

Traduction et adaptation française du manuel de pgfkeys*

Extrait de *The TikZ and PGF Packages*

Manual for version 3.1.4a

<https://github.com/pgf-tikz/pgf>

Till Tantau[†]

Institut für Theoretische Informatik
Universität zu Lübeck

4 août 2019

Table des matières

I	Utilitaires	2
1	Gestion des clés	3
1.1	Introduction	3
1.2	Comparaison avec d'autres extensions	3
1.3	Mode d'emploi rapide du système de clés	3
2	L'arborescence des clés	4
3	Réglage des clés	6
3.1	Détection du premier caractère	7
3.2	Arguments par défaut	8
3.3	Clés qui exécutent des commandes	9
3.4	Clés qui entreposent des valeurs	10
3.5	Clés manipulées	10
3.6	Clés inconnues	12
3.7	Chemins de recherche et clés manipulées	12
4	Manipulateurs	13
4.1	Manipulateurs de gestion du chemin	13
4.2	Valeurs par défaut	13
4.3	Définir le code d'une clé	14
4.4	Définir des styles	16
4.5	Clés de choix, booléennes, d'entreposage	17
4.6	Valeurs développées, valeurs multiples	19
4.7	Manipulateurs de transmission	19
4.8	Manipulateurs d'essais	21
4.9	Manipulateurs d'inspection	22
5	Clés d'erreur	22

*Le T_EXnicien de surface, version 0.1, 4 août 2019.

[†]Directeur de cette documentation. Certaines parties de cette documentation ont été écrites par d'autres auteurs comme indiqué dans ces parties ou chapitres.

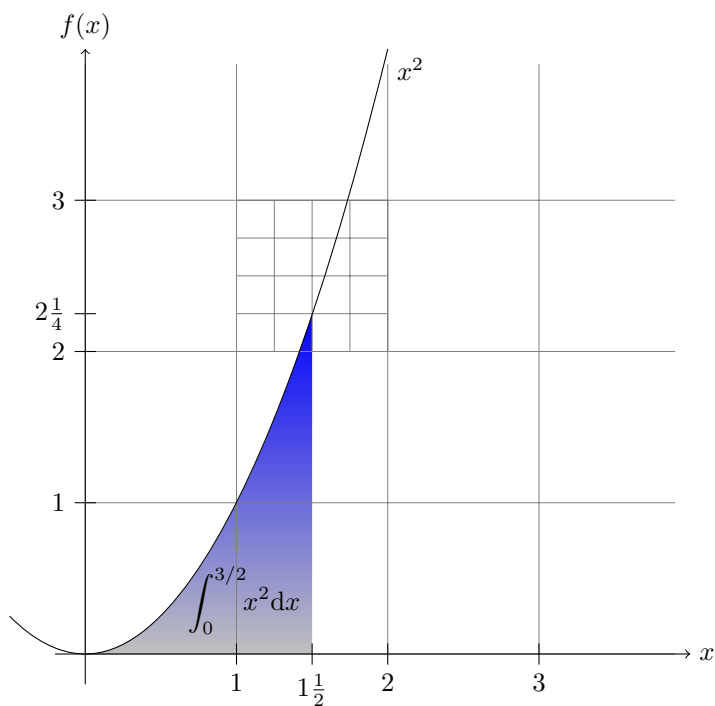
6	Filtrage de clés	22
6.1	Exemple préliminaire	23
6.2	Définition des filtres	23
6.3	Manipulateurs de clés non-traitées	25
6.4	Gestion des familles	25
6.5	Autres filtres de clés	27
6.6	Interface de programmation	28
6.7	Définir ses propres filtres et manipulateurs de filtres	28
	Index	30

Première partie

Utilitaires

par Till TANTAU

Ces extensions « utilitaires » ne sont pas directement impliquées dans la création des graphiques mais on peut néanmoins les trouver utiles. Elles sont toutes soit directement dépendantes de PGF soit conçues pour fonctionner correctement avec PGF même si on peut les utiliser seules.



```
\begin{tikzpicture}[scale=2]
  \shade[top color=blue,bottom color=gray!50] (0,0) parabola (1.5,2.25) |- (0,0);
  \draw (1.05cm,2pt) node[above] {$\displaystyle\int_0^{3/2} \! \! \! \! x^2 \mathrm{d}x$};

  \draw[help lines] (0,0) grid (3.9,3.9)
    [step=0.25cm] (1,2) grid +(1,1);

  \draw[->] (-0.2,0) -- (4,0) node[right] {$x$};
  \draw[->] (0,-0.2) -- (0,4) node[above] {$f(x)$};

  \foreach \x/\xtext in {1/1, 1.5/1\frac{1}{2}, 2/2, 3/3}
    \draw[shift={(\x,0)}] (0pt,2pt) -- (0pt,-2pt) node[below] {$\xtext$};

  \foreach \y/\ytext in {1/1, 2/2, 2.25/2\frac{1}{4}, 3/3}
    \draw[shift={f(0,\y)}] (2pt,0pt) -- (-2pt,0pt) node[left] {$\ytext$};

  \draw (-.5,.25) parabola bend (0,0) (2,4) node[below right] {$x^2$};
\end{tikzpicture}
```

1 Gestion des clés

Cette section décrit l’extension `pgfkeys`. Cette extension est chargée automatiquement autant par PGF que par TikZ.

```
\usepackage{pgfkeys} %  $\TeX$ 
\input pgfkeys.tex % plain  $\TeX$ 
\usemodule[pgfkeys] % Con $\TeX$ t
```

On peut se servir de cette extension indépendamment de PGF. Notez qu’aucune extension faisant partie de PGF ne doit être chargée — de fait ni la couche de simulation ni la couche de système ne sont requises. L’abréviation pour Con \TeX t est `pgfkey` si on ne charge pas `pgfmod`.

1.1 Introduction

1.2 Comparaison avec d’autres extensions

L’extension `pgfkeys` définit un système de gestion de clés-valeurs qui ressemble, en quelque sorte, à l’extension `keyval`, qui est plus légère, et à son amélioration `xkeyval`. Toutefois, `pgfkeys` met en œuvre une philosophie légèrement différente de celle de ces systèmes et coexistera pacifiquement avec elles deux.

Les différences principales entre `pgfkeys` et `xkeyval` sont les suivantes :

- `pgfkeys` organise les clés dans une **arborescence** (*tree*), alors que `keyval` et `xkeyval` utilisent des familles. Dans `pgfkeys` les familles correspondent aux entrées à la racine de l’arborescence des clés ;
- `pgfkeys` n’a pas d’impact sur la **pile de sauvegarde** (*save-stack*) — il faudra lire le \TeX livre très attentivement pour en comprendre l’importance — ;
- `pgfkeys` est légèrement plus lent que `keyval` mais pas de beaucoup ;
- `pgfkeys` gère les styles c.-à-d. qu’une clé peut simplement représenter plusieurs clés — qui peuvent, à leur tour, représenter d’autres clés ou simplement exécuter du code. TikZ utilise énormément ce mécanisme ;
- `pgfkeys` permet d’avoir des clés dont le code utilise plusieurs arguments. On peut, toutefois, simuler ce comportement avec `keyval` ;
- `pgfkeys` gère les **manipulateurs** (*handler*). Ce sont des **fonctions de rappel** (*call-backs*), utilisées quand une clé est inconnue. Ces manipulateurs sont très souples et, de fait, même les différentes définitions des clés sont réalisées à l’aide de manipulateurs.

1.3 Mode d’emploi rapide du système de clés

Le présent mode d’emploi du mécanisme de clés de PGF ne traite que les fonctionnalités les plus courantes. On consultera le reste de ce document pour une présentation complète de ce qui se passe.

Les clés sont rangées dans une grande arborescence qui rappelle celle des fichiers d’un système Unix. Une clé typique peut être, par exemple, `/tikz/coordinate system/x` ou simplement `/x`. Une fois encore comme dans Unix, on peut, quand on spécifie une clé, donner le chemin complet mais on utilisera habituellement uniquement le nom de la clé — ce qui correspond au nom de fichier sans chemin —, le chemin sera ajouté automatiquement.

Généralement, — mais pas obligatoirement —, on associera du code à une clé. Pour exécuter ce code, on utilisera la commande `\pgfkeys`. Cette commande prend en argument une liste de « paires de clé-valeur ». Chaque paire est de la forme `\pgfkeys{<clef>=<valeur>}`. Pour chaque paire la commande `\pgfkeys` exécutera le code attaché à la `<clef>` en lui passant la `<valeur>`.

Voici un exemple générique d’utilisation de la commande `\pgfkeys` :

```
\pgfkeys{/ma clef = salut,
/vos clefs/clef principale = untruc\bizarre,
nom de clef sans chemin = autre chose}
```

À ce stade on n’a pas besoin d’apprendre une nouvelle commande pour attacher du code à une clé puisque c’est encore `\pgfkeys` que l’on peut utiliser pour ça. On le fait en utilisant ce qu’on appelle des manipulateurs. Ils ressemblent à des clés dont le nom ressemble à celui d’un « fichier caché dans Unix » puisqu’il commence par un point. Le manipulateur permettant d’attacher du code à une clé est appelé, avec à propos, `/.code` et utilisé comme ici :

La valeur est [salut!].

```
\pgfkeys{/ma clef/.code = La valeur est [#1].}
\pgfkeys{/ma clef = salut! }
```

Comme on le voit, dans la première ligne de l'exemple, on définit le code pour la clé `/ma clef`. Dans la deuxième, on exécute le code avec l'argument ayant pour valeur `salut!`.

Il y a beaucoup d'autres manipulateurs pour définir des clés. On peut, par exemple, définir aussi une clé dont la valeur consiste en plus d'un paramètre.

Les valeurs sont `[a1]` et `[a2]`.

```
\pgfkeys{/ma clef/.code 2 args=Les valeurs sont [#1] et [#2].}
\pgfkeys{/ma clef={a1}{a2}}
```

On veut souvent des clés dont le code est appelé avec une valeur par défaut si l'utilisateur n'en fournit pas. C'est aussi, sans surprise, à l'aide d'un manipulateur que l'on réalise ça, cette fois il est nommé `/.default`.

```
(hello)(salut!) \pgfkeys{/ma clef/.code={#1}}
\pgfkeys{/ma clef/.default=salut!}
\pgfkeys{/ma clef=hello, /ma clef}
```

À l'inverse, on peut préciser qu'une valeur *doit* être donnée grâce au manipulateur `/.value required`. Enfin, on peut interdire le passage d'une valeur avec `/.value forbidden`.

Toutes les clés d'une extension comme, p. ex., TikZ, commencent avec le chemin `/tikz`. Il est clair que l'on a pas envie d'écrire ce chemin en entier à chaque fois que l'on utilise une clé — on ne veut pas, p. ex., devoir écrire des choses comme `\draw[/tikz/line width =1cm]`. Ce dont nous avons besoin est, en quelque sorte, de « changer le chemin par défaut en un chemin précis ». C'est ce que l'on fait avec le manipulateur `/.cd` — pour *change directory* c.-à-d. « changer de dossier ». Une fois que l'on a utilisé ce manipulateur sur une clé, toutes les clés suivantes, *mais seulement dans l'appel en cours de* `\pgfkeys` seront préfixées par ce chemin, si nécessaire.

Voici un exemple :

```
\pgfkeys{/tikz/.cd,line width=1cm,line cap=round}
```

Cela facilite la définition de commande comme `\tikzset` que l'on pourrait définir comme suit — la véritable définition est un peu plus rapide mais l'effet est le même — :

```
\def\tikzset#1{\pgfkeys{/tikz/.cd,#1}}
```

Quand une clé est manipulée, plutôt que d'exécuter du code, elle peut aussi faire que d'autres clés soient exécutées. De telles clés seront appelées **styles** (*styles*). Un style est, en substance, une liste de clés qui doivent être exécutées chaque fois que le style est exécuté. Voici un exemple :

(a : fou)(b : bar)(a : ouah)

```
\pgfkeys{/a/.code={a: #1}}
\pgfkeys{/b/.code={b: #1}}
\pgfkeys{/mon style/.style={a=fou,b=bar,a=#1}}
\pgfkeys{/mon style=ouah}
```

Comme le montre l'exemple ci-dessus, un style peut être paramétré, exactement comme une clé normale.

