre algo. `\keys_set:nV` recorre el *keyval* y, por cada *clave=valor*, busca en el espacio de nombres `{ pyst }` una clave que se llame igual y ejecuta la instrucción que tiene asociada, con el valor como su parámetro #1.

La *V* de la firma repite el mismo asunto que el `\exp_args` anterior: lo que se procesa es el *valor* de la variable, no su nombre. El primer argumento, en cambio, lleva *n* y se pasa literal.

Al terminar esta línea, las variables globales que los `.code:n` de esas claves manipulan han quedado escritas. Si los valores por defecto declaraban `language=spanish`, `\g_pyst_language_tl` contiene ya `spanish`, y seguirá conteniéndolo salvoprop_gput:Nnn que una fuente posterior escriba encima.

El tipo de documento ha sido validado en el anterior bloque, por lo que solo debemos aplicar su valor para recuperar la lista de propiedades (el *keyval*) del *property list* correspondiente, como vemos en lo siguiente:

```
\prop_get:NVN \g_pyst_types_prop \g_pyst_current_type_tl
\l_pyst_scratch_tl
```

`\prop_get:NVN` es el comando inverso del comando `\prop_gput:Nnn` que se utilizó para construir la variable *prop* `\g_pyst_types_prop`, ver el [Apartado 2.2](#). Si `\prop_gput:Nnn` deposita un nombre del tipo y su *keyval* asociado en `\g_pyst_types_prop`, el comando `\prop_get:NVN` extrae el *keyval* asociado a su segundo parámetro de la variable señalada por el primer parámetro y lo deposita en la variable nombrada por su tercer parámetro. Su firma *NVN* nombra esos tres argumentos, y tiene la particularidad de que el último no es un dato de entrada sino de salida

N La lista de propiedades de la que leer, `\g_pyst_types_prop`.

V La clave a buscar, dada como el *valor* de una variable, el nombre del tipo capturado en el pre-escaneo.

N La variable de destino donde depositar lo encontrado, `\l_pyst_scratch_tl`.

Esa tercera *N* es una convención frecuente en [expl3](#): en lugar de «devolver» un resultado como haría una función de otro lenguaje, se le indica dónde escribirlo. Y adviértase la correspondencia de nombres: las dos primeras son *g_*, globales, porque deben sobrevivir de un bloque a otro; el destino es *l_*, local, porque su contenido se consume en las dos líneas siguientes.

Recuperar el *keyval* en lugar de aplicarlo directamente es la contrapartida natural de haberlo guardado en bruto en la declaración ([Apartado 3](#)). El trabajo de interpretarlo se pospone hasta este momento, y solo ocurre para el tipo que de verdad se ha pedido, nunca para los restantes.

5.5 Bloque 5. Aplicación del departamento

Resuelto ya el nombre del departamento en el Bloque 3, y aplicadas las claves que configuran el documento, este último bloque de la cadena de precedencia es el que hace efectivo

el departamento, recuperando sus claves del catálogo y aplicándolas. Analicemos el bloque correspondiente de código.

```
\tl_if_empty:NF \g_pyst_current_dept_tl
{
  \prop_get:NVN \g_pyst_depts_config_prop \g_pyst_current_dept_tl
    \l_pyst_scratch_tl
  \exp_args:NV \pyst_validate_keyval:nnn \l_pyst_scratch_tl
    { pyst / dept } { pyst-depts.cfg }
  \keys_set:nV { pyst / dept } \l_pyst_scratch_tl
  \tl_gset_eq:NN \g_pyst_dept_own_logo_tl \g_pyst_dept_logo_tl
}
\bool_gset_false:N \g_pyst_user_has_logo_bool
```

La condición `\tl_if_empty:NF` sobre `\g_pyst_current_dept_tl` interroga si la variable está vacía, que puede haber ocurrido porque el usuario no ha asignado ningún valor a la clave `dept-name` o bien, el valor que ha dado no existe en el catálogo. La primera línea del interior es la recuperación propiamente dicha si estas condiciones no se han cumplido; esto es, la variable no está vacía y el departamento (su acrónimo) se ha encontrado en el catálogo correspondiente

```
\prop_get:NVN \g_pyst_depts_config_prop \g_pyst_current_dept_tl
  \l_pyst_scratch_tl
```

Es el mismo `\prop_get:NVN` descrito en el bloque anterior, ahora sobre otra *property list*, `\g_pyst_depts_config_prop`, generada en el [Apartado 2.3](#). En este caso, los tres argumentos de `\prop_get:NVN` son

- N** La lista de propiedades de la que leer: `\g_pyst_depts_config_prop`.
- V** La clave a buscar, dada como el *valor* de una variable, el acrónimo que el Bloque 3 dejó e n `\g_pyst_current_dept_tl`.
- N** La variable **de destino**, donde depositar lo encontrado, `\l_pyst_scratch_tl`.

Podemos observar que `\l_pyst_scratch_tl` es la misma variable de trabajo que el Bloque 4 acababa de usar para el *keyval* del tipo. Su contenido anterior se pierde aquí, y no importa, ya se había consumido. Es exactamente la razón por la que se llama *scratch* y lleva el prefijo `l_` de las variables locales.

Tras esta línea, `\l_pyst_scratch_tl` contiene el texto completo de las claves del departamento tal como se escribieron en `pyst-depts.cfg` todavía sin interpretar. Sigue siendo texto en bruto: lo que se ha recuperado es la cadena, no su significado. Es la contrapartida natural de haberlas guardado sin procesar en la declaración ([Apartado 3](#)): el trabajo de interpretarlas se pospone hasta este momento, y solo ocurre para el departamento que de verdad se ha pedido.

De nuevo, antes de aplicar ese *keyval* se le somete a una validación respecto al espacio de claves definido para el departamento, `pyst/dept`.

```
\exp_args:NV \pyst_validate_keyval:nnn \l_pyst_scratch_tl
  { pyst / dept } { pyst-depts.cfg }
```

La función `\pyst_validate_keyval:nnn` se estudia en detalle en [Apartado 5.9](#) y ya es conocida del apartado anterior. Validado el *keyval*, la línea siguiente lo aplica por fin:

```
\keys_set:nV { pyst / dept } \l_pyst_scratch_tl
```

Y aquí está la diferencia esencial con el tipo, que no es de mecanismo sino de destino, la llamada se hace contra `{ pyst / dept }` y no contra `{ pyst }`. Los datos institucionales del departamento van a parar a un espacio de nombres propio, disjunto del que configura el documento. Es esa separación, y no ninguna vigilancia añadida, la que hace imposible por construcción que el catálogo del departamento sobrescriba lo que fijó el catálogo de los tipos de documentos, ya que no comparten ni una sola clave sobre la que colisionar.

Cierran el bloque dos asignaciones que no aplican ninguna clave, pero que preparan el terreno para lo que viene después.

```
\tl_gset_eq:NN \g_pyst_dept_own_logo_tl \g_pyst_dept_logo_tl
}
\bool_gset_false:N \g_pyst_user_has_logo_bool
```

La primera usa una función que no habíamos visto, `\tl_gset_eq:NN`, con firma `NN`. El sufijo `_eq` significa *copiar una variable en otra*, y por eso ambos argumentos son nombres de variable, sin llaves. Podemos comparar con `\tl_gset:Nn`, que ya conocemos. Si aquella asigna un *texto* a una variable, esta copia el contenido de una variable a otra. Escribir `\tl_gset:Nn \g_destino { \l_origen }` guardaría el *nombre* de la variable, no su contenido, un error clásico que `_eq` evita por diseño de `expl3`.

Lo que se copia es el logotipo propio del departamento a `\g_pyst_dept_own_logo_tl`, antes de que el usuario tenga ocasión de declarar `\g_pyst_dept_logo_tl` en el Bloque 6. La fase POST la usa como valor de salvaguarda si el usuario no declara un logotipo o el logotipo declarado no se encuentra.

La última línea baja la bandera¹⁰ de “logotipo puesto por el usuario”. El departamento puede haber escrito un `logo=...` propio, pero eso no debe contar como una elección del usuario. Nótese que está *fuera* del condicional, y no por descuido. `\g_pyst_user_has_logo_bool` debe quedar en falso tanto si hubo departamento como si no, puesto que esta variable valdrá `true` únicamente cuando el usuario asigne a la clave `logo` el nombre de un fichero existente en `\documentclass`.

5.6 Bloque 6. Aplicación de los valores declarados por el usuario

Apliquemos a continuación los valores declarados por el usuario en el momento de escribir la lista de opciones en `\documentclass[...]`.

```
\bool_gset_true:N \g_pyst_phase_user_bool
\cs_if_exist:NTF \@raw@classoptionslist
{
  \keys_set:nV { pyst } \@raw@classoptionslist
  \keys_set:nV { pyst / dept } \@raw@classoptionslist
}
{
  \keys_set:nV { pyst } \@classoptionslist
```

¹⁰ Asigna un valor `false` a `\g_pyst_user_has_logo_bool`.

```
\keys_set:nV { pyst / dept } \@classoptionslist
}
```

Aquí se levanta¹¹ `\g_pyst_phase_user_bool`, la bandera que gobierna tanto el encauzado de los canales aditivos (Apartado 3.2.4) como la salvaguarda de claves reservadas (Apartado 4), y se procesa, por fin, la lista cruda completa contra `pyst`. Es la única de las cuatro llamadas que lee directamente lo que el usuario escribió en `\documentclass[...]{pyst-pub}`; las tres anteriores leen de las listas de propiedades que rellenaron los ficheros `.cfg`. Y es, por construcción, la última. Cualquier clave que las cuatro compartan queda con el valor que fija esta llamada, porque ninguna otra se ejecuta después. Ese simple hecho, que las cuatro escriben en las mismas variables globales y que el usuario escribe el último, es toda la implementación del compromiso

1.- Defaults < 2.- Tipo
3.- Departamento } < 4.-Usuario

No es necesario comprobar si una variable está vacía en ningún punto de este código.

5.7 Bloque 7. Filtrar y no interceptar

Queda lo que `pyst` no reconoció, las opciones de KOMA. La regla fundamental definida en esta nota¹² dice que todo las claves que no se han interceptado hay que entregárselas a KOMA tal cual, sin reinterpretarlas, así que el Bloque 7 recorre una *copia* de la lista cruda y aparta de ella las claves propias de PYST-PUB, dejando pasar el resto sin tocarlo. En el siguiente bloque de código, que también numeramos por su complejidad, se realiza esta operación.

```
1 \clist_const:Nn \c_pyst_container_keys_clist { doc-options , koma-layout ,
   geometry-layout }
2 \clist_new:N \l_pyst_rest_clist
3 \seq_new:N \l_pyst_pair_seq
4 \tl_new:N \l_pyst_keyname_tl
5
6 \cs_new_protected:Npn \pyst_filter_koma_options:n #1
7 {
8   \clist_map_inline:nn { #1 }
9   {
10     \seq_set_split:Nnn \l_pyst_pair_seq { = } { ##1 }
11     \tl_set:Nx \l_pyst_keyname_tl { \seq_item:Nn \l_pyst_pair_seq { 1 } }
12   }
13   \tl_trim_spaces:N \l_pyst_keyname_tl
14   \clist_if_in:NnF \c_pyst_container_keys_clist { \l_pyst_keyname_tl
15   {
16     \exp_args:Nne \keys_if_exist:nnF { pyst } { \l_pyst_keyname_tl
17     { \clist_put_right:Nn \l_pyst_rest_clist { ##1 } } }
```

¹¹Asigna un valor `true` a `\g_pyst_phase_user_bool`.