Comme exemple typique d'utilisation des styles, supposons que l'on veuille définir la clé `/tikz` de telle sorte qu'elle change le chemin par défaut en `/tikz`, voici comment on s'y prendrait :

```
\pgfkeys{/tikz/.style=/tikz/.cd}
\pgfkeys{/tikz,line width=1cm,draw=red}
```

On notera que quand `\pgfkeys` est exécuté, le chemin par défaut est réglé sur `/`. Cela signifie que le premier `tikz` sera complété en `/tikz`. Comme `/tikz` est un style, il est alors remplacé par `/tikz/.cd` qui change le chemin par défaut en `/tikz`, ce qui fait que `line width` est préfixée par `/tikz` comme voulu.

2 L'arborescence des clés

L'extension `pgfkeys` organise les clés dans une **arborescence des clés** (*key tree*). Cette arborescence sera familière à quiconque a utilisé un système de type Unix : une clé est repérée par un **chemin** (*path*) qui consiste en différentes parties séparées par des barres obliques `/`. Un exemple de clé est `/tikz/line width` ou, simplement, `/tikz` mais ce peut être quelque chose de plus compliqué comme `/tikz/cs/x/.store in`.

Fixons le vocabulaire : la clé `/a/b/c` étant donné, nous appellerons chemin de cette clé la partie initiale jusqu'à la dernière barre oblique — c.-à-d. `/a/b` —, tout ce qui vient après la dernière barre oblique sera appelé **nom** (*name*) de la clé — dans un système de fichier ce serait le nom du fichier.

On voudrait ne pas avoir toujours à préciser entièrement le nom de la clé. De fait, on ne spécifie habituellement qu'une partie de la clé — généralement uniquement son nom — et le **chemin par défaut** (*default path*) est alors placé devant. Ainsi, quand le chemin par défaut est `/tikz` et que l'on fait référence à la clé — partielle — `line width`, la véritable clé est `/tikz/line width`. Il existe une règle simple pour dire si une clé est partielle ou complète : si elle commence par une barre oblique `/`, c'est une clé complète et elle n'est pas modifiée, sinon elle est partielle et elle est complétée avec le chemin par défaut.

On notera que le chemin par défaut n'est pas la même chose que le chemin de recherche. En particulier, le chemin par défaut est un unique chemin. Quand on donne une clé partielle, seul ce chemin par défaut est préfixé ; `pgfkeys` ne cherche pas la clé dans les différentes parties du chemin de recherche. On peut, toutefois, simuler les chemins de recherche mais on a, pour cela, besoin d'un mécanisme beaucoup plus compliqué.

Quand on règle des clés — ce que l'on décrira plus bas —, on peut mélanger les clés partielles et les clés complètes et aussi changer le chemin par défaut. Cela permet d'utiliser temporairement des clés situées dans une autre partie de l'arborescence — ce qui se révèle être une fonctionnalité très utile.

Chaque clé peut conserver des **lexèmes** (*tokens*) et il existe des commandes, décrites ci-dessous, pour définir, obtenir et transformer ces lexèmes entreposés dans la clé. Toutefois on utilisera rarement ces commandes directement, la manière normale d'utiliser des clés passe, plutôt, par la commande `\pgfkeys` ou une autre commande qui y fait appel en interne comme, p. ex., `\tikzset`. On peut donc ignorer ce qui suit et passer directement à la section suivante.

`\pgfkeyssetvalue`{*clé complète*}{*texte*}

entrepose le *texte* dans la *clé complète*. La *clé complète* ne peut pas être partielle aussi on n'ajoute pas de chemin par défaut. *texte* est une chaîne de lexèmes arbitraires qui peut même contenir des choses comme `#` ou des si \TeX iens incomplets¹.

```
Bonjour! \pgfkeyssetvalue{/ma famille/ma clef}{Bonjour!}
          \pgfkeysvalueof{/ma famille/ma clef}
```

Le réglage de la clé est toujours local au groupe \TeX ien courant.

`\pgfkeyslet`{*clé complète*}{*macro*}

réalise un `\let` afin que la *clé complète* pointe vers le contenu de la *macro*.

```
Bonjour! \def\helloworld{Bonjour!}
          \pgfkeyslet{/ma famille/ma clef}{\helloworld}
          \def\helloworld{Hello world!}
          \pgfkeysvalueof{/ma famille/ma clef}
```

On ne doit pas rendre une clé égale à `\relax` car une telle clé serait, ou pas, indistinguable d'un clé indéfinie.

La troisième ligne de l'exemple précédent permet de voir que la clé est définie comme égale au contenu de la macro au moment où l'on fait appel à `\pgfkeyslet`. Un changement postérieur de définition de la macro n'aura aucun effet sur le contenu de la clé.

`\pgfkeysgetvalue`{*clé complète*}{*macro*}

extrait les lexèmes entreposés dans *clé complète* et rend la *macro* égale à cette chaîne de lexèmes. Si la clé n'a pas été réglée, la *macro* sera égale à `\relax`.

```
Salut le monde! \pgfkeyssetvalue{/ma famille/ma clef}{Salut le monde!}
                \pgfkeysgetvalue{/ma famille/ma clef}{\helloworld}
                \helloworld
```

`\pgfkeysvalueof`{*clé complète*}

insère la valeur entreposée dans la *clé complète* à la position courante dans le texte.

```
Salut le monde! \pgfkeyssetvalue{/ma famille/ma clef}{Salut le monde!}
                \pgfkeysvalueof{/ma famille/ma clef}
```

1. NdTdS : On peut donc trouver dans cette chaîne des `\if` sans `\fi` correspondant, p. ex.

`\pgfkeysifdefined{<clé complète>}{<if>}{<else>}`

vérifie si la `<clé complète>` a été définie à l'aide de `\pgfkeyssetvalue` ou `\pgfkeyslet`. Si c'est le cas, le code `<si oui>` est exécuté, sinon c'est le code `<si non>` qui l'est.

Cette commande utilisera la commande `\ifcename` de ε -TeX si elle est disponible pour des raisons d'efficacité. Cela entraîne, toutefois, que `\pgfkeysifdefined` peut se comporter différemment suivant que l'on compile avec TeX ou ε -TeX lorsque l'on a réglé la clé comme égale à `\relax`. C'est pour cette raison qu'il ne faut pas le faire.

```
oui \pgfkeyssetvalue{/ma famille/ma clef}{Salut le monde!}  
\pgfkeysifdefined{/ma famille/ma clef}{oui}{non}
```

3 Réglage des clés

On règle les clés à l'aide d'une commande puissante nommée `\pgfkeys`. Cette commande prend en argument une liste de paires de *clés-valeurs*. Ce sont des paires de la forme `<clef> = <valeur>`. L'idée principale est la suivante : pour chaque paire de la liste une action est accomplie. Cette action est l'une des suivantes :

1. une commande est exécutée dont l'argument est `<valeur>`. Cette commande est entreposée dans une sous-clé spéciale de `<clef>` ;
2. la `<valeur>` est entreposée dans la `<clef>` elle-même ;
3. si le nom de la `<clef>` est un manipulateur connu, c'est lui qui intervient ;
4. si la `<clef>` est complètement inconnue, c'est un des quelques *manipulateurs d'erreurs* qui est appelé.

Par ailleurs, si la `<valeur>` est absente, une valeur par défaut peut éventuellement être utilisée. Avant de plonger dans les détails, jetons un œil sur la commande elle-même.

`\pgfkeys{<liste de paires>}`

La `<liste de paires>` doit être une liste de paires de clés-valeurs séparées par des virgules. Une paire de clé-valeur peut avoir l'une des deux formes `<clef> = <valeur>` ou simplement `<clef>`. Les espaces entourant la `<clef>` ou la `<valeur>` sont retirés. On peut entourer la `<clef>` aussi bien que la `<valeur>` par des accolades qui sont également retirées. Il faut, de fait, très souvent entourer la `<valeur>` par des accolades, à chaque fois qu'elle contient un signe égal ou une virgule.

Les paires clés-valeurs de la liste sont traitées dans leur ordre d'apparition. La suite de cette section décrit ce traitement.

Si la `<clef>` est partielle, elle est préfixée avec le chemin par défaut en cours et c'est cette clé « mise à jour » qui est utilisée ensuite. Le chemin par défaut est simplement la racine / quand la première clé est exécutée² mais il peut être changer par la suite. À la fin de la commande, le chemin par défaut reprend la valeur qu'il avait avant que cette commande soit exécutée.

On peut imbriquer les appels à `\pgfkeys`. Ainsi, on peut appeler cette commande dans le code qui est exécuté pour une clé. Comme le chemin par défaut est rétabli à la fin de l'appel à `\pgfkeys`, le chemin par défaut ne changera pas quand on appellera `\pgfkeys` lors de l'exécution du code attaché à une clé — ce qui exactement ce que l'on voulait.

`\pgfqkeys{<chemin par défaut>}{<liste de paires>}`

Cette commande à le même effet que `\pgfkeys {<chemin par défaut>/cd, <liste de paires>}`, elle est juste marginalement plus rapide. On ne devrait pas faire appel à cette commande dans un code d'utilisateur final mais plutôt dans des commandes comme `\tikzset` ou `\pgfset` qui sont appelées très souvent.

`\pgfkeysalso{<liste de paires>}`

Cette commande a exactement le même effet que `\pgfkeys` sauf que le chemin par défaut n'est modifié ni avant ni après que les clés ont été réglées. Cette commande est destinée essentiellement à être appelé par le code attaché à une clé.

2. NdTdS : Il s'agit de la première clé dans un appel de `\pgfkeys` au niveau du document. La suite du texte montre que ce n'est pas toujours le cas quand les appels sont imbriqués.

`\pgfqkeysalso{<chemin par défaut>}{<liste de paires>}`

Cette commande a le même effet que `\pgfkeysalso{<chemin par défaut>/<cd>,<liste de paires>}`, elle est juste plus rapide. Changer le chemin par défaut à l'intérieur d'une `\pgfkeysalso` est dangereux, on l'utilisera donc avec précaution. Un endroit assez sûr pour un appel à cette commande est le début d'un groupe \TeX ien.

3.1 Détection du premier caractère

D'habitude les clés ont la forme `<clef>=<valeur>` et la manière dont de telles clés sont manipulées est présentée ci-après. Toutefois on peut définir une syntaxe différente pour certaines parties des arguments de `\pgfkeys`. Comme il s'agit d'une option plutôt avancée, on peut souhaiter sauter cette partie à la première lecture. On a placé cette présentation ici parce que cette détection est la première chose effectuée lorsqu'une clé est traitée avant qu'aucune autre opération ne soit faite.

La commande `\pgfkeys` et ses variantes décomposent leur argument en une liste de `<chaîne>`s — comprendre **chaîne de caractères** (*string*) — séparées par des virgules. Par défaut, chacune de ces `<chaîne>`s doit avoir la forme `<clef>=<valeur>` ou la forme `<clef>` où la valeur est manquante. Toutefois on peut vouloir que certaines de ces chaînes soient interprétées différemment. P. ex. lorsqu'une `<chaîne>` a la forme `"<texte>"`, on peut vouloir qu'elle soit interprétée comme si on avait écrit `label text=<texte>`. Dans ce cas, on pourrait écrire :

```
\myset{red, "main valve", thick}
```

au lieu du plus verbeux

```
\myset{red, label text=main valve, thick}
```

Un exemple d'une telle réinterprétation de la syntaxe est faite dans la bibliothèque `quotes` qui permet d'écrire des choses comme

```
a  $\xrightarrow{1}$  b  $\xrightarrow{0}$  c
```

```
\tikz \graph { a ->["1" red] b ->["0"] c };
```

au lieu de

```
a  $\xrightarrow{1}$  b  $\xrightarrow{0}$  c
```

```
\tikz \graph { a ->[edge node={node[red,auto]{1}}] b ->[edge label=0] c };
```

Afin de détecter si une `<chaîne>` a une syntaxe spéciale, on peut demander ce que le *premier caractère* de cette `<chaîne>` soit analysé par l'**analyseur de clef** (*key parser*). Si le premier caractère correspond à un des caractères qui ont été marqués comme spéciaux, la `<chaîne>` n'est pas interprétée comme une paire de clé-valeur ordinaire mais elle est passée en argument d'une macro spéciale qui doit s'occuper de la `<chaîne>`. Quand cette macro a fini, l'analyse continue avec la `<chaîne>` suivante dans la liste.