¹²En realidad, en el sistema PYST-PUB.

```

17         }
18     }
19 }
20
21 \cs_if_exist:NTF \@raw@classoptionslist
22 { \exp_args:NV \pyst_filter_koma_options:n \@raw@classoptionslist }
23 { \exp_args:NV \pyst_filter_koma_options:n \@classoptionslist }
24
25 \clist_gput_right:NV \g_pyst_user_opts_clist \l_pyst_rest_clist

```

Este bloque es el más denso de la fase PRE y conviene recorrerlo línea a línea, porque casi cada línea aporta una función nueva.

En la línea 1 se crea una variable del tipo `clist` constante, `\c_pyst_container_keys_clist` que contendrá los valores separados por comas `doc-options` , `koma-layout` , `geometry-layout` . En las líneas 2 3 y 4 simplemente se crea un grupo de variables locales de distintos tipos.

Entre las líneas 6-19 se crea una nueva función `\pyst_filter_koma_options:n` mediante un comando conocido `\cs_new_protected:Npn`. En la línea 8 nos encontramos con un nuevo comando, `\clist_map_inline:nn`.

Los módulos de `expl3` que gestionan colecciones ofrecen siempre una función de *recorrido*, y esta es la de `clist`. De acuerdo con su firma, `nn`, toma dos argumentos entre llaves. El primer argumento es una lista y el segundo es un bloque de código que ejecuta una vez por cada elemento de la lista, poniendo el elemento en curso a disposición del bloque. Es el equivalente al bucle *for each* de otros lenguajes. Ya habíamos visto a su hermana `\prop_map_inline:Nn` al construir las secuencias derivadas; el patrón es idéntico.

Y aquí aparece una peculiaridad sintáctica de $\text{\TeX}$ que merece especial atención, porque es fuente segura de errores. Dentro del bloque, el elemento en curso se identifica como `##1`, con **dos** almohadillas, no una. La razón es que ese bloque está a su vez dentro del cuerpo de una definición, la de `\pyst_filter_koma_options:n` y $\text{\TeX}$ exige duplicar la almohadilla en cada nivel de anidamiento. Si el mismo `\clist_map_inline:nn` se escribiera suelto, a nivel de fichero y fuera de toda definición, se escribiría `#1` con una sola `#`. Confundir ambos contextos produce, en el mejor de los casos, un error de compilación, y en el peor una almohadilla suelta que se cuela en silencio.

`\seq_set_split:Nnn` ya la conocemos de los canales aditivos, aunque aquí es local (`set`, no `gset`): parte el elemento por el signo igual y deja los trozos como elementos de una secuencia. De `doc-options={paper=A5}` obtiene dos: `doc-options` y `{paper=A5}`.

`\seq_item:Nn` recupera un elemento de una secuencia por su posición: `N` la secuencia, `n` el número de orden. El `{ 1 }` pide el primero, es decir, el nombre de la clave. Nótese, de paso, que `expl3` cuenta desde uno, no desde cero.

La línea que lo recoge usa `\tl_set:Nx`, y ese sufijo `x` es imprescindible. Recuérdese que `x` significa *expansión completa*. Sin él, la variable guardaría la llamada `\seq_item:Nn ...` sin resolver, en lugar del nombre de clave que esa llamada produce. Es exactamente la misma necesidad que motivaba el `\exp_args:Ne` de `\pyst_set_bool:Nnn`, resuelta aquí por la vía más directa de elegir la variante adecuada de la función.

`\tl_trim_spaces:N` elimina los espacios sobrantes al principio y al final del contenido de una variable. Hace falta porque el usuario bien puede haber escrito `doc-options = {...}`

con espacios alrededor del igual, y el troceo anterior los arrastra consigo: sin recortarlos, la comparación posterior fallaría por un espacio invisible.

`\keys_if_exist:nnF` es la tercera función del módulo `keys` que aparece en esta nota, y cierra el trío: si `\keys_define:nn` declara y `\keys_set:nV` aplica, esta *consulta* si una clave existe en un espacio dado. Sus dos `n` son el espacio y el nombre de la clave; el sufijo `F` indica que solo interesa el caso negativo, que no exista porque es entonces cuando hay que conservar el elemento para KOMA.

Va precedida de `\exp_args:Nne` por el motivo de siempre: el nombre de la clave que hay que consultar está en una variable, y `\keys_if_exist:nnF` espera texto. La firma se lee igual que antes, `N` para la función, `n` para `{ pyst }` literal, `e` para expandir por completo `{ \l_pyst_keyname_tl }` y obtener el nombre real.

Por último, la invocación de la función sobre la lista cruda usa `\exp_args:NV`, la forma más simple de la familia: pasa el primer argumento sin tocar y el segundo por su valor. Y la línea final emplea `\clist_gput_right:NV`, variante de la que ya conocíamos con `V` en lugar de `n`, añade a la lista del usuario el *contenido* de la lista de supervivientes, no su nombre.

Con todo ello, el efecto conjunto es simple de enunciar. Por cada elemento de la lista cruda se trocea por el primer signo igual para obtener el nombre de la clave, y se comprueba si ese nombre existe en `pyst`: si no existe, es de KOMA y se conserva íntegro; si existe, es de PYST-PUB y se descarta, ya se aplicó anteriormente.

Hay una excepción que necesita un trato aparte, y que delata una trampa real, las claves *contenedoras*, recogidas en la lista constante `\c_pyst_container_keys_clist` que tiene como valor `doc-options`, `koma-layout` y `geometry-layout`). En la lista cruda, una opción como `doc-options={paper=A5, DIV=8}` es *un solo elemento*, con sus llaves. Trocearlo por el primer signo igual da `doc-options` como nombre de clave, correctamente, existe en el espacio de nombres `pyst` pero descartar el elemento entero por eso tiraría también las opciones de KOMA que lleva anidadas dentro. Esas ya se encauzaron a la lista correcta durante un paso anterior, mediante `\pyst_koma_route:n`; el filtro solo tiene que dejarlas en paz, no descartarlas por segunda vez. De ahí el `\clist_if_in:NnF` que precede a la comprobación. Si el nombre de clave es una de las tres contenedoras, el elemento no se toca en absoluto, ni para conservarlo ni para descartarlo, su contenido ya viajó por otro camino.

5.8 Bloque 8. El conflicto de *layout*

Un último ajuste, que no lee ninguna clave nueva sino que decide entre dos que ya se aplicaron. Si en algún momento se escribió contenido en las listas de `geometry-layout`, del tipo o del usuario, esa estrategia gana:

```
\bool_if:NT \l_pyst_geo_present_bool
{
  \bool_gset_true:N \g_pyst_geometry_layout_bool
  \bool_if:NT \g_pyst_koma_layout_bool
  {
    \msg_warning:nnnn { pyst } { layout-conflict-override }
    { koma-layout } { geometry-layout }
    \bool_gset_false:N \g_pyst_koma_layout_bool
  }
}
```

```
}
```

Este bloque no introduce funciones nuevas, `\bool_if:NT`, `\bool_gset_true:N` y `\msg_warning:nn` ya han aparecido pero sí un idioma que conviene reconocer. La variable `\l_pyst_geo_present_bool` que gobierna el bloque no la fija ninguna clave: se calcula justo antes, así:

```
\bool_new:N \l_pyst_geo_present_bool
\clist_if_empty:NF \g_pyst_geometry_clist
{ \bool_set_true:N \l_pyst_geo_present_bool }
\clist_if_empty:NF \g_pyst_geometry_layout_user_clist
{ \bool_set_true:N \l_pyst_geo_present_bool }
```

`\clist_if_empty:NF` pregunta si una lista está vacía, con el sufijo `F` porque solo interesa el caso contrario: si tiene contenido, levantar la bandera. Dos comprobaciones sucesivas sobre la misma variable equivalen a un «o lógico»: basta con que cualquiera de las dos listas lleve algo. Es una forma deliberadamente elemental de escribirlo, existen funciones de lógica booleana en el módulo `bool`, que se prefiere por legibilidad cuando, como aquí, las condiciones son solo dos.

Obsérvese también que la bandera es `l_` y se fija con `set`, no con `gset`: no tiene por qué sobrevivir más allá de este bloque, y la convención de nombres lo declara.

Si además había `koma-layout` declarado, se avisa del conflicto y su bandera se baja, las listas de opciones de KOMA no se tocan, solo la bandera que decide qué estrategia gobierna el `layout` más adelante.

5.9 La función `\pyst_validate_keyval:nnn`

La función `\pyst_validate_keyval:nnn` se utiliza para validar un *keyval* respecto a un *espacio de nombres*. La *procedencia* del *keyval* también se utilizará para realizar los mensajes que son la salida de esta función. En su definición forman parte otras funciones presente en un bloque de `pyst-options.ltx`, que transcribimos a continuación,.

```
\cs_new_protected:Npn \pyst_validate_keyval:nnn #1 #2 #3
{
  \keyval_parse:nnn
  { \pyst_validate_key:nnn { #2 } { #3 } }
  { \pyst_validate_pair:nnnn { #2 } { #3 } }
  { #1 }
}
\cs_new_protected:Npn \pyst_validate_key:nnn #1 #2 #3
{
  \clist_if_in:NnTF \c_pyst_scan_only_keys_clist { #3 }
  { \msg_warning:nnnn { pyst } { scan-only-key } { #3 } { #2 } }
  {
    \keys_if_exist:nnF { #1 } { #3 }
    { \msg_warning:nnnn { pyst } { unknown-cfg-key } { #3 } { #2 } }
  }
}
```

```
\cs_new_protected:Npn \pyst_validate_pair:nnnn #1 #2 #3 #4
{ \pyst_validate_key:nnn { #1 } { #2 } { #3 } }
```

La firma `nnn` de `\pyst_validate_keyval:nnn` anuncia tres argumentos de texto:

- n El *keyval* a revisar. Aquí, el contenido de `\g_pyst_type_defaults_tl`.
- n El espacio de nombres contra el que hay que revisarlo. Aquí `{ pyst }`, porque los valores por defecto configuran el documento. Si alguien escribiera `phone=` en `\DeclareDefaultTypeOptions`, esta validación lo denunciaría como clave desconocida, y con razón: `phone` existe, pero en el otro espacio de nombres.
- n La procedencia, que *no interviene en la comprobación* y solo sirve para que el aviso que se generará pueda decir dónde está el problema.

Observemos que en su llamada se ha hecho uso de `\exp_args:NV`. La razón resulta evidente. Si no se expande el primer argumento de la función `\pyst_validate_keyval:nnn`, que recibe como primer parámetro el nombre de una variable, y no el valor de la misma (un *keyval*), la validación que realiza la función (todavía no la hemos descrito), nunca tendría lugar. Pasemos a describir su funcionamiento.

La función nueva del cuerpo es `\keyval_parse:nnn`, del módulo `keyval`, que es la maquinaria de bajo nivel sobre la que está construido el propio `l3keys`. Trocea un *keyval* y, por cada elemento que encuentra, llama a una de *dos* funciones distintas según ese elemento lleve valor o no:

- n La función a llamar cuando aparece una clave *sin* valor. Recibirá un argumento: la clave.
- n La función a llamar para un par `clave=valor`. Recibirá *dos*: la clave y el valor.
- n La cadena a trocear.

Aquí está la razón de que haya tres funciones para lo que conceptualmente es una sola pregunta. Quien de verdad valida es `\pyst_validate_key:nnn`, que necesita saber tres cosas: el espacio de nombres, la procedencia y la clave, que son, respectivamente, los parámetros `#2` y `#3`. En su llamada, vemos que esos son, aparentemente, los únicos valores que se les pasa. Sin embargo, en lo que se conoce como una *aplicación parcial* el comando `\keyval_parse:nnn` trocea el *keyval* que recibe y lo añade como tercer parámetro; esto es, la clave que acaba de extraer del *keyval*, en el caso de un argumento en el *keyval* que no tenga valor. Es decir, `\pyst_validate_key:nnn` recibe tres parámetros que son, respectivamente, el espacio de nombres (`#1`), la procedencia (`#2`) y la clave a validar, (`#3`).

En el caso de una clave con valor, `\keyval_parse:nnn` añade como tercer y cuarto parámetro a `\pyst_validate_pair:nnnn` esta clave y el valor. Así, `\pyst_validate_pair:nnnn` recibe cuatro parámetros pero descarta el valor, `#4` que no aparece en su cuerpo (porque para validar un *nombre* de clave el valor es irrelevante) y reenvía los tres que importan a `\pyst_validate_key:nnn`.

`\pyst_validate_key:nnn` no hace una comprobación sino dos, y emite avisos distintos. La primera consulta `\c_pyst_scan_only_keys_clist`, la lista de las claves constante que solo tienen sentido en `\documentclass` formada por `doc-type` y `dept-name`. En un fichero `.cfg` no son una errata sino un malentendido existen, pero no ahí, y por eso merecen un mensaje

propio, `scan-only-key`, que puede explicar exactamente eso en lugar de limitarse a decir que la clave no se reconoce.

La segunda, en la rama contraria, es `\keys_if_exist:nnF`, que pregunta si una clave está declarada en un espacio de nombres dado. Su sufijo `F` indica que solo interesa el caso negativo. Si la clave **no** existe, se emite `unknown-cfg-key`, nombrando la clave y la procedencia.

Es importante señalar que los mensajes emitidos por la función son `\msg_warning` y no `\msg_error`. Es decir, una errata en un *keyval* de una determinada *procedencia*, deja el documento sin ese ajuste, pero no impide componerlo. Además, la responsabilidad de la función es de validación, *no repara* nada. Recorre, comprueba y avisa; la clave errónea seguirá su curso y acabará, un instante después, en el `unknown` silencioso del espacio de nombres. Lo único que cambia es que ahora habrá quedado constancia en el `.log`.

6 La entrega a KOMA

Con los seis bloques del motor de opciones completos, la fase PRE ya tiene todo lo que necesita: una clase base elegida, dos listas de opciones para ella y un puñado de banderas internas. Lo que le falta es convertir ese estado en algo que el resto del sistema pueda usar.

Ese es el cometido de `pyst-build.ltx`, el último fichero de la fase PRE y el único que se ejecuta con la fase de lectura ya cerrada. No decide nada: no lee catálogos, no interpreta claves del usuario, no resuelve precedencias. Todo eso ocurrió antes. Su trabajo es de *traducción*, y traduce en dos direcciones a la vez:

- Hacia KOMA. Convierte las estructuras internas de `expl3` en las cadenas de texto plano que `\LoadClass` y `\KOMAOPTIONS` saben leer.
- Hacia el resto de PYST-PUB. Proyecta los booleanos de `expl3` a condicionales `\newif` clásicos, para que los ficheros de la fase POST —escritos casi todos en $\TeX$ corriente— puedan consultarlos sin saber que por debajo hay `expl3`.

Dicho de otro modo: `pyst-build.ltx` es la frontera entre el mundo en el que se ha escrito la fase PRE y el mundo en el que está escrito casi todo lo demás. Es un fichero de aduana.

Está organizado en ocho bloques numerados, y conviene tener el mapa delante antes de entrar en el detalle:

Bloque	Qué produce	Comentario
—	<code>ifPystBackground</code>	Conmutador temprano, fuera de numeración
1	<code>PystBaseClass, PystClassOptions</code>	El ensamblado, y el destino de todo el motor de opciones
2	—	<i>advanced-math</i> implica <i>math</i>
3	Nueve <code>ifPyst...</code>	La proyección de los booleanos
4	<code>ifPystCap</code>	Deducción: la clase base tiene capítulos o no
5	—	Dos comprobaciones defensivas, ambas inalcanzables
6	—	Configuración de babel por defecto
7	<code>PystGeoConfig, PystBabelConfig</code>	Geometría y babel, como cadenas
8	Seis accesores más	Lo que queda por exponer

Los bloques 1, 3, 7 y 8 producen la interfaz pública. Los bloques 2, 4, 5 y 6 no producen nada visible: rellenan huecos y comprueban coherencias antes de que alguien más pueda leerlas.

6.1 Bloque 1. Ensamblado de la lista de opciones

Las dos listas de opciones de KOMA se funden en el orden que ya se explicó, tipo primero y usuario después:

```
\clist_if_empty:NF \g_pyst_type_opts_clist
{ \clist_put_right:NV \g_pyst_final_opts_clist \g_pyst_type_opts_clist }
\clist_put_right:No \g_pyst_final_opts_clist { \g_pyst_user_opts_clist }
```

Ese orden no es una preferencia estética: es el mecanismo entero de la precedencia. KOMA resuelve una clave que aparece dos veces conservando la *última* aparición, de modo que basta con escribir las opciones del usuario detrás de las del tipo para que las del usuario ganen. No hay código de comparación, no hay tabla de prioridades; hay un orden de concatenación. Es la misma economía que se ha visto en toda la fase PRE: no reimplementar lo que la herramienta de destino ya hace.

Estas tres líneas cierran además el vocabulario de expansión de la nota, y lo hacen mostrando dos variantes de la misma operación. `\clist_put_right:NV` añade a la lista de destino el *valor* de otra lista; `\clist_put_right:No` hace lo mismo mediante una sola expansión de su argumento. Aquí el efecto es equivalente, y la diferencia es de matiz: **V** está pensado específicamente para «toma el valor de esta variable», mientras que **o** es más general y expande un paso lo que se le dé, sea lo que sea. Que aquí convivan las dos es una pequeña inconsistencia de estilo, sin consecuencias.

Adviértase igualmente que son `put_right` y no `gput_right`: sin la **g** de global, la asignación es local. Basta, porque `g_pyst_final_opts_clist` se consume unas líneas más abajo, dentro del mismo fichero y sin que medie ningún grupo.

6.2 El conflicto de *layout*, otra vez

Inmediatamente después del ensamblado, y antes de construir las constantes, el fichero vuelve sobre el conflicto entre *koma-layout* y *geometry-layout* que el bloque 8 del motor de opciones ya había resuelto:

```
\bool_lazy_or:nnT
{ ! \clist_if_empty_p:N \g_pyst_geometry_clist }
{ ! \clist_if_empty_p:N \g_pyst_geometry_layout_user_clist }
{ \bool_gset_true:N \g_pyst_geometry_layout_bool }

\bool_lazy_and:nnT { \g_pyst_koma_layout_bool } {
  \g_pyst_geometry_layout_bool }
{
  \msg_warning:nnnn { pyst } { layout-conflict-override }
  { koma-layout } { geometry-layout }
  \bool_gset_false:N \g_pyst_koma_layout_bool
}
```

Conviene decir con claridad lo que ocurre aquí, porque es un caso instructivo: **este código no se ejecuta nunca**. El bloque 8 de *pyst-options.ltx* se carga antes que este fichero y resuelve ya el mismo conflicto, así que al llegar aquí sólo caben dos situaciones y en las dos la conjunción de abajo es falsa:

- Hubo geometría. El bloque 8 ya bajó la bandera de *koma-layout*, luego la conjunción falla en su primer argumento.
- No hubo geometría. Entonces nadie levantó la de *geometry-layout*, y falla en el segundo.

Comprobado por compilación: en un conflicto real —un tipo con *geometry-layout* más un *koma-layout* del usuario— el aviso se emite una sola vez, no dos. Y el *\bool_lazy_or* de arriba tampoco aporta nada: levanta una bandera que el bloque 8 ya había levantado.

Se conserva deliberadamente. Funciona, es inocuo, y cuál de las dos copias debe sobrevivir es una decisión de diseño abierta. La de aquí está en el punto donde se ensambla la lista de opciones, que es un lugar defendible para resolver el conflicto. Queda a criterio de quien mantenga el sistema.

Y una precisión sobre qué significa aquí que «*koma-layout* pierde»: pierde la bandera, no las claves. Las que el usuario escribió dentro de *koma-layout=* ya se enrutaron a *\g_pyst_user_opts_clist* al procesarlas, y siguen llegando a *\KOMAOPTIONS*. No se purgan a propósito: *geometry* se carga después con valores explícitos e invalida por completo el *DIV* y el *BCOR* que hayan pasado, se den como se den. Geometría gana porque es la opción más intuitiva y la que previsiblemente elegirá el usuario típico.

Dos funciones nuevas de *expl3*, ambas útiles más allá de este caso. *\bool_lazy_and:nnT* y *\bool_lazy_or:nnT* evalúan dos condiciones y ejecutan el tercer argumento si se cumple la conjunción o la disyunción. Lo de *lazy* es literal: la evaluación es *perezosa*, es decir, si el primer argumento ya decide el resultado, el segundo no se evalúa. En un *and*, un primer argumento falso basta; en un *or*, un primero verdadero. Importa cuando el segundo argumento es costoso o cuando sólo tiene sentido evaluarlo si el primero se cumple.

La otra novedad es el sufijo `_p`, en `\clist_if_empty_p:N`. Un nombre que acaba en `_p` es un **predicado**: no ramifica por sí mismo, sino que *devuelve* un valor lógico para que otra función lo combine. De ahí que aquí vaya precedido de `!`, la negación, dentro del argumento del `bool_lazy_or`. Es la contrapartida de las formas `:NTF` y `:NT`, que sí ramifican por sí mismas. Casi toda función condicional de `expl3` ofrece ambas presentaciones, y la elección depende de si se quiere actuar o se quiere componer.

6.3 Las dos constantes públicas

Con la lista ya ensamblada, el bloque 1 construye las dos constantes que son el destino de todo el motor de opciones:

```
\tl_const:Nx \PystBaseClass { \tl_use:N \g_pyst_base_class_tl }
\tl_const:Nx \PystClassOptions { \clist_use:Nn \g_pyst_final_opts_clist { ,
} }
```

Tres funciones nuevas, las tres relacionadas con *sacar* un valor del sistema de variables para dejarlo disponible al mundo exterior.

`\tl_const:Nx` crea una **constante**: una variable de texto que se fija una sola vez y no volverá a cambiar durante la compilación. Nótese que no lleva `g` de global, porque las constantes lo son por naturaleza, y que el sufijo `x` vuelve a significar expansión completa: lo que se guarda no es la llamada a función que aparece entre llaves, sino el texto que esa llamada produce.

`\tl_use:N` es la operación de *leer* una variable de texto: entrega su contenido allí donde se escriba. Su papel aquí es proporcionar, dentro del argumento expandido, el valor de la clase base.

`\clist_use:Nn` hace lo propio con una lista, pero con una diferencia importante: como una lista tiene varios elementos, hay que decirle qué poner *entre* ellos. Ese es su segundo argumento, `{ , }`, la coma que separará las opciones. La función convierte así una estructura interna de `expl3` en la cadena de texto plano que $\text{\LaTeX}2\epsilon$ y KOMA saben leer, que es exactamente el propósito de todo este bloque.