Afin de créer une syntaxe spéciale pour la manipulation d'une `<chaîne>` commençant par un certain caractère, il faut faire deux choses :

1. on *enclenche* d'abord la détection du premier caractère puisque, par défaut, elle n'est pas active pour des raisons d'efficacité — toutefois la surcharge est plutôt faible. On utilise la clé suivante :

`/handlers/first char syntax=<true ou false>` (défaut 'true', initialement 'false') (sans défaut)

2. ensuite, pour traiter d'une façon particulière les chaînes commençant par un certain `<caractère>`, on doit entreposer une macro dans la clé suivante :

`/handlers/first char syntax/<signification du caractere>` (sans valeur)

Cette `<signification du caractere>` doit être le texte que la commande \TeX ienne `\meaning` retourne pour une macro rendue `\let`-égale au caractère. P. ex. si les chaînes commençant par " doivent être traitées d'une façon particulière, `<signification du caractere>` devrait être `the character "` puisque c'est ce que \TeX retourne quand on écrit :

```
the character " \let\mycharacter="
\meaning\mycharacter
```


Ensuite, la clé `/handlers/first char syntax/⟨signification du caractère⟩` doit être évaluée — à l'aide de `\pgfkeyssetvalue` ou à l'aide du manipulateur `/.initial` — afin d'entreposer une `⟨macro⟩`.

Dans ce cas si une `⟨chaîne⟩` commence par le `⟨caractère⟩` — les blancs du début de la `⟨chaîne⟩` sont supprimés au préalable — alors la `⟨macro⟩` est appelée avec `⟨chaîne⟩` comme argument.

Voyons maintenant un exemple. Nous installons deux manipulateurs, un pour les chaînes commençant par " et un pour celles commençant par <.

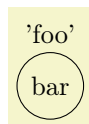
Quoted: "foo". Pointed: <bar>.

```
\pgfkeys{
  /handlers/first char syntax=true,
  /handlers/first char syntax/the character "/.initial=\myquotemacro,
  /handlers/first char syntax/the character </.initial=\mypointedmacro,
}

\def\myquotemacro#1{Quoted: #1. }
\def\mypointedmacro#1{Pointed: #1. }

\ttfamily \pgfkeys{"foo", <bar>}
```

Naturellement, dans les exemples précédents, les deux macros manipulatrices n'accomplissaient rien de bien excitant. Dans le suivant, nous créons une macro plus élaborée qui singe une partie du comportement de la bibliothèque `quotes`, uniquement pour les *quotes simples*³ :



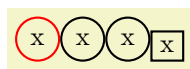
```
\pgfkeys{
  /handlers/first char syntax=true,
  /handlers/first char syntax/the character '/.initial=\mysinglequotemacro
}

\def\mysinglequotemacro#1{\pgfkeysalso[label={#1}]}

\tikz \node [circle, 'foo', draw] {bar};
```

Notez que dans l'exemple ci-dessus, la macro `\mysinglequotemacro` reçoit la chaîne complète, *simples quotes* compris. Son boulot est de s'en débarrasser si nécessaire.

Le mécanisme de détection du premier caractère permet de réaliser des transformations assez puissantes sur la syntaxe des clés — à condition de pouvoir attacher cette syntaxe sur le seul premier caractère. Dans l'exemple suivant on pourra placer une expression entre parenthèses devant une paire de clé-valeur et cette paire ne sera exécutée — prise en compte — que si l'expression entre parenthèse est évaluée à VRAI.



```
\pgfkeys{
  /handlers/first char syntax=true,
  /handlers/first char syntax/the character (</.initial=\myparamacro
}

\def\myparamacro#1{\myparaparser#1\someendtext}
\def\myparaparser(#1)#2\someendtext{
  \pgfmathparse{#1}
  \ifx\pgfmathresult\onetext
    \pgfkeysalso{#2}
  \fi
}
\def\onetext{1}

\foreach \i in {1,...,4}
  \tikz \node [draw, thick, rectangle, (pi>\i) circle, (pi>\i*2) draw=red] {x};
```

3.2 Arguments par défaut

Les arguments de la commande `\pgfkeys` peuvent être de la forme `⟨clef⟩=⟨valeur⟩` ou `⟨clef⟩` sans la valeur. Dans le deuxième cas, `\pgfkeys` tentera de fournir une **valeur par défaut** (*default value*) pour la `⟨valeur⟩`. Si une telle valeur par défaut est définie, elle sera utilisée comme si on avait écrit `⟨clef⟩=⟨valeur par défaut⟩`.

3. NdTds : J'appelle *quote simple* ce que l'on obtient, par défaut, en tapant la touche qui porte aussi le chiffre 4 sur un clavier de PC. Caractère inusité en typographie que d'aucuns persistent à nommer « apostrophe » !

On décrit, dans ce qui suit, comment sont déterminées les valeurs par défaut ; toutefois on utilisera généralement les manipulateurs `/.default` et `/.value required` comme expliqué en 4.2, page 13 et on peut vouloir passer au-dessus des détails suivants.

Quand `\pgfkeys` rencontre une $\langle clef \rangle$ sans signe égal, voici ce qui se passe :

1. l'entrée est remplacée par $\langle clef \rangle = \backslash pgfkeysnovalue$. En fait, les commandes `ma clef` et `ma clef = \pgfkeysnovalue` ont exactement le même effet et on peut simuler une valeur manquante en fournissant pour valeur la commande `\pgfkeysnovalue`, ce qui est parfois utile ;
2. si la valeur est `\pgfkeysnovalue`, alors la commande vérifie que la sous-clé $\langle clef \rangle /.@def$ existe. p. ex., si on écrit `\pgfkeys {/ma clef}`, alors la commande vérifie l'existence de `/ma clef/.@def` ;
3. si la clé $\langle clef \rangle /.@def$ existe, alors les lexèmes qui y sont entreposés sont utilisés pour la $\langle valeur \rangle$;
4. si la clé n'existe pas, `\pgfkeysnovalue` est utilisée comme valeur ;
5. à la fin, si la $\langle valeur \rangle$ est maintenant `\pgfkeysnovalue`, alors le code — ou quelque chose de tout à fait équivalent — `\pgfkeys {/errors/value required = \langle clef \rangle {}}` est exécuté. Ainsi, en changeant cette clé, on peut changer le message d'erreur affiché ou traiter l'absence de valeur d'une autre manière.

3.3 Clés qui exécutent des commandes

Après le processus de transformation décrit ci-dessus, on arrive à une paire de la forme $\langle clef \rangle = \langle valeur \rangle$ dans laquelle $\langle clef \rangle$ est une clé complète. Plusieurs choses différentes peuvent se passer maintenant mais en tous les cas la macro `\pgfkeyscurrentkeys` est définie de telle sorte que son développement est le texte de la $\langle clef \rangle$ en cours de traitement.

La première chose testée est l'existence de la clé $\langle clef \rangle /.@cmd$. Si c'est le cas, on suppose que cette clé contient le code d'une macro et cette macro est exécutée. L'argument de cette macro est $\langle valeur \rangle$ immédiatement suivie de `\pgfeov` qui signifie « fin de valeur » — *end of value*. La $\langle valeur \rangle$ n'est pas entourée d'accolades. Quand le code a été exécuté, `\pgfkeys` continue avec la paire suivante de la $\langle liste de paire \rangle$.

Il peut sembler étrange que la macro entreposée dans la clé $\langle clef \rangle /.@cmd$ ne soit pas simplement exécuté avec l'argument $\langle valeur \rangle$. Toutefois, l'approche suivie dans l'extension `pgfkeys` permet plus de souplesse. p. ex., supposons que l'on veuille une clé qui attende une $\langle valeur \rangle$ sous la forme « $\langle texte \rangle + \langle autre\ texte \rangle$ » et que $\langle texte \rangle$ et $\langle autre\ texte \rangle$ soient entreposés dans deux macros distinctes. Voici comment procéder :

```

\ a : salut, \ b : monde.
\def\panier#1+#2\pgfeov{\def\ a{#1}\def\ b{#2}}
\pgfkeyslet{/ma clef/.@cmd}{\panier}
\pgfkeys{/ma clef=salut+monde}

\ a | : \ a, \ b | : \ b.

```

Naturellement, définir du code à entreposer dans une clé de la manière précédente est trop maladroit. Les commandes qui suivent facilitent un peu les choses mais la façon habituelle de le faire est d'utiliser un des manipulateurs décrits en 4.3, page 14.

`\pgfkeysdef{\langle clef \rangle}{\langle code \rangle}`

cette commande définit temporairement une macro TeXienne de liste d'arguments `#1\pgfeov` puis rend $\langle clef \rangle /.@cmd$ égale à cette macro. L'effet de tout cela est que l'on a défini du code pour la clé $\langle clef \rangle$ de telle sorte que lorsque l'on écrit `\pgfkeys {\langle clef \rangle = \langle valeur \rangle }`, le $\langle code \rangle$ est exécuté avec toutes les occurrences de `#1` remplacées par $\langle valeur \rangle$. Ce comportement est totalement semblable à celui de la commande `\define@key` de `keyval` et `xkeyval`.

```

salut, salut.
\pgfkeysdef{/ma clef}{#1, #1.}
\pgfkeys{/ma clef=salut}

```

`\pgfkeysedef{\langle clef \rangle}{\langle code \rangle}`

cette commande travaille comme `\pgfkeysdef` sauf qu'elle utilise `\edef` au lieu de `\def` pour définir la macro de la clé. Si vous ne connaissez pas la différence entre les deux, vous n'avez pas besoin de cette commande ; si vous connaissez la différence, vous savez aussi quand vous en avez besoin.

`\pgfkeysdefnargs{\langle clef \rangle}{\langle nombre d'arguments \rangle}{\langle code \rangle}`

cette commande travaille comme `\pgfkeysdef` mais permet de fournir un $\langle nombre d'arguments \rangle$ quelconque — de 0 à 9.

<code>\a : 'salut', \b : 'monde'.</code>	<pre>\pgfkeysdefnargs{/ma clef}{2}{\def\a{#1}\def\b{#2}} \pgfkeys{/ma clef= {salut} {monde}} a : 'a', b : 'b'.</pre>
--	--

La clé ainsi définie attend $\langle \text{nombre d'arguments} \rangle$ arguments.

`\pgfkeysdefnargs` $\langle \text{clef} \rangle$ $\langle \text{nombre d'arguments} \rangle$ $\langle \text{code} \rangle$

est la version `\edef` de `\pgfkeysdefnargs`.

`\pgfkeysdefargs` $\langle \text{clef} \rangle$ $\langle \text{patron d'arguments} \rangle$ $\langle \text{code} \rangle$

cette commande travaille comme `\pgfkeysdefnargs` mais permet de fournir un $\langle \text{patron d'arguments} \rangle$ ⁴ quelconque au lieu de simplement le nombre d'arguments.

<code>\a : salut, \b : monde.</code>	<pre>\pgfkeysdefargs{/ma clef}{#1+#2}{\def\a{#1}\def\b{#2}} \pgfkeys{/ma clef=salut+monde} a : \a, b : \b.</pre>
--------------------------------------	--

Notez que `\pgfkeysdefnargs` est meilleure que `\pgfkeysdefargs` quand il suffit, simplement, de compiler les arguments⁵.