El resultado son dos comandos expandibles, pensados para escribirse directamente donde $\text{\LaTeX}$ espera un nombre de clase o una lista de opciones. Un detalle que importará más adelante: el contenido de `PystClassOptions` se tokenizó *aquí*, bajo catcodes de `expl3`. Los espacios que hubiera en las opciones ya desaparecieron en este punto, no en el momento de usarlas.

6.4 Bloque 3. La proyección a condicionales `\newif`

Es el bloque más repetitivo del fichero y el más importante para todo lo que viene después. Cada booleano que la fase PRE ha fijado recibe aquí un condicional $\text{\LaTeX}2\epsilon$ equivalente, siempre con la misma forma: declarar el condicional (que nace falso) y levantarlo si el booleano subyacente es verdadero.

```
\newif\ifPystMath
\bool_if:NT \g_pyst_module_math_bool { \PystMathtrue }
```

Nueve pares idénticos a ése, más tres que se derivan de otra cosa:

Condicional	Se deriva de	Cómo
<code>ifPystMath</code>	<code>g_pyst_module_math_bool</code>	<code>bool_if:NT</code>
<code>ifPystAdvancedMath</code>	<code>g_pyst_module_advanced_math_bool</code>	<code>bool_if:NT</code>
<code>ifPystGraphics</code>	<code>g_pyst_module_graphics_bool</code>	<code>bool_if:NT</code>
<code>ifPystGeometryLayout</code>	<code>g_pyst_geometry_layout_bool</code>	<code>bool_if:NT</code>
<code>ifPystDebug</code>	<code>g_pyst_debug_bool</code>	<code>bool_if:NT</code>
<code>ifPystCompat</code>	<code>g_pyst_compat_bool</code>	<code>bool_if:NT</code>
<code>ifPystCopyRight</code>	<code>g_pyst_copyright_bool</code>	<code>bool_if:NT</code>
<code>ifPystCap</code>	<code>g_pyst_has_chapters_bool</code>	<code>bool_if:NT</code>
<code>ifPystDept</code>	<code>g_pyst_current_dept_tl</code>	<code>tl_if_empty:NF</code>
<code>ifPystEnglish</code>	<code>g_pyst_language_tl</code>	<code>str_if_eq:VnT</code>
<code>ifPystMinion</code>	<code>g_pyst_font_tl</code>	<code>str_if_eq:VnT</code>

Los tres últimos merecen atención porque no vienen de un booleano. `ifPystDept` no se deriva de una bandera «hay departamento», sino de si `g_pyst_current_dept_tl` está vacía; el bloque 5 del motor de opciones sólo la rellena cuando el `prop_if_in:NVTF` encuentra coincidencia, así que la variable vacía es la bandera. `ifPystEnglish` y `ifPystMinion` comparan una cadena con un valor literal mediante `\str_if_eq:VnT`, donde `V` toma el valor de la variable y `n` deja el literal sin expandir. Es un patrón que ahorra declarar un booleano para cada caso particular, a cambio de repetir la comparación en cada sitio donde haga falta.

Por qué todos los nombres son sólo letras

Hay una regla de nomenclatura en este bloque que no es evidente y que conviene entender, porque afecta a cualquier condicional que se añada en el futuro: los doce se llaman *exclusivamente con letras*. Ningún `_`, ningún `:`.

No es por capricho ni por elegancia. `\newif` funciona perfectamente bajo `\ExplSyntaxOn`, en contra de lo que sugiere la intuición: no lee el nombre del conmutador como texto, sino que lo obtiene con `\string` y reconstruye `...true` y `...false` con `\csname`, al que le es indiferente el catcode de unos caracteres que ya son caracteres. Bajo `expl3` funcionan nombres de sólo letras, con espacios intercalados, con guion bajo y con `@`.

El peligro está en otro sitio: en declarar un condicional en un régimen y *usarlo* en el otro. Un conmutador llamado `ifpyst_flag` es una única secuencia de control bajo `expl3`, donde `_` es letra; leído en régimen `ℒTEX2e`, donde `_` es subíndice, se parte y produce

```
! Undefined control sequence.
! Missing $ inserted.
```

Y ése es exactamente el recorrido de estos doce condicionales: se declaran aquí, bajo `expl3`, y se consultan a lo largo de toda la fase POST en régimen `ℒTEX2e`. La restricción a letras es lo que hace posible la traducción que da sentido al fichero.

6.5 Bloques 2 y 4. Los hechos que se deducen

Dos bloques cortos que no producen interfaz, sino coherencia. Se ejecutan antes de que nadie pueda leer lo que arreglan, que es la única razón de que estén donde están.

El bloque 2 impone que `advanced-math` implique `math`. Si el usuario pidió las matemáticas avanzadas sin pedir las básicas, se le conceden las básicas en silencio:

```
\bool_if:NT \g_pyst_module_advanced_math_bool
{ \bool_gset_true:N \g_pyst_module_math_bool }
```

En silencio y no con un aviso, porque no hay ambigüedad sobre la intención: quien pide lo avanzado quiere lo básico. Avisar sería señalar como error algo que el sistema sabe resolver de una única manera posible.

El bloque 4 deduce si el documento tiene capítulos. No es una clave que el usuario ni el tipo fijen: es un hecho de la clase base, y por tanto se lee de ella.

```
\str_case:VnF \g_pyst_base_class_tl
{
  { scrbook } { \bool_gset_true:N \g_pyst_has_chapters_bool }
  { scrreprt } { \bool_gset_true:N \g_pyst_has_chapters_bool }
}
{ \bool_gset_false:N \g_pyst_has_chapters_bool }
```

`\str_case:VnF` es la construcción de selección múltiple de `expl3`: compara el valor de la variable contra una lista de casos y ejecuta el que coincida. El sufijo `F` da la salida por defecto —*false*, ninguna coincidencia— que aquí corresponde a `scrartcl`, la única de las tres clases de KOMA que no tiene capítulos. Nótese que no se enumera `scrartcl` como caso: se deja caer en la rama por defecto, que es más robusto frente a una clase base inesperada.

El resultado se proyecta a `ifPystCap`, cuyo nombre —por «capítulo»— es más breve que descriptivo, y que la fase POST consulta en abundancia: los estilos de capítulo, la numeración de ecuaciones y los estilos de página dependen de él.

6.6 Bloque 5. Las comprobaciones defensivas

El bloque 5 comprueba dos cosas: que `advanced-math` no esté activo con `math` apagado, y que la clase base sea una de las tres de KOMA. Emite un aviso y corrige en el primer caso, y sólo un aviso en el segundo.

Ninguna de las dos ramas puede dispararse en el sistema actual, y el propio fichero lo dice: el primer conflicto acaba de reconciliarlo el bloque 2, unas líneas más arriba; y `base-class` sólo llega hasta aquí con uno de los tres valores admitidos, porque su clave en `pyst-configuration.ltx` ya lo valida contra esa misma lista y se repliega a `scrartcl` en caso contrario.

Es la segunda vez en este fichero que aparece código inalcanzable, y la comparación entre los dos casos es instructiva. El del conflicto de *layout* es una duplicación de lógica: dos sitios distintos resuelven lo mismo, y sobra uno. Éste es otra cosa: es una segunda línea de defensa deliberada, escrita para el caso de que la garantía de más arriba cambie. La diferencia práctica está en qué ocurriría si la condición llegara a cumplirse. Aquí, un aviso útil. Allí, un aviso duplicado.

Que un código sea inalcanzable no lo hace inútil, pero sí obliga a declarar por qué se conserva. Las dos veces está declarado en los comentarios del fichero, y ésa es la razón de que se puedan distinguir.

6.7 Bloques 6 y 7. Geometría y babel, como cadenas

El bloque 6 rellena un hueco. Si ni el tipo ni el usuario escribieron nada en `babel`, se aplica aquí una configuración española completa, para que la constante que viene después no quede vacía:

```
\clist_if_empty:NT \g_pyst_babel_clist
{
  \clist_gset:Nn \g_pyst_babel_clist
    { spanish, es-ucroman, es-noshorthands, es-nosectiondot,
      es-noindentfirst, es-nolists, shorthands=off }
}
```

El bloque 7 convierte a cadenas de texto la geometría y babel, igual que el bloque 1 hizo con las opciones de KOMA. Pero hay aquí una decisión de diseño que merece explicación:

```
\clist_new:N \l_pyst_geometry_merged_clist
\clist_concat:NNN \l_pyst_geometry_merged_clist
  \g_pyst_geometry_clist \g_pyst_geometry_layout_user_clist
\tl_const:Nx \PystGeoConfig
{ \clist_use:Nn \l_pyst_geometry_merged_clist { , } }
\tl_const:Nx \PystBabelConfig { \clist_use:Nn \g_pyst_babel_clist { , } }
```

`\clist_concat:NNN` concatena dos listas en una tercera. Lo relevante es que *concatena*, no fusiona clave por clave: las dos listas se conservan enteras, una detrás de la otra. Y eso es exactamente lo que se quiere, por la misma razón que en el bloque 1. Si el usuario escribe en `\documentclass` una sobreescritura parcial —digamos sólo `left=10mm`—, todos los demás márgenes que el tipo declaró deben seguir en pie. Fusionar clave por clave obligaría a decidir qué hacer con las claves que sólo aparecen en una de las dos listas; concatenar deja que *geometry* aplique su propia regla de última aparición, que es la que el usuario espera.

Es el mismo principio que gobierna toda la fase PRE, aplicado por tercera vez: la precedencia se implementa como un orden, y la resuelve el paquete de destino.

6.8 Bloque 8. Los accesorios públicos

El último bloque expone, uno por uno, los datos que la fase POST necesita leer sin pasar por `expl3`:

```
\cs_new:Npn \PystCurrentType { \tl_use:N \g_pyst_current_type_tl }
\cs_new:Npn \PystFontName { \tl_use:N \g_pyst_font_tl }

\tl_const:Ne \c_pyst_language_tl { \tl_use:N \g_pyst_language_tl }

\cs_new:Npn \PystLanguageStr { \tl_use:N \c_pyst_language_tl }
\cs_new:Npn \PystCoverStyleFile { \tl_use:N \g_pyst_coverstyle_file_tl }
\cs_new:Npn \PystDocStyle { \tl_use:N \g_pyst_doc_style_tl }

\NewDocumentCommand { \PystDefaultPageStyle } { } { scrheadings }
```

`\cs_new:Npn` define una función nueva. Nótese la ausencia de `_protected`: estos accesores son **expandibles** a propósito, porque se usan dentro de argumentos que se expanden —nombres de fichero, comparaciones de cadenas— y un comando protegido se quedaría ahí sin expandir. Es la contrapartida de `\cs_new_protected:Npn`, que es lo que se usa para todo lo que tenga efectos laterales.

Aparece también un sufijo de expansión nuevo. `\tl_const:Ne` usa **e** donde el bloque 1 usaba **x**. Los dos significan expansión completa del argumento y el resultado es el mismo; la diferencia es de implementación, no de semántica: **e** emplea la primitiva de expansión del motor, disponible en los motores modernos, mientras que **x** se apoya en un `\edef`. En código nuevo, **e** es la forma preferida. Que este fichero use las dos es, otra vez, una inconsistencia de estilo sin consecuencias.

Hay una singularidad en la lista. `PystLanguageStr` no lee directamente la variable global, sino que pasa por una constante intermedia, `c_pyst_language_tl`, creada en la línea de arriba. El prefijo `c_` es el que `expl3` reserva para las constantes, frente a `g_` para las globales y `l_` para las locales. Congelar el idioma en una constante antes de exponerlo impide que nadie lo cambie a mitad de documento, lo que en un sistema con babel ya configurado sería una fuente de incoherencias difíciles de rastrear.