`\pgfkeysdefargs` $\langle \text{clef} \rangle$ $\langle \text{patron d'arguments} \rangle$ $\langle \text{code} \rangle$

est la version `\edef` de `\pgfkeysdefargs`.

3.4 Clés qui entreposent des valeurs

Continuons avec ce qui se passe quand `\pgfkeys` traite la paire courante et que la sous-clé $\langle \text{clef} \rangle / .@cmd$ n'est pas définie. Alors on regarde si la $\langle \text{clef} \rangle$ elle-même est définie — a reçu préalablement une valeur à l'aide, p. ex., de `\pgfkeyssetvalue`. Dans ce cas, les lexèmes entreposés dans $\langle \text{clef} \rangle$ sont remplacés par la $\langle \text{valeur} \rangle$ et `\pgfkeys` continue avec la paire suivante dans la liste.

3.5 Clés manipulées

Si ni la $\langle \text{clef} \rangle$ elle-même ni la sous-clé $\langle \text{clef} \rangle / .@cmd$ ne sont définies, alors la $\langle \text{clef} \rangle$ ne peut être traitée « toute seule », il faut un $\langle \text{manipulateur} \rangle$ pour cette clé. La plus grande partie de la puissance de `pgfkeys` provient de tels manipulateurs.

Rappelons-nous que la $\langle \text{clef} \rangle$ est toujours une clé complète — si ça ne l'était pas à l'origine, elle a, à ce stade, été complétée. Cette $\langle \text{clef} \rangle$ est partagée en deux :

1. le $\langle \text{chemin} \rangle$ de la $\langle \text{clef} \rangle$ — tout ce qui précède le dernier / — est placé dans `\pgfkeyscurrentpath` ;
2. le $\langle \text{nom} \rangle$ de la $\langle \text{clef} \rangle$ — tout ce qui suit le dernier / — est placé dans `\pgfkeyscurrentname`.

Il est recommandé — mais pas indispensable — que le nom d'un manipulateur commence par un point — mais pas par `/.@` —, afin qu'il soit facile à repérer par le lecteur.

Pour des raisons d'efficacité, ces deux macros ne sont définies qu'une fois arrivé à ce stade ; aussi, quand le code d'une clé est exécuté de la manière « habituelle », ces deux macros ne sont pas définies.

`\pgfkeys` regarde maintenant si la clé `/handlers/⟨nom⟩/.@cmd` existe. Si c'est le cas, elle devrait contenir une commande est cette commande est alors exécutée exactement de la manière décrite dans 3.3, page 9. Ainsi, le code reçoit, pour argument, la $\langle \text{valeur} \rangle$ qui était originellement destinée à $\langle \text{clef} \rangle$ suivie de `\pgfeov`. C'est la tâche des manipulateurs de faire quelque chose d'utile avec la $\langle \text{valeur} \rangle$.

Écrivons, p. ex., un manipulateur qui placera dans le fichier de rapport `.log` la valeur entreposée dans une clé. Appelons le `/.print to log`. L'idée est que, quand on essaiera d'employer la clé `/ma clef/.print to log` alors cette clé ne sera pas définie et le manipulateur sera exécuté. Le manipulateur aura alors accès au chemin de la clé — qui vaut `/ma clef` — grâce à `\pgfkeyscurrentpath`. Il peut alors regarder la valeur entreposée dans cette clé et l'imprimer.

4. NdTds : il s'agit ici de paramètres à la T_EX, comme on les utilise avec `\def`, p. ex.

5. Quand on utilise les clés ainsi définies, la variante `defnargs` autorise les espaces entre les arguments alors que la variante `defargs` ne les autorise pas — elle considère les espaces comme faisant partie des arguments.

```

\pgfkeysdef{/handlers/.print to log}
{
  \pgfkeysgetvalue{\pgfkeyscurrentpath}{\temp}
  \writetolog{\temp}
}
\pgfkeyssetvalue{/ma clef}{Salut!}
...
\pgfkeys{/ma clef/.print to log}

```

Le code ci-dessus imprimera **Salut!** dans le fichier `.log` pourvu que la macro `\writetolog` soit définie correctement.

Passons à un manipulateur plus intéressant. Programmons en un qui définira une clé de telle sorte que quand la clé est utilisée du code est exécuté. Ce code sera fourni en tant que *⟨valeur⟩*. Tout ce que doit faire le manipulateur c'est appeler `\pgfkeys` sur la paire formée du chemin de la clé — qui ne contient plus le nom du manipulateur — et du paramètre.

```

(salut) \pgfkeysdef{/handlers/.mon code}{\pgfkeysdef{\pgfkeyscurrentpath}{#1}}
\pgfkeys{/ma clef/.mon code=#1}
\pgfkeys{/ma clef=salut}

```

Il existe quelques paramètres pour les clés manipulées qui s'avère utiles dans quelques — rares peut-être — cas spéciaux :

`/handler config=all|only existing|full or existing` (sans défaut, initialement `all`)

change la configuration initiale de la manière dont sont utilisés les manipulateurs.

Cette configuration ne concerne que les utilisateurs avancés et rarement nécessaire.

`all` Le réglage préconfiguré `all` fonctionne comme décrit ci-dessus et n'impose aucune restriction sur le processus de réglage des clés.

`only existing` La valeur `only existing` modifie l'algorithme des clés manipulées comme suit : un manipulateur *⟨nom de clef⟩*/*⟨manipulateur⟩* sera exécuté uniquement si *⟨nom de clef⟩* est une clé qui entrepose sa valeur directement ou si c'est une clé de commande pour laquelle `/.@cmd` existe. Si *⟨nom de clef⟩* n'existe pas déjà, la chaîne *⟨nom de clef⟩*/*⟨manipulateur⟩* est considérée comme une clé inconnue et la procédure décrite dans la section suivante est appliquée — pour le chemin de *⟨nom de clef⟩*.

Definition initiale.Re-Definition.Unknown key '.

```

% [1]
\pgfkeys{/the/key/.code={Definition initiale. }}
\pgfkeys{/handlers/.unknown/.code={Unknown key '\pgfkeyscurrentkey'. }}
% [2]
\pgfkeys{/the/key}
% [3]
\pgfkeys{/handler config=only existing}
% [4]
\pgfkeys{/the/key/.code={Re-Definition. }}
% [5]
\pgfkeys{/the/key}
% [6]
\pgfkeys{/the/other key/.code={New definition. }}

```

Je place ci-dessous les commentaires du code, l'utilisation de `pdflatex` sur un codage UTF-8 pose quelques difficultés au code de l'environnement `codeexample` utilisé dans cette documentation.

En [1] on définit une clé et un manipulateur d'erreur. En [2], l'utilisation de la clé `/the/key` produit « `Definition initiale.` ». En [3], on change la configuration ce qui permet, en [4], de redéfinir la clé `/the/key`. De fait, en [5], l'utilisation de la clé `/the/key` produit, cette fois, « `Re-Definition.` ». En [6], on tente une manœuvre interdite : la définition d'une nouvelle clé. Cela entraîne la recherche de `/the/other key/.unknown` et le manipulateur `/handlers/.unknown` produit, pour finir, « `Unknown key '/the/other key/.code'` ».

Il est nécessaire d'exclure quelques manipulateurs de clés de cette procédure. La procédure détaillée est la suivante :

1. si on rencontre une clé manipulée comme `/un chemin/une clef/.un manipulateur=valeur`, on vérifie que l'on peut invoquer ce manipulateur. C'est le cas si

- il existe, pour cette clé, une exception à `only existing`, voir ci-après ;
- la clé `/un chemin/une clef` existe déjà — soit comme clé d’entreposage direct soit avec le suffixe `/.@cmd`.

2. si la vérification réussit, tout fonctionne comme avant ;
3. si la vérification échoue, on considère la clé complète comme inconnue. Dans ce cas, le traitement des clés inconnues s’applique comme décrit dans la section suivante. Ici, le chemin de la clé courante sera `/un chemin` et le nom de la clé `une clef/.un manipulateur`.

Une conséquence de cette configuration est de fournir un processus plus sensé de traitement des clés manipulées si un chemin de recherche de clés est actif, voir 3.6page 12, pour un exemple.

`full or existing` Finalement, le choix `full or existing` est une variante de `only existing` : il fonctionne de la même façon pour les clés qui n’ont pas de chemin complet. p. ex., le style

```
\pgfkeys{/mon chemin/.cd, clef/.style={...}}
```

ne peut être que redéfini : il n’a pas de chemin complet donc le mécanisme de `only existing` s’applique. Par contre, le style

```
\pgfkeys{/mon chemin/clef/.style={...}}
```

fonctionnera encore. Cela permet à l’utilisateur de redéfinir les caractéristiques `only existing` s’il sait ce qu’il fait — et fournit un chemin complet pour la clé.

```
/handler config/only existing/add exception={⟨nom de manipulateur⟩} (sans défaut)
```

permet d’ajouter des exceptions aux fonctionnalités de `/handler config = only existing`. Les exceptions initiales sont `/.cd`, `/.try`, `/.retry`, `/.lastretry` et `/.unknown`. La valeur `⟨nom de manipulateur⟩` doit être le nom d’un manipulateur de clé.

3.6 Clés inconnues

Parfois, la clé n’est pas définie, sa sous-clé `/.@cmd` ne l’est pas plus et aucun manipulateur n’est définie pour elle. Dans ce cas, on vérifie si la clé `⟨chemin courant⟩/.unknown/.@cmd` existe. Ainsi, quand on essaie d’utiliser la clé `/tikz/bizarre`, on vérifie l’existence de `/tikz/.unknown/.@cmd`. Si cette clé existe — et, de fait, elle existe —, elle est exécutée. Le code peut alors se débrouiller pour donner du sens à la clé. De fait, le manipulateur pour TikZ essaiera d’interpréter le nom de la clé comme une couleur, une spécification de flèche ou une option de PGF.

On peut définir des manipulateurs de clés inconnues pour ses propres clés en définissant simplement le code de la clé `⟨préfixe de mon chemin⟩/.unknown`. C’est ce qui permet également de définir des « chemins de recherche ». L’idée est la suivante : on voudrait que la recherche des clés se fasse non pas seulement dans un seul chemin courant mais dans plusieurs. Supposons que l’on veuille chercher les clés dans `/a`, `/b` et `/b/c`. Définissons, pour ce faire, une clé `/my search path` :

```
\pgfkeys{/my search path/.unknown/.code=
  {%
  |let\searchname=\pgfkeyscurrentname%
  \pgfkeysalso{%
    /a/\searchname/.try=#1,
    /b/\searchname/.retry=#1,
    /b/c/\searchname/.retry=#1%
  }%
  }%
}
\pgfkeys{/my search path/.cd,foo,bar}
```

Dans le code précédent, `foo` et `bar` — dans l’appel de `\pgfkeys` de la dernière ligne — seront cherchées dans les trois dossiers `/a`, `/b` et `/b/c`. Avant d’implémenter des chemins de recherche en suivant ce modèle, on pourra considérer le manipulateur `/.search also` présenté ci-après.

Si la clé `⟨chemin courant⟩/.unknown/.@cmd` n’existe pas, on invoque à sa place le manipulateur `/handlers/.unknown` qui est toujours défini et qui, par défaut, imprime un message d’erreur.

3.7 Chemins de recherche et clés manipulées

Un cas particulier peut apparaître dans l’exemple des chemins de recherche ci-dessus. Que se passe-t-il si on veut changer un style ? P. ex.

```
\pgfkeys{/my search path/.cd,custom/.style={variables}}
```

pourrait signifier un style dans `/my search path/`, `/a/`, `/b/` ou même `/b/c/` !