Y una última observación, que no es sobre `expl3` sino sobre el dato: `PystLanguageStr` devuelve `spanish` o `english`, los nombres que usa babel. No son códigos de idioma de la norma BCP-47, que serían `es-ES` y `en-GB`. Mientras el consumidor sea babel, es lo correcto. Si algún día el sistema declara el idioma del documento en los metadatos del PDF —donde la norma exige BCP-47—, hará falta una traducción, y este accesor es el sitio natural para ella.

El bloque cierra con `PystDefaultPageStyle`, definido con `\NewDocumentCommand` en lugar de con `cs_new:Npn`. No responde a ninguna variable: es un valor literal, `scrheadings`, que los estilos de documento redefinen si quieren otro. Un documento que escriba `\pagestyle{\PystDefaultPageStyle}` obtiene así el estilo que le corresponda sin tener que saber cuál es.

6.9 Cargar, configurar, recalcular

Todo lo anterior existe para que la entrega final, ya en `pyst-pub.cls`, quepa en cinco líneas:

```
\LoadClass{\PystBaseClass}
\ExplSyntaxOff
\KOMAOptions{\PystClassOptions}
\recalcctypearea
\ExplSyntaxOn
```

Son cinco y no tres, y cada una de las diferencias tiene su motivo.

Por qué `\KOMAOptions` y no `\PassOptionsToClass`

La clase se carga primero, *sin* opciones. Después se le entregan todas mediante `\KOMAOptions`, el comando que KOMA-Script ofrece para reconfigurar una clase ya cargada. Es una forma menos inmediata de escribirlo que `\PassOptionsToClass` seguido de `\LoadClass`, y la razón para preferirla es concreta, no de estilo.

KOMA analiza algunas claves —`paper` entre ellas— dentro de su propio `\DeclareOption*`, leyendo `\CurrentOption`, un valor que sólo tiene sentido durante el bucle `\ProcessOptions de la propia clase`. Las opciones entregadas con `\PassOptionsToClass` nunca entran en ese bucle: quedan registradas, pero jamás se llegan a aplicar, y acaban en `\@unusedoptionlist` sin que nada avise en el momento. `\KOMAOPTIONS`, en cambio, procesa cada clave por el camino normal de KOMA, con el mismo efecto que tendría haberla escrito directamente entre los corchetes de `\documentclass`.

Por qué `\recalctypearea`

`\KOMAOPTIONS` aplica cada clave según la va leyendo, y `DIV` se recalcula al instante. Supóngase que `paper` aparece *después* de `DIV` en la misma lista, lo que ocurre siempre que el tipo declara un `DIV` propio y el usuario pide, además, un tamaño de papel. El área de texto queda entonces calculada para el papel anterior y no se vuelve a recalcular. El resultado sería un papel del tamaño correcto con un bloque de texto pensado para otro.

`\recalctypearea` es el comando que KOMA ofrece exactamente para esta situación, y se invoca sin condición: si no hubo cambio de papel, recalcula los mismos valores que ya había, y no cuesta nada haberlo llamado de más.

Por qué `\ExplSyntaxOff`

Las dos líneas de `\ExplSyntaxOff` y `\ExplSyntaxOn` son nuevas, y corrigen un defecto que todo documento del sistema arrastraba en silencio. `\ProvidesExplClass` deja los catcodes de `expl3` activos durante todo el fichero de clase, de modo que `\KOMAOPTIONS` se ejecutaba bajo ellos. La comprobación interna de versión de KOMA tokeniza su cadena de referencia con los catcodes en vigor, y con los espacios ignorados el resultado era un aviso en cada compilación:

```
LaTeX Warning: \KOMAScriptVersion is`2023/07/07 v3.41 KOMA-Script',
but`2023/07/07v3.41KOMA-Script'wasexpected!
Youshouldnotuseclasses,packagesorfilesfrom
differentKOMA-Scriptversions.
```

Las dos cadenas comparadas son la misma cadena. La «esperada» aparece sin espacios porque se leyó bajo `expl3`. Nada estaba mal en la instalación: la clase leía la contabilidad interna de KOMA con la tabla de catcodes equivocada, y el aviso resultante señalaba a un problema inexistente y bastante alarmante.

`\LoadClass` no está afectado y se queda dentro de `expl3`: la comprobación no salta al cargar la clase, sino en los ficheros `.clo` que `\KOMAOPTIONS` dispara. `\recalctypearea` se mantiene dentro de la misma isla de `LaTeX2e` porque pertenece a la misma operación.

Un detalle que deja de ser inocuo

Nótese que los argumentos van pegados a las llaves: `{\PystClassOptions}` y no `{ \PystClassOptions }`. Mientras `\KOMAOPTIONS` se ejecutaba bajo `expl3`, los espacios interiores se ignoraban al tokenizar y las dos formas eran el mismo argumento. En cuanto la línea pasa a régimen `LaTeX2e` dejan de serlo, y KOMA recibe un nombre de opción que empieza y acaba por espacio:

```
./pyst-pub.cls:98: Package scrbase Error: unknown option `
\pystClassOptions '.
```

Es un buen recordatorio de algo que atraviesa toda esta nota: cambiar el régimen de una línea obliga a revisar todos los espacios que contiene, no sólo su lógica. Los espacios son código.

6.10 Qué queda al final de la fase PRE

Cuando `pyst-build.ltx` termina, la fase PRE ha dejado de existir como tal. Lo que queda es una interfaz, y toda ella es $\text{\LaTeX}$ ordinario:

Qué	Cuántos
Constantes de texto expandibles <code>PystBaseClass</code> , <code>PystClassOptions</code> <code>PystGeoConfig</code> , <code>PystBabelConfig</code>	4
Condicionales <code>ifPyst...</code>	12
Accesores expandibles	6

Veintidós nombres. Ni uno de ellos exige de quien los use el menor conocimiento de `expl3`: son constantes que se expanden, condicionales que se prueban con `\if` y comandos que devuelven una cadena. Toda la maquinaria de `prop`, de `clist`, de `keys_define:nn` y de los sufijos de expansión ha quedado del otro lado de la aduana.

Las cinco líneas de la entrega, tomadas juntas, son la regla fundamental de esta nota hecha código: cargar la clase, entregarle las opciones por su propio cauce, y dejar que sea ella —no PYST-PUB— quien decida las consecuencias sobre el papel y la caja de texto. Lo mismo vale para la interfaz completa: la fase PRE no impone nada a la fase POST, le entrega hechos.

7 Resumen

En un entorno académico hay una amplia variedad de documentos de naturaleza muy distinta y con objetivos bien diferenciados. No es lo mismo un examen que un certificado de asistencia al mismo. O las necesidades de un profesor al diseñar unos apuntes que las de un estudiante al escribir su TFG o TFM; o no es igual la dificultad de escritura de un texto matemático que el utilizado en una Facultad de Historia.

Todos ellos tienen, sin embargo, un hecho en común. Todos pueden compartir un conjunto de variables que deben quedar reflejadas en el texto. Nos referimos a datos tales como el autor y el título del mismo, la fecha, el departamento en el que se está adscrito, el centro, y así sucesivamente. Por no decir de las características es sí del texto: tipografía, tamaño del cuerpo del documento en el que es escribe, interlineado, portada, etc, etc.

Cuando se opta por utilizar $\text{\LaTeX}$ para abordar esta tarea, nos encontramos con una serie de herramientas a nuestra disposición que resuelven algunos de esos aspectos. Pero, evidentemente, no todos. $\text{\LaTeX}$, sus clases de documentos y sus paquetes facilitan enormemente la labor que tenemos por delante pero no está pensado para resolver fácilmente problemas concretos de una situación concreta.

Existen paquetes como Beamer, o KOMA scripts que ayudan a resolver muchos detalles pero cualquier usuario de $\text{\LaTeX}$ conoce la necesidad de integrar, junto a esos paquetes generalistas, una buena cantidad de otros que resuelvan elementos como la creación de hiperenlaces, gestión de índices, color en determinados elementos, creación de gráficos, etc.

El propósito de un sistema como PYST-PUB es facilitar toda esta integración entre datos, tipos de documentos, paquetes específicos, tipografía, diseño de capítulos, creación de gráficos, flexibilidad del lenguaje, etc. Es lo que se denomina en la terminología de $\text{\LaTeX}$, una *clase envolvente* de una más genérica, en este caso KOMA.

La escritura de esta clase envolvente se ha dividido en dos etapas bien diferenciadas. Por un lado, la gestión de la entrada de datos y selección de tipo de documento, hasta invocar la clase KOMA que se desea utilizar. A esta etapa la hemos denominado PRE. Y, de manera bastante obvia, la etapa POST se encarga de gestionar todo lo anterior junto con la carga de paquetes adicionales que hemos considerado necesarios para generar un documento con niveles de calidad elevados lo más adecuado posible para satisfacer nuestros estándares de calidad.

7.1 En qué consiste la fase PRE

En este documento hemos recorrido cada uno de los apartados de la fase PRE del sistema PYST-PUB. Ahora que tenemos una visión clara de su contenido, conviene explicar todo lo que se ha hecho de un vistazo, para darle sentido al desarrollo completo.

Como hemos dichos, el problema de partida de la fase PRE se enuncia de la manera siguiente: *hay un conjunto de valores de un conjunto de claves descritos en varios sitios que hay que llevarlos hasta unas variables internas del sistema*. Eso es todo. En un ejemplo concreto, cuando un usuario desea utilizar una tipografía concreta, (**Paso 1**), debe escribir, por ejemplo, `\documentclass[doc-type=texto, font=minion]{pyst-pub}` y eso debe traducirse en que algún elemento en la fase POST indague si es posible cargar un paquete específico de $\text{\LaTeX}$ y gestionar la carga de un fichero específico de fuente (**Paso 2**), en este caso, un fichero (o varios) que permitan escribir el documento con esa tipografía. Y, si además, vamos a utilizar ecuaciones en nuestro documento, hemos debido escribir algo como, (**Paso 1**), `\documentclass[doc-type=texto, font=minion, math=true]{pyst-pub}` o `\documentclass[doc-type=texto, font=minion, math]{pyst-pub}`, que haga replantearse lo anterior y no solo se carguen unos ficheros para el texto sino que, además, **Paso 2**, se carguen ficheros para utilizar todos los símbolos que distinguen a un documento que gestiona ecuaciones matemáticas de otros que no. Entre los **Paso 1** y **Paso 2** se encuentra esta fase PRE.

Que ese trayecto tan corto de enunciar exija todas los pasos que hemos visto en los apartados previos de este documento, no es un capricho de diseño. Cada paso responde a una dificultad concreta, y vale la pena verlos en orden.

7.2 Los valores vienen de cuatro fuentes diferentes

¹³Podríamos haber escrito `\documentclass[DIV=12, twoside, paper=A4]{scrbook}` y KOMA habría generado un documento de la clase `scrbook` con una gran calidad. Eso sí, tal vez sin la tipografía que nos guste, o siendo incapaz de definir las características editoriales del libro, etc.

Si el único origen de los valores que se desean incorporar fuera `\documentclass`, no haría falta casi nada de lo que sigue¹³. Pero hay cuatro fuentes de posibles valores para construir un documento, repartidas en tres ficheros y no por casualidad.

1. **Los valores por defecto** de cualquier tipo de documento, declarados con el comando `\DeclareDefaultTypeOptions`. Se definen en `pyst-types.cfg` pero no pertenecen a ningún tipo de documento específico, sino que se aplican a todos los tipos de documentos que se creen.
2. **El tipo de documento** elegido, uno de los creados con `\DeclareDocumentType` en el fichero `pyst-types.cfg`. Cualquier documento que se genere debe definir *obligatoriamente* qué tipo de documento se desea generar.
3. **El departamento**, elegido entre los declarados con `\DeclareDepartment` en `pyst-depts.cfg`. Es un valor opcional. Se pueden crear documentos sin elegir ningún departamento y utilizar los metadatos que se usan en la declaración de un departamento.
4. **El propio usuario**, con un conjunto de opciones definidas en `\documentclass`. Estas opciones son propias del sistema PYST-PUB o del sistema KOMA.

Y no solo son esas cuatro fuentes de opciones. Además, existe un *orden concreto* en el que se procesan. Si un tipo de documento se ha definido en la forma

```
\DeclareDocumentType{leccion}{
  base-class      = scrreprt,
  doc-style       = book,
  coverstyle      = guion,
  koma-layout     = {DIV=10, BCOR=0mm},
  advanced-math   = true,
  modules         = {pyst-book-common}
```

y a la hora de construir el documento el usuario declara

```
\documentclass[%
  doc-type=leccion,
  doc-style=book04,
  font=minion,
  modules=pyst-manual,
  DIV=12,
  fontsize=11pt
]{pyst-pub}
```

el documento se compondría con `DIV=12` y `modules={pyst-book-common, pyst-manual}`. Es decir, las opciones del usuario sustituyen o se incorporan a las opciones del tipo de documento.

7.3 Primer paso. Guardar sin entender

Los dos catálogos se leen enteros, pero no se interpretan. `\DeclareDocumentType` no ejecuta nada, se limita a guardar el texto del *keyval*, tal cual, en una lista de propiedades, bajo la clave del tipo de documento. Lo mismo hace `\DeclareDepartment` con el suyo, utilizando como clave el acrónimo del departamento.

La razón es que en ese momento todavía no se sabe qué tipo va a usarse. Interpretar los tipos del catálogo para acabar usando uno sería trabajo desperdiciado, y algo peor: los restantes dejarían por el camino sus efectos colaterales sobre variables que nadie ha pedido tocar. Guardar en bruto pospone la interpretación hasta que se sepa cuál hace falta.

Al terminar esta etapa el sistema tiene los valores, pero no ha hecho nada con ellos. Sigue sin saber siquiera qué documento se le está pidiendo.

7.4 Segundo paso. Declarar un espacio de nombres

Aquí aparece la pieza que más cuesta entender al llegar de otros lenguajes de programación. El sistema declara, con `\keys_define:nn`, un conjunto de *claves propias* y les asigna a cada una un comportamiento: qué hacer cuando alguien les dé a esa clave un valor. La clave `font` del sistema no guarda nada por sí misma; lo que tiene es una instrucción como *normaliza lo que te den y déjalo en `\g_pyst_font_tl` que se ejecutará si alguien la utiliza y cuando alguien la declare*¹⁴. A este conjunto de claves definidas por el sistema se les denomina *namespace* o *espacio de nombres*.

Conviene utilizar una imagen para no confundirse con los nombres de las claves declaradas en un espacio de nombres o namespace. Cada clave tiene, por así decirlo, un *pronombre* y un *nombre*. El nombre es `font`, `doc-style`, `phone`: el mismo que declara el usuario o se utiliza en uno de los catálogos. El *pronombre es el espacio de nombres al que pertenece*: `{ pyst }` o `{ pyst / dept }`. El emparejamiento entre las claves propias y las claves del usuario se hace por el nombre, dentro del pronombre (espacio de nombres) que se indique.

En el sistema se han definido dos espacios de nombres para evitar que las claves definidas en los dos ficheros de catálogos no compartieran el mismo pronombre y lograr con esa separación la posibilidad de realizar asignaciones cruzadas. Es una decisión de diseño y su objetivo es distinguir de manera clara dos tipos de fuentes de datos utilizadas por el usuario.

7.5 Tercer paso. Volcar unas claves sobre las otras

Con los valores asignados a las claves por un lado, provenientes de las cuatro posibles fuentes que hemos enumerado, y los espacios de nombre declarados por el otro, falta fusionarlos. De eso se encarga el comando `\keys_set:nV`: se le da un espacio de nombre y un *keyval*¹⁵, y por cada `clave=valor` del segundo busca en el primero una clave que se llame igual y ejecuta su instrucción con el valor correspondiente.

Ahí es donde por fin ocurre algo. La clave `font` del usuario encuentra la clave `font` del namespace `{ pyst }`, se ejecuta la instrucción que se le había asignado, y `\g_pyst_font_tl` pasa a contener `minion`. El mismo mecanismo, repetido para todas las claves de todas las fuentes.

Conviene detenerse en por qué declarar y aplicar son dos operaciones separadas, porque de esa separación depende todo lo demás.

Los espacios de nombres se declaran *una sola vez*. La instrucción asociada a `font` —normalizar el valor recibido y depositarlo en `\g_pyst_font_tl`— se escribe en un único lugar y no

¹⁴En el sentido de que le *asigne un valor*, en este caso, como opción en `\documentclass[...]`.

¹⁵Volva recordar, una cadena de `clave=valor`.

vuelve a escribirse. El volcado, en cambio, se hace *cuatro veces*: una por fuente, en el orden del invariante.

Imagínese la alternativa. Que cada fuente trajera consigo su propia interpretación de las claves: que `\DeclareDocumentType` supiera por su cuenta qué hacer con un `font=`, y que el procesamiento de `\documentclass` lo supiera por la suya, y el del departamento otra vez. Serían cuatro copias del mismo conocimiento. El día que `font` cambiara de comportamiento habría que acertar a corregir las cuatro, y la que se olvidara seguiría funcionando en silencio con la regla antigua. Este proyecto ya pagó una vez ese precio, y de ahí viene la norma de no mantener dos fuentes de verdad para lo mismo.

Con la declaración única eso es imposible: solo existe una interpretación de `font`, de modo que `font=minion` significa exactamente lo mismo lo escriba el catálogo de tipos, el de departamentos o el usuario.

Lo que sí depende de quién lo escriba no es el significado, sino el valor que queda al final. Como las cuatro aplicaciones son sucesivas y todas terminan escribiendo sobre la misma variable, la última en ejecutarse es la que deja su valor puesto. El usuario gana al tipo no porque su `font` signifique otra cosa, sino porque su turno llega después.

Ésa es la división: los espacios de nombres contestan *qué significa* cada clave, y el orden de aplicación contesta *cuál de los valores prevalece*. Son dos preguntas independientes, y tenerlas en piezas distintas del sistema es lo que permite responder a cada una sin tocar la otra.

7.6 Un tercer espacio de nombre que no es un espacio de nombre

La imagen del pronombre y el nombre describe bien `{ pyst }` y `{ pyst / dept }`, pero hay un tercer namespace, `{ pyst / prescan }`, al que no se aplica y conviene decir por qué.

Los otros dos son espacios de nombre. Definen qué significa cada clave del sistema. `{ pyst / prescan }` no realiza esa definición. Tiene dos claves, `doc-type` y `dept-name`, cuya única instrucción es copiar el valor recibido en una variable local, y un operador `unknown` que descarta en silencio todo lo demás. No es en realidad un espacio de nombre; es más bien un *cedazo* que sirve para mirar dentro de la lista del usuario y sacar dos datos sin procesar ni descartar nada más.

Este espacio de nombre, declarado incluso en un fichero diferente en el que se declaran los espacios de nombre `pyst` y `pyst/dept` existe porque hay un problema de huevo y gallina. Para aplicar el tipo de documento que declara el usuario hay que saber cuál es, y eso lo dice una clave que está dentro de la misma lista que aún no se puede procesar. El cedazo resuelve la paradoja usando la misma maquinaria con otro propósito: la de `\keys_set` sabe recorrer un *keyval* y emparejar nombres, y eso es justo lo que hace falta para pescar dos valores e ignorar el resto. Que la herramienta sea la misma no debe hacer pensar que la intención con la que se usa lo sea.

7.7 Y lo que queda después

Al terminar las tres etapas, las variables internas del sistema contienen los valores correctos, resueltos según la precedencia. Lo que resta es de otra naturaleza: recoger lo que PYST-PUB no reconoce y pasárselo intacto a KOMA-Script (bloque 7), resolver el conflicto entre las dos estrategias de *layout* (bloque 8), y proyectar el resultado a la forma que `\LoadClass` y `\KOMAOptions` esperan (`pyst-build.ltx`).

Con ese arco a la vista, la tabla que sigue puede leerse como un detalle de cuándo ocurre exactamente cada cosa.

Tabla 5: La fase PRE, bloque a bloque.

Bloque	Qué ocurre
<code>pyst- types.cfg</code>	<code>\DeclareDefaultTypeOptions</code> y <code>\DeclareDocumentType</code> guardan <i>keyval</i> en bruto en una lista de propiedades. Sin efecto todavía.
<code>pyst- depts.cfg</code>	<code>\DeclareDepartment</code> hace lo mismo, en una lista de propiedades similar, leída en régimen $\LaTeX$ 2 ϵ . Sin efecto todavía.
0	Autocomprobación: cada clave reservada se aplica en seco para confirmar que su guardián existe. Va la primera, antes de que ningún bloque del motor haya llegado a ejecutarse.
1	Se definen las funciones de validación que usarán los bloques 4 y 5. No aplica nada.
2	Se leen los dos <code>.cfg</code> y se construyen las secuencias de tipos y departamentos disponibles.
3	Pre-escaneo de <code>doc-type</code> y <code>dept-name</code> sobre la lista cruda, y resolución de ambos nombres contra sus catálogos. Aborta si falta el tipo o no consta; avisa si el departamento no consta.
4	Si se pidió un departamento, se aplica su <i>keyval</i> contra <code>{ pyst / dept }</code> . Un nombre no reconocido avisa, no aborta.
5	Se aplican los defaults globales y, sobre ellos, el tipo seleccionado, ambos contra <code>{ pyst }</code> . Primer efecto real sobre la configuración del documento.
6	Se aplica la lista cruda completa del usuario, sobre los dos namespaces. Gana sobre todo lo anterior por ser lo último en escribir; las claves reservadas se rechazan aquí, con aviso.
7	Se filtra una copia de la lista cruda: lo que PYST-PUB no reconoce —salvo el contenido ya enrutado de las claves contenedoras— pasa intacto al canal del usuario.
8	Si hay <code>geometry-layout</code> , gana sobre <code>koma-layout</code> , con aviso.
<code>pyst- build.ltx</code>	Funde las listas de opciones (tipo, después usuario) y las proyecta a <code>\PystClassOptions</code> / <code>\PystBaseClass</code> .
<code>pyst-pub.cls</code>	<code>\LoadClass</code> , luego <code>\KOMAOptions</code> , luego <code>\recalctypearea</code> . La clase base queda cargada con las opciones correctas.

Una advertencia final sobre esta tabla, para quien la use como mapa del fichero: los bloques 4 y 5 aparecen en `pyst-options.ltx` en ese mismo orden, es decir, el departamento antes que los defaults y el tipo. El invariante de precedencia que se enunció al principio no queda alterado por ello, porque departamento y tipo escriben en espacios de nombres diferentes.