Du fait des règles gouvernant les clés manipulées, la réponse est `/my search path/custom/.style = {variable}`. Il peut être utile de modifier ce comportement par défaut. Quelque chose d'utile serait de rechercher les styles nommés `custom existants` et de les redéfinir. Par exemple, si le style `/b/custom` existe, l'affectation `custom/.style = {variable}` devrait le redéfinir lui plutôt que `/my search path/custom`. On peut le faire en employant `handler config` :

```
This is '/b/custom'. This is '/b/custom'.Modified.
```

```
\pgfkeys{/my search path/.unknown/.code=
  {%
    \let\searchname=\pgfkeyscurrentname%
    \pgfkeysalso{%
      /a/\searchname/.try=#1,
      /b/\searchname/.retry=#1,
      /b/c/\searchname/.retry=#1%
    }%
  }%
}

% [1]
\pgfkeys{/b/custom/.code={This is '\pgfkeyscurrentkey'. }}
% [2]
\pgfkeys{/handler config=only existing}
% [3]
\pgfkeys{/my search path/.cd,custom}
% [4]
\pgfkeys{/my search path/.cd,custom/.append code={Modified. }}
% [5]
\pgfkeys{/my search path/.cd,custom}
```

En [1], on définit `/b/custom` puis, en [2], on reconfigure le traitement des manipulateurs. En [3] la procédure de recherche de chemin trouve `/b/custom` ce qui produit « `This is '/b/custom'` ». En [4], du fait de la reconfiguration, le code trouve `/b/custom` au lieu de définir `/my search path/custom` et, en [5], on retrouve `/b/custom` ce qui produit « `This is '/b/custom' Modified` ».

On peut réaliser une approche légèrement différente des chemins de recherche avec le manipulateur `/.search also` comme on le verra ci-dessous.

4 Manipulateurs

Nous décrivons maintenant les manipulateurs définis par défaut. On peut aussi en définir de nouveaux comme décrit en 3.5, page 10.

4.1 Manipulateurs de gestion du chemin

Manipulateur `<clef>/ .cd`

Ce manipulateur définit le chemin par défaut comme égal à `<clef>`. Notez que le chemin par défaut est redéfini comme égal à `/` au début de chaque appel à `\pgfkeys`.

Exemple: `\pgfkeys{/tikz/.cd,...}`

Manipulateur `<clef>/ .is family`

Ce manipulateur assure les réglages qui font que, quand `<clef>` est exécutée, le chemin par défaut est égal à `<clef>`. Une utilisation typique est :

```
\pgfkeys{/tikz/.is family}
\pgfkeys{tikz,line width=1cm}
```

L'effet de ce manipulateur est identique à ce que l'on aurait avec `<clef>/ .style = <clef>/ .cd` mais le code produit par `/.is family` est plus rapide.

4.2 Valeurs par défaut

Manipulateur `<clef>/ .default=<valeur>`

Définit la valeur par défaut de $\langle clef \rangle$ à $\langle valeur \rangle$. Cela entraîne qu'à chaque appel de `\pgfkeys` dans lequel on ne passe pas de valeur à la $\langle clef \rangle$, c'est cette $\langle valeur \rangle$ qui sera utilisée.

Exemple: `\pgfkeys{/width/.default=1cm}`

Manipulateur $\langle clef \rangle/.value\ required$

Value required signifie « valeur exigée ». Ce manipulateur exécute la clé d'erreur `/errors/value required` à chaque fois que l'on appelle la $\langle clef \rangle$ sans lui fournir de valeur.

Exemple: `\pgfkeys{/width/.value required}`

Manipulateur $\langle clef \rangle/.value\ forbidden$

Value forbidden signifie « valeur interdite ». Ce manipulateur exécute la clé d'erreur `/errors/value forbidden` à chaque fois que l'on appelle la $\langle clef \rangle$ en lui fournissant une valeur.

Ce manipulateur fonctionne en ajoutant du code au code attaché à la clé : en conséquence on doit d'abord définir la clé puis utiliser ce manipulateur⁶.

```
\pgfkeys{/ma clef/.code=I do not want an argument!}
\pgfkeys{/ma clef/.value forbidden}

\pgfkeys{/ma clef} % Oui
\pgfkeys{/ma clef=foo} % Erreur!
```

4.3 Définir le code d'une clé

Il existe plusieurs manipulateurs pour définir le code attaché à une clé.

Manipulateur $\langle clef \rangle/.code=\langle code \rangle$

Ce manipulateur exécute `\pgfkeysdef` avec, pour arguments, $\langle clef \rangle$ et $\langle code \rangle$. Cela signifie qu'ensuite, à chaque fois que la $\langle clef \rangle$ est utilisée, le $\langle code \rangle$ est exécuté. Plus précisément, quand $\langle clef \rangle = \langle valeur \rangle$ est trouvé dans une liste de clés, $\langle code \rangle$ est exécuté avec chaque occurrence de `#1` remplacée par $\langle valeur \rangle$. Comme toujours, si on ne fournit pas de $\langle valeur \rangle$, la valeur par défaut est utilisée si elle existe, sinon `\pgfkeysnovalue` l'est.

Il est permis à $\langle code \rangle$ d'appeler la commande `\pgfkeys`. Il lui est même permis d'appeler `\pgfkeysalso`, ce qui est utile pour les styles — voir plus bas.

```
\pgfkeys{/par indent/.code={\parindent=#1},/par indent/.default=2em}
\pgfkeys{/par indent=1cm}
...
\pgfkeys{/par indent}
```

Manipulateur $\langle clef \rangle/.ecode=\langle code \rangle$

Ce manipulateur fonctionne comme `/.code` mais utilise `\pgfkeysedef`.

Manipulateur $\langle clef \rangle/.code\ 2\ args=\langle code \rangle$

Ce manipulateur fonctionne comme `/.code` mais quand $\langle code \rangle$ est utilisé il attend deux arguments au lieu d'un seul. Cela signifie que quand on trouve $\langle clef \rangle = \langle valeur \rangle$ dans une liste de clés, la $\langle valeur \rangle$ doit être composée de deux arguments. P. ex., $\langle valeur \rangle$ peut être `{premier}{second}`, et, dans ce cas, $\langle code \rangle$ est exécuté avec chaque occurrence de `#1` remplacée par `premier` et chaque occurrence de `#2` remplacée par `second`.

```
\pgfkeys{/page size/.code 2 args={\paperheight=#2\paperwidth=#1}}
\pgfkeys{/page size={30cm}{20cm}}
```

Le deuxième argument est optionnel : si on ne le fournit pas, il sera considéré comme égal à une chaîne vide.

Du fait de la manière spéciale dont la $\langle valeur \rangle$ est analysée, si $\langle valeur \rangle$ vaut `premier` — sans les accolades — alors `#1` sera `p` et `#2` `remier`.

6. NdTds : Dans l'exemple qui suit, la phrase anglaise a un double sens : on peut la rendre comme écrit ci-dessous « je ne veux pas d'argument » mais elle signifie aussi « je ne veux pas me disputer ». J'ai préféré cette note à une tentative de rendre ce *double entendre* comme disent nos amis d'Outre-Manche.

Manipulateur $\langle clef \rangle / .e\text{code } 2 \text{ args} = \langle code \rangle$

Ce manipulateur fonctionne comme `/.code 2 args` mais utilise `\edef` au lieu de `\def` pour définir la macro.

Manipulateur $\langle clef \rangle / .code \ n \ \text{args} = \{ \langle \text{nombre d'arguments} \rangle \} \{ \langle code \rangle \}$

Ce manipulateur aussi fonctionne comme `/.code` sauf que l'on peut préciser un nombre d'arguments compris entre 0 et 9.

```
First='A', Second='B' \pgfkeys{/a key/.code n args={2}{First='#1', Second='#2'}}
\pgfkeys{/a key={A}{B}}
```

Contrairement à `/.code 2 args`, il doit y avoir exactement $\langle \text{nombre d'arguments} \rangle$ arguments, ni plus ni moins, et ces arguments doivent être proprement délimités — voir ci-dessus.

Manipulateur $\langle clef \rangle / .e\text{code } \ n \ \text{args} = \{ \langle \text{nombre d'arguments} \rangle \} \{ \langle code \rangle \}$

Ce manipulateur fonctionne comme `/.code n args` mais utilise `\edef` au lieu de `\def` pour définir la macro.

Manipulateur $\langle clef \rangle / .code \ \text{args} = \{ \langle \text{patron d'arguments} \rangle \} \{ \langle code \rangle \}$

Ce manipulateur fournit la façon la plus souple de définir une clé de commande : on peut spécifier un $\langle \text{patron d'arguments} \rangle$ quelconque. Un tel patron est du type habituel en T_EX⁷. P. ex., supposons que ce $\langle \text{patron d'arguments} \rangle$ soit $(\#1/\#2)$ et que $\langle clef \rangle = \langle \text{valeur} \rangle$ soit rencontrée avec $\langle \text{valeur} \rangle$ donnée comme $(\text{premier}/\text{second})$ alors le $\langle code \rangle$ sera exécuté avec chaque occurrence de $\#1$ remplacée par `premier` et chaque occurrence de $\#2$ remplacée par `second`. La $\langle \text{valeur} \rangle$ effective est comparée au patron de la manière T_EXienne habituelle.

```
\pgfkeys{/page size/.code args={#1 and #2}{\paperheight=#2\paperwidth=#1}}
\pgfkeys{/page size=30cm and 20cm}
```

On notera qu'il faut préférer `/.code n args` lorsque l'on a simplement besoin d'un nombre précis d'arguments⁸ — quand on utilise la clé ainsi définie, les espaces entre les arguments sont ignorés alors qu'avec `/.code args` il font partie de l'argument.

Manipulateur $\langle clef \rangle / .e\text{code } \ \text{args} = \{ \langle \text{patron d'arguments} \rangle \} \{ \langle code \rangle \}$

Ce manipulateur fonctionne comme `/.code args` mais utilise `\edef` au lieu de `\def` pour définir la macro.

On a également des manipulateurs pour modifier des clés existantes.

Manipulateur $\langle clef \rangle / .add \ \text{code} = \{ \langle \text{prefixe} \rangle \} \{ \langle \text{suffixe} \rangle \}$

Ce manipulateur ajoute — *add* — du code à une clé existante. Le code $\langle \text{prefixe} \rangle$ est ajouté devant le $\langle code \rangle$ entreposé dans $\langle clef \rangle / .cmd$, $\langle \text{suffixe} \rangle$ est ajouté après ce même $\langle code \rangle$. L'un ou l'autre peut être vide. On ne peut pas changer la liste des arguments de $\langle code \rangle$ avec ce manipulateur. On notera que autant $\langle \text{préfixe} \rangle$ que $\langle \text{suffixe} \rangle$ peuvent contenir des paramètres comme $\#2$.

```
\pgfkeys{/par indent/.code={\parindent=#1}}
\newdimen\myparindent
\pgfkeys{/par indent/.add code={}\myparindent=#1}}
...
\pgfkeys{/par indent=1cm} % \parindent et \myparindent
                          % prennent la valeur 1~cm
```

Manipulateur $\langle clef \rangle / .prefix \ \text{code} = \langle \text{prefixe} \rangle$

Ce manipulateur est un raccourci pour $\langle clef \rangle / .add \ \text{code} = \{ \langle \text{prefixe} \rangle \} \{ \}$ c.-à-d. que le code $\langle \text{prefixe} \rangle$ est placé devant le code entreposé dans $\langle clef \rangle / .@cmd$.

Manipulateur $\langle clef \rangle / .append \ \text{code} = \langle \text{suffixe} \rangle$

Ce manipulateur est un raccourci pour $\langle clef \rangle / .add \ \text{code} = \{ \} \{ \langle \text{suffixe} \rangle \}$.

7. NdTds : c'est ce que Knuth appelle un *parameter text* dans le T_EXbook.

8. NdTds : autrement dit, on écrit du code comme pour `\newcommand`.

4.4 Définir des styles

Les manipulateurs qui suivent permettent de définir des styles. Un style est une liste de clés qui sont traitées à chaque fois que l'on place le style dans une liste de clés. Ainsi, un style « remplace » une liste de clés. On peut paramétrer les styles comme on le fait avec le code normal.

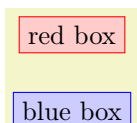
Manipulateur $\langle clef \rangle /.style = \langle liste\ de\ clefs \rangle$

Ce manipulateur règle les choses pour que, à chaque fois que $\langle clef \rangle = \langle valeur \rangle$ est rencontrée dans une liste de clés, alors la $\langle liste\ de\ clefs \rangle$, dans laquelle chaque occurrence de #1 est remplacée par $\langle valeur \rangle$, est traitée à sa place.

On peut obtenir le même effet en utilisant $\langle clef \rangle /.code = \backslashpgfkeysalso\{liste\ de\ clefs\}$. Cela signifie que le code attaché à une clé peut également exécuter d'abord du code normal puis traiter d'autres clés.

```
\pgfkeys{/par indent/.code={\parindent=#1}}
\pgfkeys{/no indent/.style={/par indent=0pt}}
\pgfkeys{/normal indent/.style={/par indent=2em}}
\pgfkeys{/no indent}
...
\pgfkeys{/normal indent}
```

L'exemple suivant montre un style paramétré :



```
\begin{tikzpicture}[outline/.style={draw=#1,fill=#1!20}]
\node [outline=red] {red box};
\node [outline=blue] at (0,-1) {blue box};
\end{tikzpicture}
```

Manipulateur $\langle clef \rangle /.estyle = \langle liste\ de\ clefs \rangle$

Ce manipulateur fonctionne comme $/.style$ mais le $\langle code \rangle$ est défini en utilisant \backslashedef au lieu de \backslashdef c.-à-d. que toutes les macros présentes dans le $\langle code \rangle$ sont développées avant de sauvegarder le style.

On trouve pour les styles les manipulateurs suivants qui correspondent à ceux déjà vus pour les codes, voir 4.3 à partir de la page 14.

Manipulateur $\langle clef \rangle /.style\ 2\ args = \langle liste\ de\ clefs \rangle$

Ce manipulateur fonctionne comme $/.code\ 2\ args$ mais uniquement pour les styles. Ainsi, la $\langle liste\ de\ clefs \rangle$ peut contenir des occurrences de #1 et aussi de #2 et quand on utilise le style, on doit donner deux paramètres comme $\langle valeur \rangle$.

```
\pgfkeys{/paper height/.code={\paperheight=#1},/paper width/.code={\paperwidth=#1}}
\pgfkeys{/page size/.style\ 2\ args={/paper height=#1,/paper width=#2}}
\pgfkeys{/page size={30cm}{20cm}}
```

Manipulateur $\langle clef \rangle /.estyle\ 2\ args = \langle liste\ de\ clefs \rangle$

Ce manipulateur fonctionne comme $/.style\ 2\ args$ mais on emploie \backslashedef au lieu de \backslashdef pour définir la macro.

Manipulateur $\langle clef \rangle /.style\ args = \{ \langle patron\ d'arguments \rangle \} \{ \langle key\ list \rangle \}$

Ce manipulateur fonctionne comme $/.code\ args$ mais uniquement pour les styles.

Manipulateur $\langle clef \rangle /.estyle\ args = \{ \langle patron\ d'arguments \rangle \} \{ \langle code \rangle \}$

Ce manipulateur fonctionne comme $/.ecode\ args$ mais uniquement pour les styles.

Manipulateur $\langle clef \rangle /.style\ n\ args = \{ \langle nombre\ d'arguments \rangle \} \langle liste\ de\ clefs \rangle$

Ce manipulateur fonctionne comme $/.code\ n\ args$ mais uniquement pour les styles. Ici, la $\langle liste\ de\ clefs \rangle$ peut dépendre de tous les $\langle nombre\ d'arguments \rangle$ paramètres.

Manipulateur $\langle clef \rangle /.add\ style = \{ \langle liste\ prefixe \rangle \} \{ \langle liste\ suffixe \rangle \}$

Ce manipulateur fonctionne comme $/.add\ code$ mais uniquement pour les styles. Toutefois, on peut ajouter des styles à des clés qui ont été préalablement définies avec $/.code$ — il est également permis d'ajouter du $\langle code \rangle$ normal à une clé préalablement définie avec $/.style$. Quand on ajoute un style à

une $\langle clef \rangle$ préalablement définie avec $\langle code \rangle$, voici ce qui se passe : lorsque la $\langle clef \rangle$ est traitée, la $\langle liste\ prefixe \rangle$ de clés sera traitée d’abord, puis le $\langle code \rangle$, entreposé auparavant dans $\langle clef \rangle/.@cmd$, sera exécuté, enfin les clés de la $\langle liste\ suffixe \rangle$ seront traitées à leur tour.

```
\pgfkeys{/par indent/.code={\parindent=#1}}
\pgfkeys{/par indent/.add style={}/ma clef=#1}}
...
\pgfkeys{/par indent=1cm} % donne la valeur \a \parindent
                          % puis ex\ecute /ma clef=#1
```

Manipulateur $\langle clef \rangle/.prefix\ style=\langle liste\ prefixe \rangle$

Fonctionne comme `/.add style` mais uniquement pour la $\langle liste\ prefixe \rangle$ de clés.

Manipulateur $\langle clef \rangle/.append\ style=\langle liste\ suffixe \rangle$

Fonctionne comme `/.add style` mais uniquement pour la $\langle liste\ suffixe \rangle$ de clés.

4.5 Clés de choix, booléennes, d’entreposage

Quelques clés ont un code attaché plutôt « spécialisé ». P. ex., il arrive souvent que le code d’une clé se contente de mettre à vrai ou faux un si \TeX ien. Des manipulateurs prédéfinis facilitent, dans ces cas, la définition du code.

Cependant, commençons par quelques manipulateurs utiles à gérer la valeur directement entreposée dans une clé.

Manipulateur $\langle clef \rangle/.initial=\langle valeur \rangle$

Ce manipulateur règle la valeur de la $\langle clef \rangle$ à $\langle valeur \rangle$. On notera qu’aucune sous-clé n’est impliquée dans l’affaire. Une fois ce manipulateur utilisé, on peut — du fait des règles gouvernant les clés — changer la valeur de la $\langle clef \rangle$ avec un simple $\langle clef \rangle=\langle valeur \rangle$. De fait, ce manipulateur est utilisé pour régler la valeur initiale de la $\langle clef \rangle$.

```
\pgfkeys{/ma clef/.initial=red}
% "/ma clef" contient la valeur "red"
\pgfkeys{/ma clef=blue}
% "/ma clef" contient maintenant la valeur "blue"
```

On notera que, après le code de cet exemple, écrire `\pgfkeys{/ma clef}` n’aura pas l’effet que l’on pourrait attendre c.-à-d. insérer `blue` dans le texte principal. De fait, `/ma clef` sera changé en `/ma clef = \pgfkeysnovalue` et donc `\pgfkeysnovalue` sera entreposé dans `/ma clef`. Pour récupérer la valeur entreposée dans une clé, on doit utiliser le manipulateur `/.get`⁹.

Manipulateur $\langle clef \rangle/.get=\langle macro \rangle$

Exécute un `\let` afin que la $\langle macro \rangle$ contienne la valeur entreposée dans $\langle clef \rangle$.

```
blue \pgfkeys{/ma clef/.initial=red}
\pgfkeys{/ma clef=blue}
\pgfkeys{/ma clef/.get=\mymacro}
\mymacro
```

Manipulateur $\langle clef \rangle/.add=\{\langle valeur\ prefixe \rangle\}\{\langle valeur\ suffixe \rangle\}$

Ajoute la $\langle valeur\ prefixe \rangle$ avant la $\langle valeur \rangle$ entreposée dans la $\langle clef \rangle$ et la $\langle valeur\ suffixe \rangle$ après.

Manipulateur $\langle clef \rangle/.prefix=\{\langle valeur\ prefixe \rangle\}$

Ajoute la $\langle valeur\ prefixe \rangle$ avant la $\langle valeur \rangle$ entreposée dans la $\langle clef \rangle$.

Manipulateur $\langle clef \rangle/.append=\{\langle valeur\ suffixe \rangle\}$

Ajoute la $\langle valeur\ suffixe \rangle$ après la $\langle valeur \rangle$ entreposée dans la $\langle clef \rangle$.

Manipulateur $\langle clef \rangle/.link=\langle autre\ clef \rangle$

Entrepose la valeur `\pgfkeysvalueof{\langle autre\ clef \rangle}` dans la $\langle clef \rangle$. L’idée est que lorsque l’on développe la $\langle clef \rangle$, c’est la valeur de $\langle autre\ clef \rangle$ qui est développée à sa place. Cela correspond vaguement à la notion de lien symbolique dans Unix, d’où le nom.

9. NdTdS : ou encore, p. ex., `\pgfkeysvalueof` comme vu page 5.

Le prochain manipulateur est utile pour la situation fréquente où $\langle clef \rangle = \langle valeur \rangle$ devrait entreposer la $\langle valeur \rangle$ dans une macro. On notera que on pourrait tout aussi bien entreposer la $\langle valeur \rangle$ dans la $\langle clef \rangle$ elle-même.

Manipulateur $\langle clef \rangle /.store in = \langle macro \rangle$

L'effet de ce manipulateur est le suivant : quand on écrit $\langle clef \rangle = \langle valeur \rangle$, le code `\def\langle macro \rangle{\langle valeur \rangle}` est exécuté et la valeur fournie est « entreposée » dans la $\langle macro \rangle$.

```
Salut Gruffalo! \pgfkeys{/text/.store in=\montexte}
                 \def\a{monde}
                 \pgfkeys{/text=Salut \a!}
                 \def\a{Gruffalo}
                 \montexte
```

Manipulateur $\langle clef \rangle /.estore in = \langle macro \rangle$

Ce manipulateur est semblable à `/.store in` sauf que c'est le code `\edef\langle macro \rangle{\langle valeur \rangle}` qui est exécuté. C'est donc la version développée de $\langle valeur \rangle$ qui est entreposée dans $\langle macro \rangle$.

```
Salut monde! \pgfkeys{/text/.estore in=\montexte}
              \def\a{monde}
              \pgfkeys{/text=Salut \a!}
              \def\a{Gruffalo}
              \montexte
```

Dans une autre situation courante la clé est utilisée pour régler un si TeXien — un `\if\langle QuelqueChose \rangle` — sur vrai ou faux.

Manipulateur $\langle clef \rangle /.is if = \langle nom de si \rangle$

L'effet de ce manipulateur est le suivant : quand on écrit $\langle clef \rangle = \langle valeur \rangle$, il vérifie d'abord que $\langle valeur \rangle$ est `true` ou `false` — la valeur par défaut, si on ne fournit pas de valeur, est `true`. Si ce n'est pas le cas, la clé d'erreur `/errors/boolean expected` est exécutée. Dans le cas contraire, le code `\langle nom de si \rangle \langle valeur \rangle` est exécuté, ce qui règle le si TeXien comme attendu.

On notera qu'il faut définir le `si` en dehors d'un appel à `\pgfkeys` comme le montre l'exemple suivant.

```
Ronde? \newif\iftheterreestplate
        \pgfkeys{/terre plate/.is if=theterreestplate}
        \pgfkeys{/terre plate=false}
        \iftheterreestplate
            Plate
        \else
            Ronde?
        \fi
```

Le manipulateur suivant s'occupe des situations où seul un petit ensemble de $\langle valeur \rangle$ s est utilisable avec $\langle clef \rangle = \langle valeur \rangle$. P. ex., dans TikZ, l'extrémité d'un segment ne peut être que `rounded`, `rectangle` ou `butt` et rien d'autre. Dans cette situation, le manipulateur suivant s'avère utile.

Manipulateur $\langle clef \rangle /.is choice$

Ce manipulateur règle les choses de telle sorte que, quand on donne $\langle clef \rangle = \langle valeur \rangle$, la sous-clé $\langle clef \rangle / \langle valeur \rangle$ est exécutée. Il faut donc que les différents choix soient donnés comme des sous-clés de $\langle clef \rangle$.

```
\pgfkeys{/line cap/.is choice}
\pgfkeys{/line cap/round/.style={\pgfsetbuttcap}}
\pgfkeys{/line cap/butt/.style={\pgfsetroundcap}}
\pgfkeys{/line cap/rect/.style={\pgfsetrectcap}}
\pgfkeys{/line cap/rectangle/.style={/line cap=rect}}
...
\draw [/line cap=butt] ...
```

Si la sous-clé $\langle clef \rangle / \langle valeur \rangle$ est inconnue, la clé d'erreur `/errors/unknown choice value` est exécutée.