Parte II

Aspectos técnicos en profundidad

8 Análisis del comando `\pyst_coverstyle_normalize:n`

```

\cs_new_protected:Npn \pyst_coverstyle_normalize:n #1
{
  \tl_set:Nn \l_tmpa_tl { #1 }
  \tl_trim_spaces:N \l_tmpa_tl
  \str_set:Nx \l_tmpa_str { \tl_to_str:N \l_tmpa_tl }
  \str_if_eq:VnTF \l_tmpa_str { none }
  { \tl_gclear:N \g_pyst_coverstyle_file_tl }
  {
    \str_if_empty:NTF \l_tmpa_str
    { \tl_gclear:N \g_pyst_coverstyle_file_tl }
    {
      \str_if_in:NnTF \l_tmpa_str { / }
      { \msg_warning:nnnnnn { pyst } { invalid-choice }
        { coverstyle } { \l_tmpa_str } { <n>|<n>.def|none } {
none } }
      {
        \exp_args:NNV \str_if_in:NnTF \l_tmpa_str \c_backslash_str
        { \msg_warning:nnnnnn { pyst } { invalid-choice }
          { coverstyle } { \l_tmpa_str } { <n>|<n>.def|none } {
none } }
        {
          \exp_args:NnV \regex_match:nnTF { \.def \Z }
\l_tmpa_str
          { \tl_gset:Nx \g_pyst_coverstyle_file_tl {
\l_tmpa_str } }
          { \tl_gset:Nx \g_pyst_coverstyle_file_tl {
\l_tmpa_str .def } }
        }
      }
    }
  }
}

```

8.1 Firma del Comando y Normas de Nomenclatura (expl3)

`\cs_new_protected:Npn`: Define una nueva función **protegida** (no se expandirá de forma prematura en contextos de expansión total como `\tl_gset:Nx` o `\write`).

- `:Npn`: Especificador de tipo que indica la estructura del definidor:
 - `N`: Toma una única secuencia de control como nombre del comando (`\pyst_coverstyle_normalize:n`).
 - `p`: Define la firma de parámetros tradicionales de T_EX (`#1`).

- `n`: El cuerpo de la función delimitado entre llaves `{ ... }`.

`\pyst_coverstyle_normalize:n`: Estructura del identificador:

- `pyst`: Espacio de nombres (*namespace*) del paquete.
- `coverstyle_normalize`: Nombre descriptivo de la función.
- `:n`: Indica que la función acepta **un argumento estándar** delimitado entre llaves `{...}`.

8.2 Análisis Detallado Línea por Línea

8.2.1 Paso 1: Limpieza e Higienización Inicial

```
\tl_set:Nn \l_tmpa_tl { #1 }
\tl_trim_spaces:N \l_tmpa_tl
\str_set:Nx \l_tmpa_str { \tl_to_str:N \l_tmpa_tl }
```

1. `\tl_set:Nn \l_tmpa_tl { #1 }` (Firma `:Nn` → Variable `N`, Valor `n`): Asigna el argumento `#1` introducido por el usuario a la lista de tokens local temporal `\l_tmpa_tl`.
2. `\tl_trim_spaces:N \l_tmpa_tl` (Firma `:N` → Variable `N`): Elimina los espacios en blanco iniciales y finales de la variable `\l_tmpa_tl`.
3. `\str_set:Nx \l_tmpa_str { \tl_to_str:N \l_tmpa_tl }` (Firma `:Nx` → Variable `N`, Valor totalmente expandido `x`): Convierte el contenido de `\l_tmpa_tl` en una cadena de caracteres puros (categoría 12) mediante `\tl_to_str:N` y guarda el resultado en la variable de cadena local `\l_tmpa_str`.

8.2.2 Paso 2: Evaluación de Casos Especiales ("none" o Vacío)

```
\str_if_eq:VnTF \l_tmpa_str { none }
{ \tl_gclear:N \g_pyst_coverstyle_file_tl }
{
  \str_if_empty:N \l_tmpa_str
  { \tl_gclear:N \g_pyst_coverstyle_file_tl }
  { ... }
}
```

1. `\str_if_eq:VnTF \l_tmpa_str { none }` (Firma `:VnTF` → Valor de Variable `V`, Literal `n`, Rama Verdadero `T`, Rama Falso `F`): Compara si el valor de `\l_tmpa_str` equivale exactamente a la cadena "none". Si se cumple, ejecuta `\tl_gclear:N \g_pyst_coverstyle_file_tl` (vacía globalmente la variable del archivo de portada) desactivando cualquier portada.
2. `\str_if_empty:N \l_tmpa_str` (Firma `:NTF` → Variable `N`, Rama Verdadero `T`, Rama Falso `F`): Si no es "none", comprueba si la cadena está vacía (por ejemplo, si el usuario pasó `coverstyle={}` o `coverstyle={ }`). Si está vacía, también vacía globalmente `\g_pyst_coverstyle_file_tl`.

8.2.3 Paso 3: Validación de Seguridad y Filtrado de Caracteres Prohibidos

```
\str_if_in:NnTF \l_tmpa_str { / }
{ \msg_warning:nnnnnn { pyst } { invalid-choice }
  { coverstyle } { \l_tmpa_str } { <n>|<n>.def|none } { none } }
{
  \exp_args:NNV \str_if_in:NnTF \l_tmpa_str \c_backslash_str
  { \msg_warning:nnnnnn { pyst } { invalid-choice }
    { coverstyle } { \l_tmpa_str } { <n>|<n>.def|none } { none } }
  { ... }
}
```

1. `\str_if_in:NnTF \l_tmpa_str { / }` (Firma :NnTF → Variable **N**, Subcadena **n**, **T**, **F**): Verifica si la entrada contiene una barra diagonal /. Si la contiene, emite una advertencia mediante `\msg_warning:nnnnnn` pasando el módulo (`pyst`), el nombre del mensaje (`invalid-choice`), y los 4 parámetros del mensaje (clave, valor recibido, opciones válidas y valor asumido por defecto).
2. `\exp_args:NNV \str_if_in:NnTF \l_tmpa_str \c_backslash_str`: Comprueba si la cadena contiene una barra invertida (\).
 - `\exp_args:NNV`: Modifica la expansión de los argumentos para el comando subsecuente (`\str_if_in:NnTF`):
 - 1.^{er} arg (**N**): `\str_if_in:NnTF` se mantiene sin cambios.
 - 2.^o arg (**N**): `\l_tmpa_str` se mantiene como variable.
 - 3.^{er} arg (**V**): `\c_backslash_str` se expande a su **valor** (la constante del sistema que representa el carácter \).

8.2.4 Paso 4: Normalización de la Extensión (.def)

```
\exp_args:NnV \regex_match:nnTF { \.def \Z } \l_tmpa_str
{ \tl_gset:Nx \g_pyst_coverstyle_file_tl { \l_tmpa_str } }
{ \tl_gset:Nx \g_pyst_coverstyle_file_tl { \l_tmpa_str .def } }
```

1. `\exp_args:NnV \regex_match:nnTF { \.def \Z } \l_tmpa_str`: Evalúa si el nombre introducido por el usuario ya termina con la extensión `.def`.
 - `\exp_args:NnV`: Modifica la orden de evaluación de argumentos para `\regex_match:nnTF`:
 - 1.^{er} arg (**N**): `\regex_match:nnTF`.
 - 2.^o arg (**n**): Expresión regular `{ \.def \Z }` (donde `\.def` busca la subcadena literal y `\Z` fuerza que sea el final de la cadena).
 - 3.^{er} arg (**V**): Expande el **valor** contenido en la variable `\l_tmpa_str`.
2. **Asignación Final:**
 - **Si coincide (Rama T):** `\tl_gset:Nx \g_pyst_coverstyle_file_tl { \l_tmpa_str }` (asigna directamente el valor porque ya incluye `.def`).

- **Si no coincide (Rama F):** `\tl_gset:Nx \g_pyst_coverstyle_file_tl { \l_tmpa_str .def }` (concatena `.def` al final del nombre y lo almacena).

En la [Figura 1](#) se representa el diagrama del flujo lógico de ejecución y en la [Tabla 6](#) resumimos las funciones y variables del núcleo `expl3` empleadas en la función.

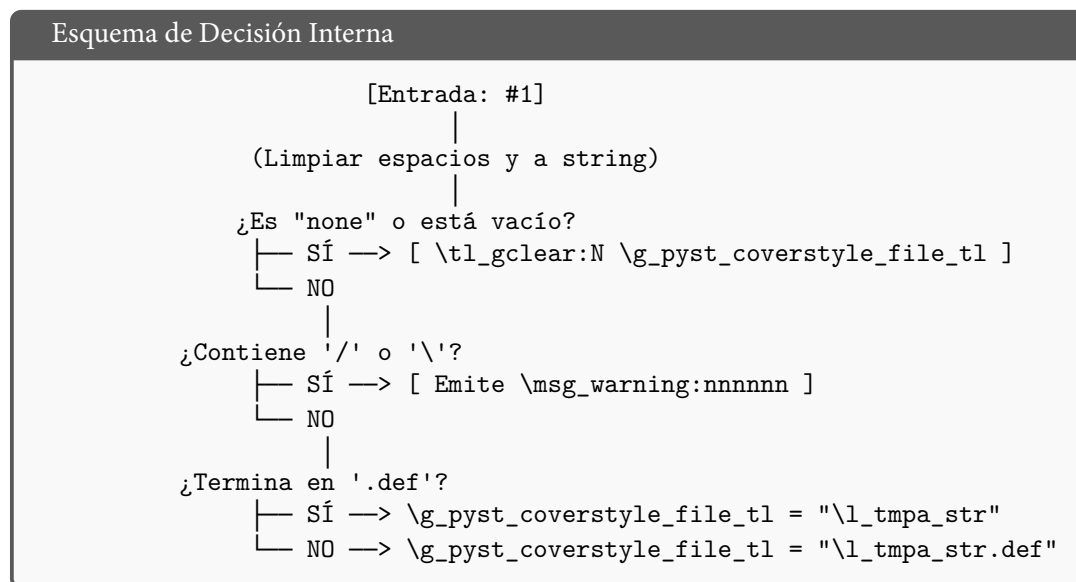


Figura 1

Tabla 6: Funciones y variables del núcleo `expl3` empleadas en la función..

Comando / Variable	Tipo / Firma	Propósito en el Código
<code>\cs_new_protected:Npn</code>	Definidor	Declara la macro protegida evitando expansiones no deseadas.
<code>\tl_set:Nn / \tl_gset:Nx</code>	Token List	Asigna el valor localmente (<code>:Nn</code>) o globalmente (<code>:Nx</code>).
<code>\tl_trim_spaces:N</code>	Manipulación	Elimina espacios al inicio y final de la variable.
<code>\tl_to_str:N</code>	Conversión	Convierte tokens en cadena pura (catcode 12).
<code>\str_if_eq:VnTF</code>	Lógica	Compara el valor de una variable con un literal (<code>none</code>).
<code>\str_if_empty:NTF</code>	Lógica	Evalúa si la variable de cadena no contiene caracteres.
<code>\str_if_in:NnTF</code>	Lógica	Busca la existencia de una subcadena (<code>/</code> o <code>\</code>).
<code>\exp_args:NNV / :NnV</code>	Expansión	Intercambia el orden de evaluación pasando el valor (<code>V</code>).
<code>\c_backslash_str</code>	Constante	Representa el carácter <code>\</code> como cadena.
<code>\regex_match:nnTF</code>	Regex	Comprueba si el final (<code>\Z</code>) coincide con <code>.def</code> .
<code>\msg_warning:nnnnnn</code>	Mensajes	Lanza una advertencia formal con 4 argumentos.

9 Análisis de la Función del núcleo de expl3 \keys_set:nn

El comando `\keys_set:nn` es la función núcleo del módulo `l3keys` en expl3 (el entorno de programación de $\text{\LaTeX}$ 3), encargada de interpretar, validar y ejecutar una lista de pares clave-valor (`[clave = valor]`) asociada a un espacio de nombres (*namespace*) o módulo específico.