4.6 Valeurs développées, valeurs multiples

Quand on écrit $\langle clef \rangle = \langle valeur \rangle$, on veut généralement utiliser la $\langle valeur \rangle$ « telle quelle ». De fait, nous avons fait extrêmement attention à ce que l'on puisse même écrire des choses comme #1 ou des *si* T_EXiens incomplets dans $\langle valeur \rangle$. Toutefois, il arrive parfois que l'on veuille que la $\langle valeur \rangle$ soit développée avant qu'elle soit utilisée. P. ex., $\langle valeur \rangle$ pourrait être le nom d'une macro comme `\mamacro` et on voudrait non pas employer `\mamacro` comme macro mais plutôt employer son *contenu*. Au lieu, donc, d'utiliser $\langle valeur \rangle$, on voudrait utiliser ce en quoi $\langle valeur \rangle$ se développe. Plutôt que des trucs à base de `\expandafter`, on emploiera les manipulateurs suivants :

Manipulateur $\langle clef \rangle /.expand\ once = \langle valeur \rangle$

Ce manipulateur développe $\langle valeur \rangle$ une fois — plus précisément, il exécute la commande `\expandafter` sur le premier lexème de $\langle valeur \rangle$ puis traite le $\langle résultat \rangle$ comme si on avait écrit $\langle clef \rangle = \langle résultat \rangle$. On notera que si la $\langle clef \rangle$ contient elle-même un manipulateur, ce dernier sera appelé normalement.

Clef 1 : \c	<code>\def\c{le fond}</code>
Clef 2 : \b	<code>\def\b{\a}</code>
Clef 3 : \a	<code>\def\c{\b}</code>
Clef 4 : le fond	<code>\pgfkeys{/clef1/.initial=\c}</code>
	<code>\pgfkeys{/clef2/.initial/.expand once=\c}</code>
	<code>\pgfkeys{/clef3/.initial/.expand twice=\c}</code>
	<code>\pgfkeys{/clef4/.initial/.expanded=\c}</code>
	 <code>\def\c{{\ttfamily}\string\c}</code>
	<code>\def\c{{\ttfamily}\string\c}}</code>
	 <code>\begin{tabular}{ll}</code>
	<code>Clef 1:& \pgfkeys{/clef1} \\\</code>
	<code>Clef 2:& \pgfkeys{/clef2} \\\</code>
	<code>Clef 3:& \pgfkeys{/clef3} \\\</code>
	<code>Clef 4:& \pgfkeys{/clef4}</code>
	<code>\end{tabular}</code>

Manipulateur $\langle clef \rangle /.expand\ twice = \langle valeur \rangle$

Ce manipulateur fonctionne comme si on écrivait $\langle clef \rangle /.expand\ once /.expand\ once = \langle valeur \rangle$.

Manipulateur $\langle clef \rangle /.expanded = \langle valeur \rangle$

Ce manipulateur développe complètement $\langle valeur \rangle$ — à l'aide d'un `\edef` — avant de traiter $\langle clef \rangle = \langle résultat \rangle$.

Manipulateur $\langle clef \rangle /.list = \langle vliste\ de\ valeurs \rangle$

Dans la $\langle vliste\ de\ valeurs \rangle$, les valeurs sont séparées par des virgules. Ce manipulateur fait que la $\langle clef \rangle$ est traitée plusieurs fois, une fois par valeur de la liste. Il faudra, bien entendu, placer la liste de valeurs entre accolades sinon T_EX n'arrivera pas à décider si une virgule est suivie d'une nouvelle clé ou d'une nouvelle valeur.

La $\langle vliste\ de\ valeurs \rangle$ est traitée à l'aide de `\foreach` et on peut donc utiliser la notation ... à l'intérieur.

(a)(b)(0)(1)(2)(3)(4)(5)	<code>\pgfkeys{/foo/.code={#1}}</code>
	<code>\pgfkeys{/foo/.list={a,b,0,1,...,5}}</code>

4.7 Manipulateurs de transmission

J'utilise « transmission » comme traduction de *forwarding*. On pensera à ce que l'on fait quand on "forwarde" un message.

Manipulateur $\langle clef \rangle /.forward\ to = \langle autre\ clef \rangle$

Ce manipulateur fait que $\langle clef \rangle$ transmet son argument à l' $\langle autre\ clef \rangle$. Quand on utilise la $\langle clef \rangle$, son code normal est effectué d'abord puis la valeur est passée à $\langle autre\ clef \rangle$. Si la $\langle clef \rangle$ n'a pas été définie au préalable, elle sera définie alors — et ne fera rien par elle-même si ce n'est transmettre la valeur à $\langle autre\ clef \rangle$. $\langle autre\ clef \rangle$ doit être un nom complet de clé c.-à-d. avec le chemin complet.

```
(b : 1)(a : 1) (a : 2) \pgfkeys{
/a/.code=(a: #1),
/b/.code=(b: #1),
/b/.forward to=/a,
/c/.forward to=/a
}
\pgfkeys{/b=1} \pgfkeys{/c=2}
```

Manipulateur $\langle clef \rangle /.search\ also=\{\langle liste\ de\ chemins \rangle\}$

Un style qui installe un manipulateur `/.unknown` dans la $\langle clef \rangle$. Ce manipulateur `/.unknown` cherchera les clés inconnues dans chacun des chemins fournis dans la $\langle liste\ de\ chemins \rangle$.

Appelle `/chemin secondaire/option` avec ‘valeur’

```
% [1]
\pgfkeys{/chemin secondaire/option/.code={Appelle /chemin secondaire/option avec '#1'}}
% [2]
\pgfkeys{/chemin principal/.search also={/chemin secondaire}}
% [3]
\pgfkeys{/chemin principal/.cd,option=valeur}
```

Avec [1] on définit une clé — à savoir `option` dont le nom complet est `/chemin secondaire/option` — puis, en [2], on définit un chemin de recherche — qui est `/chemin secondaire` — comme valeur de `/.search also`. Enfin, en [3], une fois revenu dans le chemin principal grâce à `/.cd`, on passe la valeur `valeur` à la clé `option`. Comme cette clé n’est pas dans le chemin principal, le mécanisme de recherche trouvera `/chemin secondaire/option`.

Le manipulateur `/.search also` utilise la stratégie suivante

1. si un utilisateur fournit une clé complète qui ne peut être trouvée, p. ex. la chaîne `/chemin principal/option`, il suppose que l’utilisateur sait ce qu’il fait et *ne* continue *pas* à chercher `option` dans la $\langle liste\ de\ chemins \rangle$;
2. si un utilisateur ne fournit que le nom de la clé, p. ex. `option` et que `option` est introuvable dans le chemin par défaut — qui se trouve être `/chemin principal` dans l’exemple précédent —, l’élément suivant de la $\langle liste\ de\ chemins \rangle$ — il s’agit de `/chemin secondaire` dans l’exemple — devient le chemin par défaut et `\pgfkeys` redémarre.

Cette opération sera répétée jusqu’à ce que la clé soit trouvée ou que tous les éléments de $\langle liste\ de\ chemins \rangle$ aient été essayés.

3. Si tous les éléments de $\langle liste\ de\ chemins \rangle$ ont été essayés et que la clé est toujours inconnue, le manipulateur de secours `/handlers/.unknown` est invoqué.

Appelle `/chemin secondaire/option` avec ‘valeur’

```
% [1]
\pgfkeys{/chemin secondaire/option/.code={Appelle /chemin secondaire/option avec '#1'}}
% [2]
\pgfkeys{/chemin principal/.search also={/chemin secondaire}}
% [3]
\pgfkeys{/chemin principal/.cd,option=valeur}

% [4] !
\pgfkeys{/handlers/.unknown/.code={Vu option inconnue \pgfkeyscurrentkeyRAW=#1!}}%
\pgfkeys{/chemin principal/.cd,/chemin principal/option=valeur}
```

En [4], on a un résultat négatif. La clé `option` est donnée avec son nom complet `/chemin principal/option` : elle ne sera pas trouvée et le mécanisme de `/.see also` ne sera pas utilisé.

On notera que la stratégie de `/.search also` est différente de celle exposée dans le premier exemple de 3.6 car `/.search also` ne s’applique qu’aux clés dont le nom n’est pas complet.

Pour la lectrice familière de `\pgfkeys`, le code véritable de `/.search also` peut être intéressant :

1. `\pgfkeys{/path/.search also = {/tikz}}` équivaut à

```

\pgfkeys{/path/.unknown/.code={%
  \ifpgfkeysaddeddefaultpath
  % ne traite que les clefs dont le nom est incomplet:
  \pgfkeysussuccessfalse
  \let\pgfkeys@searchalso@name=\pgfkeyscurrentkeyRAW
  \ifpgfkeysussuccess
  \else
  % cherche en prenant /tikz pour chemin
  \pgfkeys{/tikz}{\pgfkeys@searchalso@name={#1}}%
  \fi
  \else
  \def\pgfutilnext{\pgfkeysvalueof{/handlers/.unknown/.@cmd}#1\pgfeov}%
  \pgfutilnext
  \fi
}
}

```

2. `\pgfkeys{/path/.search also = {/tikz,/pgf}}` équivaut à

```

\pgfkeys{/path/.unknown/.code={%
  \ifpgfkeysaddeddefaultpath
  \pgfkeysussuccessfalse
  \let\pgfkeys@searchalso@name=\pgfkeyscurrentkeyRAW
  \ifpgfkeysussuccess
  \else
  % step 1: search in /tikz with .try:
  \pgfkeys{/tikz}{\pgfkeys@searchalso@name/.try={#1}}%
  \fi
  \ifpgfkeysussuccess
  \else
  % step 2: search in /pgf (without .try!):
  \pgfkeys{/pgf}{\pgfkeys@searchalso@name={#1}}%
  \fi
  \else
  \def\pgfutilnext{\pgfkeysvalueof{/handlers/.unknown/.@cmd}#1\pgfeov}%
  \pgfutilnext
  \fi
}
}

```

Pour autoriser également la recherche des styles — et autres clés manipulées —, on peut changer la configuration des manipulateurs à l'aide de `/handler config = full or existing` lors de l'emploi de `/.search also` c.-à-d.

```

\pgfkeys{
  /main path/.search also={/secondary path},
  /handler config=full or existing}

```

4.8 Manipulateurs d'essais

Manipulateur `<clef>/ .try=<valeur>`

Try signifie « essayer ». Ce manipulateur a le même effet que si `<clef> = <valeur>` avait été utilisé. Toutefois, si ni `<clef>/ .@cmd` ni la `<clef>` elle-même ne sont définies, aucun manipulateur n'est appelé. L'exécution de la clé s'arrête tout simplement. Ainsi, ce manipulateur essaiera simplement d'utiliser la clé mais aucune autre action ne sera entreprise si la clé n'est pas définie.

Le test TeXien `\ifpgfkeysussuccess` est réglé suivant que la `<clef>` a été exécutée avec succès ou non.

```

(a :hallo)(b :welt) \pgfkeys{/a/.code=(a:#1)}
                   \pgfkeys{/b/.code=(b:#1)}
                   \pgfkeys{/x/.try=hmm,/a/.try=hallo,/b/.try=welt}

```

Manipulateur `<clef>/ .retry=<valeur>`

Ce manipulateur fonctionne comme `/.try` mais il ne fera rien si `\ifpgfkeysussuccess` est vrai c.-à-d. qu'il n'essaiera de régler une clé que si le dernier essai a échoué.

```

(a :hallo) \pgfkeys{/a/.code=(a:#1)}
           \pgfkeys{/b/.code=(b:#1)}
           \pgfkeys{/x/.try=hmm,/a/.retry=hallo,/b/.retry=welt}

```

Manipulateur `<clef>/.lastretry=<valeur>`

Ce manipulateur fonctionne comme `/.retry` sauf qu'il invoquera les manipulateurs habituels de clés inconnues si `\ifpgfkeyssuccess` est faux. Ce manipulateur n'essaiera de régler la clé que si la dernière tentative a échoué et, dans ce cas, ce sera le dernier essai.