9.1 Sintaxis y Firma de la Función

```
\keys_set:nn { <módulo> } { <lista de claves> }
```

- **Argumento 1 (#1 – <módulo>):** Define el *namespace* o la ruta raíz donde están registradas las claves (por ejemplo, `pyst` o `pyst / dept`).
- **Argumento 2 (#2 – <lista de claves>):** Cadena de texto separada por comas con las asignaciones que se desean procesar (por ejemplo, `color = red, draft, level = 2`).

9.2 Ciclo de Vida y Flujo Interno de Procesamiento

Cuando `\keys_set:nn` recibe la lista de pares clave-valor, ejecuta secuencialmente los siguientes pasos por cada elemento:

Paso 1: Normalización y Limpieza de Espacios

- Separa la lista por comas individuales.
- Elimina los espacios en blanco iniciales y finales (*trimming*) tanto del nombre de la clave como del valor proporcionado, a menos que el valor esté explícitamente protegido entre llaves (`{ ... }`).
- Separa el nombre de la clave del valor mediante el primer carácter `=` encontrado. Si no hay `=`, el valor queda indefinido inicialmente y la clave pasa a evaluarse como una clave de presencia/*flag* o a buscar un valor por defecto.

Paso 2: Resolución de Rutas (Namespaces)

- Concatena el argumento de módulo (#1) con el nombre de la clave.
- Si el módulo es `pyst` y la clave es `font`, la ruta resultante en memoria es `pyst / font`.
- Si la clave incluye una subruta (ejemplo: `geometry / top = 2cm` llamada con el módulo `pyst`), la ruta final resuelta es `pyst / geometry / top`.

Paso 3: Verificación de Reglas de Valor

Antes de ejecutar el código de la clave, `\keys_set:nn` valida las restricciones definidas previamente en `\keys_define:nn`:

- **Sin valor proporcionado (clave):** Si la clave se pasó sin `=`, busca si tiene un valor por defecto declarado mediante `.default:n`. Si existe, asigna ese valor a #1. Si no tiene por

defecto y requiere un valor (`.value_required:n`), detiene la ejecución lanzando el error `kernel/value-required`.

- **Con valor proporcionado** (`clave = valor`): Si la clave tiene prohibido recibir valores (`.value_forbidden:n`), detiene la ejecución y lanza el error `kernel/value-forbidden`.

Paso 4: Ejecución de la Acción (`.code:n`)

Busca el código de la clave registrada en el motor `l3keys`:

- Pasa el valor procesado como argumento `#1` al cuerpo del código `.code:n` (o a las macros internas de asignación como `.tl_set:N`, `.bool_set:N`, `.choice:`, etc.).
- Ejecuta las instrucciones asociadas en el contexto actual.

Paso 5: Gestión de Claves Desconocidas

Si la clave no existe dentro del módulo:

- Comprueba si el módulo tiene definida una clave especial `unknown` (`unknown .code:n`). Si existe, le pasa el nombre de la clave y el valor recibido para que el usuario procese la excepción.
- Si no existe la clave `unknown`, `\keys_set:nn` interrumpe el procesamiento y lanza un error crítico de compilación (`kernel/unknown-key`).

9.3 Variantes de Expansión

Para adaptar la entrada a la procedencia de los datos, el núcleo de `expl3` proporciona múltiples variantes de expansión para `\keys_set:nn`, descritos en la [Tabla 7](#).

Tabla 7: Variantes de Expansión del comando `\keys_set:nn`.

Variante	Firma	Descripción	Uso típico
<code>\keys_set:nV</code>	<code>{ <módulo> } \variable_tl</code>	Expande el contenido de una variable <code>tl</code> o <code>clist</code> y pasa su valor.	Procesar opciones leídas de <code>\@raw@classoptionslist</code> o de un <i>token list</i> almacenado.
<code>\keys_set:ne/nx</code>	<code>{ <módulo> } { <expresión> }</code>	Evalúa exhaustivamente la lista mediante expansión total (<i>full expansion</i>) antes de procesar.	Pasar claves cuyos valores son construidos mediante macros dinámicas o cálculos.
<code>\keys_set:no</code>	<code>{ <módulo> } { <macro> }</code>	Expande el argumento 2 exactamente una vez (<i>once expansion</i>).	Compatibilidad con listas generadas por comandos tradicionales de $\TeX$.
<code>\keys_set:nv</code>	<code>{ <módulo> } { <nombre_var> }</code>	Toma el nombre de la variable como cadena de texto, obtiene su contenido y lo procesa.	Cuando el nombre del contenedor de las opciones se genera dinámicamente.

9.4 Variaciones de Tolerancia y Filtrado

Además de la función estándar `\keys_set:nn`, el módulo `l3keys` ofrece comandos derivados para gestionar claves no reconocidas o aplicar filtros sin detener la compilación con errores:

9.4.1 Ignorar Claves Desconocidas (`\keys_set_known:nn`)

```
\keys_set_known:nn { <módulo> } { <lista de claves> }
```

Procesa únicamente las claves reconocidas por `<módulo>`. Cualquier clave que no exista en ese *namespace* se ignora silenciosamente sin lanzar errores ni requerir una clave `unknown`.

9.4.2 Capturar Claves Desconocidas (`\keys_set_known:nnN`)

```
\keys_set_known:nnN { <módulo> } { <lista de claves> } \l_resto_tl
```

Ejecuta las claves reconocidas y almacena en la variable *token list* `\l_resto_tl` todas las claves no procesadas (manteniendo su formato `clave = valor`). Es el método estándar para delegar opciones a otras clases o paquetes.

9.4.3 Filtrado por Grupos (`\keys_set_filter:nnn`)

```
\keys_set_filter:nnn { <módulo> } { <grupos> } { <lista de claves> }
```

Procesa solo aquellas claves que pertenezcan a uno de los grupos especificados (asignados previamente con la propiedad `.groups:n = { <grupo> }`).

10 El vocabulario de expl3, reunido

A lo largo de la nota han ido apareciendo unas setenta funciones de `expl3`. Conviene cerrar reuniendo el vocabulario, porque una vez visto en conjunto se advierte que no son setenta cosas distintas que memorizar, sino un puñado muy pequeño de piezas que se combinan.

10.1 Cómo se lee un nombre

Todo nombre de `expl3` tiene la misma estructura:

```
\módulo_acción:firma
```

El *módulo* dice sobre qué tipo de dato opera; la *acción*, qué le hace; y la *firma*, cómo hay que pasarle los argumentos. Con esas tres piezas, un nombre que nunca se haya visto resulta legible.

Módulo	Qué gestiona
<code>tl</code>	<i>Token lists</i> : cadenas de texto con nombre. El tipo más general.
<code>str</code>	Cadenas de caracteres, sin catcodes. La comparación robusta de palabras.
<code>clist</code>	Listas separadas por comas. El formato que KOMA espera.
<code>seq</code>	Secuencias: listas ordenadas de elementos independientes.
<code>prop</code>	Lista de propiedades: clave → valor. El catálogo de tipos y departamentos.
<code>bool</code>	Interruptores: cierto o falso.
<code>keys</code>	El sistema de claves: declarar, aplicar y consultar espacios de opciones.
<code>keyval</code>	La maquinaria de bajo nivel sobre la que <code>keys</code> está construido.
<code>msg</code>	Mensajes al usuario: información, avisos, errores y abortos.
<code>cs</code>	<i>Control sequences</i> : definir funciones y comprobar si existen.
<code>file</code>	Ficheros: comprobar existencia y leerlos.

10.2 Los prefijos de las variables

<code>g_</code>	Global: sobrevive al grupo, legible desde cualquier punto posterior.
<code>l_</code>	Local: su valor no se garantiza fuera del bloque que la usa.
<code>c_</code>	Constante: se fija una vez y no debe cambiar.

La correspondencia con la acción es estricta: una variable `g_` se asigna con `gset` o `gput`; una `l_`, con `set` o `put`. Respetarla permite saber, con solo leer un nombre, cuánto vive un dato.

10.3 Las acciones que se repiten

<code>new</code>	Crear la variable. Toda variable debe declararse antes de usarse: <code>\bool_new:N, \tl_new:N.</code>
<code>set / gset</code>	Asignar un valor, sustituyendo el anterior: <code>\clist_gset:Nn.</code>
<code>put_right</code>	Añadir al final, conservando lo que hubiera: <code>\seq_gput_right:Nn.</code>
<code>clear</code>	Vaciar: <code>\seq_gclear:N.</code>
<code>get</code>	Extraer un valor a una variable de destino: <code>\prop_get:NvN.</code>
<code>use</code>	Entregar el contenido allí donde se escriba: <code>\tl_use:N, \clist_use:Nn.</code>
<code>if_...</code>	Preguntar: <code>\tl_if_empty:NT, \clist_if_in:NnTF, \file_if_exist:nTF, \cs_if_exist:NTF.</code>
<code>map_inline</code>	Recorrer una colección elemento a elemento: <code>\clist_map_inline:nn, \prop_map_inline:Nn.</code>
<code>_eq</code>	Copiar una variable en otra: <code>\tl_gset_eq:NN.</code>

A esa lista se añaden unas pocas operaciones específicas de un módulo que también han aparecido: `\seq_gset_split:Nnn`, que convierte una cadena en secuencia troceándola por un separador; `\seq_item:Nn`, que recupera un elemento por su posición — contando desde uno, ; `\tl_trim_spaces:N`, que recorta espacios sobrantes; `\str_lowercase:n`, que pasa una cadena a minúsculas; y `\str_case:nnF`, que compara contra una lista de casos.

10.4 Los sufijos de la firma

N	Un nombre de variable o función, sin llaves.
n	Un argumento normal, entre llaves, que se pasa literal.
V	El <i>valor</i> de la variable que se indique.
o	Una sola expansión del argumento.
e, x	Expansión completa, hasta el final.
p	Texto de parámetros, al definir una función (Npn).
c	Un nombre de función construido a partir de texto.

T	Añade la rama que se ejecuta si la respuesta es cierta.
F	Añade la rama que se ejecuta si es falsa.
TF	Añade ambas.
_p	<i>Predicado</i> : en lugar de ejecutar ramas, devuelve un booleano combinable.
	No todas las funciones lo tienen.

10.5 Dos idiomas que conviene reconocer

Cierran el repaso dos construcciones que no son funciones sino *maneras de usarlas*, y que aparecen repetidamente en el código de PYST-PUB.

La primera es `\exp_args:N...`, que no aporta lógica propia: prepara los argumentos de otra función antes de entregárselos, permitiendo construir sobre la marcha la variante de firma que no existe en la biblioteca. Cuando se lea `\exp_args:NnnV`, basta con leer la firma letra a letra como instrucciones sobre los argumentos que siguen.

La segunda es el uso de funciones propias del sistema, `\pyst_set_bool:Nnn`, `\pyst_guard_reserved:nn`, `\pyst_coverstyle_normalize:n`, `\pyst_set_language:n`, `\pyst_dept_input_twice:n`, los tres gestores de ruta y el trío de validación, que siguen exactamente la misma convención de nomenclatura que las de la biblioteca. No son un lenguaje aparte: `\pyst_koma_route:n` se lee igual que `\tl_gset:Nn`, y su firma anuncia igualmente cómo hay que llamarla. Esa continuidad es deliberada: quien sepa leer `expl3` sabe leer PYST-PUB sin aprender nada nuevo.