4.9 Manipulateurs d'inspection

Manipulateur `<clef>/.show value`

Ce manipulateur exécute une commande `\show` sur la valeur entreposée dans `<clef>`. C'est intéressant surtout pour du débogage.

Exemple: `\pgfkeys{/my/obscure key/.show value}`

Manipulateur `<clef>/.show code`

Ce manipulateur exécute une commande `\show` sur le code entreposé dans `<clef>/.@cmd`. C'est intéressant surtout pour du débogage.

Exemple: `\pgfkeys{/my/obscure key/.show code}`

Ce qui suit n'est pas un manipulateur mais on l'utilise communément pour inspecter des choses :

`/utils/exec=<code>` (sans défaut)

Cette clé exécutera simplement le `<code>` fourni.

Exemple: `\pgfkeys{some key=some value,/utils/exec=\show\hallo,obscure key=obscure}`

5 Clés d'erreur

Dans certaines situations une erreur peut apparaître, comme quand on utilise une clé indéfinie. Dans ces situations, une clé d'erreur est exécutée. Ces clés doivent contenir une macro prenant deux arguments. Le premier est la clé offensante — éventuellement après développement de macros —, le second est la valeur passée comme paramètre — là encore après un éventuel développement de macros.

Pour l'instant, les clés d'erreur sont simplement exécutées. Il serait peut-être bon qu'à l'avenir on ait différentes sous-clés exécutées en fonction de la langue en vigueur afin que l'utilisateur reçoive un message d'erreur localisé.

`/errors/value required={<clef offensante>}{<valeur>}` (sans défaut)

Cette clé — « valeur requise » — est exécutée à chaque fois qu'une `<clef offensante>` est appelée sans que l'on fournisse de valeur et qu'une valeur est requise.

`/errors/value forbidden={<clef offensante>}{<valeur>}` (sans défaut)

Cette clé — « valeur interdite » — est exécutée à chaque fois qu'une `<clef offensante>` est appelée avec une valeur alors qu'une valeur est interdite.

`/errors/boolean expected={<clef offensante>}{<valeur>}` (sans défaut)

Cette clé — « booléen attendu » — est exécutée à chaque fois qu'une `<clef>` définie avec `/.is if` est appelée avec une `<valeur>` qui n'est ni `true` ni `false`.

`/errors/unknown choice value={<clef offensante>}{<valeur>}` (sans défaut)

Cette clé — « valeur du choix inconnue » — est exécutée à chaque fois qu'une `<clef>` définie avec `/.is choice` est appelée avec une `<valeur>` pour laquelle aucun manipulateur n'est définie.

`/errors/unknown key={<clef offensante>}{<valeur>}` (sans défaut)

Cette clé — « clé inconnue » — est exécutée à chaque fois qu'une clé est inconnue et qu'on ne trouve pas de manipulateur `/.unknown` spécifique.

6 Filtrage de clés

Un complément dû à Christian FEUERSÄNGER

Normalement, un appel à `\pgfkeys` règle toutes les clés de la liste passée en argument. C'est, en général, ce qu'attend l'utilisatrice. Toutefois, l'implémentation de différentes extensions ou de bibliothèques de macros

pour PGF peut demander plus de contrôle sur la procédure de réglage des clés : une bibliothèque A peut vouloir régler ses options directement et déléguer les options restantes à une bibliothèque B.

Cette section décrit les méthodes de filtrage de clés de PGF, y compris les options de groupements de familles. Si on ne veut qu'utiliser PGF — ou ses bibliothèques — on peut passer cette section ; elle s'adresse aux auteurs d'extensions — ou de bibliothèques.

6.1 Exemple préliminaire

Les utilisateurs de `xkeyval` connaissent bien le concept de famille de clés : les clés appartiennent à un groupe et ces clés peuvent être exclues de certaines options. PGF permet de grouper des clés par familles et un mécanisme plus abstrait de sélection avec `\pgfkeysfiltered`, une variante de `\pgfkeys`. Supposons que nous ayons le groupement suivant :

```
\pgfkeys{
  /my group/A1/.code=(A1: #1),
  /my group/A2/.code=(A2: #1),
  /my group/A3/.code=(A3: #1),
  /my group/B/.code=(B: #1),
  /my group/C/.code=(B: #1),
}
```

et que nous voulions ne régler que les options A1, A2 et A3. Un appel de `\pgfkeys` produit alors ce qui suit

```
(A1 : a1)(A2 : a2)(B : b)(B : c)
```

```
\pgfkeys{/my group/A1=a1, /my group/A2=a2, /my group/B=b, /my group/C=c}
```

parce que toutes ces options de commande sont traitées consécutivement.

Définissons maintenant une famille, nommée A, contenant A1, A2 et A3 et ne réglons que les membres de cette famille. Nous préparons le réglage des clés avec :

```
\pgfkeys{
  /my group/A/.is family,
  /my group/A1/.belongs to family=/my group/A,
  /my group/A2/.belongs to family=/my group/A,
  /my group/A3/.belongs to family=/my group/A,
}
```

et

```
\pgfkeys{/pgf/key filters/active families/.install key filter}
```

Après ces préparatifs, nous pouvons utiliser

```
(A1 : a1)(A2 : a2) \pgfkeys{/my group/A/.activate family}
\pgfkeysfiltered{/my group/A1=a1, /my group/A2=a2,
/my group/B=b, /my group/C=c}
```

ou

```
(A1 : a1)(A2 : a2)(A3 : a3)
```

```
\pgfkeys{/my group/A/.activate family}
\pgfkeysfiltered{/my group/A1=a1, /my group/A2=a2,
/my group/B=b, /my group/C=c, /tikz/color=blue, /my group/A3=a3}
```

pour régler uniquement les clés qui appartiennent à la famille *active* — dans notre cas, seule la famille A était active aussi les autres clés n'ont pas été traitées. Le traitement par famille est vraiment rapide et permet d'activer un nombre quelconque de familles de clés.

Les options non traitées peuvent être rassemblées dans une macro — semblable à `\xkv@rm` de `xkeyval` —, rejetées ou traitées à la main. La section suivante décrit les détails de la sélection de clés et de la déclaration de familles.

6.2 Définition des filtres

La commande `\pgfkeysfiltered` est l'outil principal pour ne traiter que les options choisies. Il fonctionne comme suit.

`\pgfkeysfiltered{⟨key-value-list⟩}`

Traite les options exactement comme `\pgfkeys {⟨liste de paires⟩}` mais un filtre de clé est examiné dès que l'identification de la clé est complète.

Le filtre de clés dit à `\pgfkeysfiltered` s'il doit continuer à traiter l'option en cours — la valeur de retour est « vrai » — ou s'il faut faire autre chose — le filtre retourne « faux » .

Il n'y a qu'un seul filtre de clés actif et il est mis en place par le manipulateur `/.install key filter` ou par `\pgfkeyinstallfilter`.

Si le filtre de clés retourne « faux », un unique manipulateur de filtre de clés prend le contrôle. Ce manipulateur est mis en place par la méthode `/.install key filter handler` et accède au nom complet de la clé, la valeur et — éventuellement — le chemin.

Le filtrage de clés s'applique à tout appel — éventuellement imbriqué — de `\pgfkeys`, `\pgfkeysalso`, `\pgfqkeys` et `\pgfqkeysalso` pendant l'évaluation de la *⟨liste de paires⟩*. Il ne s'applique *pas* aux routines comme `\pgfkeyssetvalue` ou `\pgfkeysgetvalue`. De plus, les clés appartenant à `/errors` sont toujours traitées. Les routines de filtrage de clés ne peuvent pas être imbriquées : on ne peut pas combiner automatiquement différents filtres.

`\pgfqkeysfiltered{⟨default-path⟩}{⟨key-value-list⟩}`

Variante de `\pgfkeysfiltered` utilisant le réglage de recherche « rapide » de chemin. C'est la variante de type `\pgfqkeys` de `\pgfkeysfiltered`, voyez page 6 pour plus de détails.

`\pgfkeysalsofrom{⟨macro⟩}`

Variante de `\pgfkeysalso` qui charge une liste de paires depuis *⟨macro⟩*.

Cette commande est utile conjointement au manipulateur `/pgf/key filter handlers/append filtered to = ⟨macro⟩`.

L'exemple suivant utilise les mêmes réglages que dans 6.1, page 23.

```
(A1 : a1)(A2 : a2)(A3 : a3)Remaining : '/my group/B=b,/my group/C=c,/tikz/color=blue'.(B : b)(B : c)
```

```
\pgfkeys{/pgf/key filter handlers/append filtered to/.install key filter handler=\remainingoptions}
\def\remainingoptions{}
\pgfkeysfiltered{/my group/A1=a1, /my group/A2=a2,
/my group/B=b, /my group/C=c, /tikz/color=blue, /my group/A3=a3}

Remaining: '\remainingoptions'.
\pgfkeysalsofrom{\remainingoptions}
```

`\pgfkeysalsofiltered{⟨key-value-list⟩}`

Cette commande fonctionne comme `\pgfkeysfiltered` mais ne change pas le chemin par défaut, voir la documentation sur `\pgfkeysalso`, page 6, pour plus de détails.

`\pgfkeysalsofilteredfrom{⟨macro⟩}`

Variante de `\pgfkeysalsofiltered` qui charge une liste de paires depuis *⟨macro⟩*.

Manipulateur *⟨clef⟩* `/.install key filter=⟨optional arguments⟩`

Ce manipulateur met en place un filtre de clés. Un tel filtre est une clé de commande — c.-à-d. une clé pour laquelle le suffixe `/.@cmd` existe — qui règle le booléen TeXien `\ifpgfkeysfiltercontinue`. Voici un exemple simple qui définit un filtre retournant toujours « vrai » :

```
\pgfkeys{/foo/bar/true key filter/.code={\pgfkeysfiltercontinuetrue}}
\pgfkeys{/foo/bar/true key filter/.install key filter}
```

Si un filtre exige des arguments, ils sont également mis en place par `/.install key filter`. Un exemple — en reprenant encore les réglages de l'exemple de la page 23 — en est le manipulateur `/.pgf/key filters>equals` :

```
(A1 : a1) \pgfkeys{/pgf/key filters>equals/.install key filter={/my group/A1}}
\pgfkeysfiltered{/my group/A1=a1, /my group/A2=a2,
/my group/B=b, /my group/C=c, /tikz/color=blue, /my group/A3=a3}
```

Si la clé requiert plus d'un argument, il faut fournir la liste complète d'arguments entre accolades comme ceci : `{⟨premier⟩}{⟨second⟩}`.

On peut employer `\pgfkeysinstallkeyfilter` \langle *clef complète* \rangle \langle *arguments optionnels* \rangle pour obtenir le même effet.

Se reporter à la section 6.7 pour savoir comment écrire des filtres de clés.

Manipulateur \langle *clef* \rangle `/.install key filter handler=` \langle *optional arguments* \rangle

Ce manipulateur installe la routine qui sera invoquée pour chaque option *non traitée* c.-à-d. chaque option pour laquelle le filtre de clé retourne `false`.

Le `/.install key filter handler` est utilisé de la même manière que `/.install key filter`. Il en existe une version sous forme de macro `\pgfkeysinstallkeyfilterhandler` $\{\langle$ *clef complète* $\rangle\}$ $\{\langle$ *arguments optionnels* $\rangle\}$ qui produit le même résultat. Se reporter à la section 6.7 pour savoir comment écrire des filtres de clés.

6.3 Manipulateurs de clés non-traitées

Chaque option que le filtre de clés a décidé d'ignorer est traitée par un « manipulateur de filtre de clés ». Cette extension définit plusieurs manipulateurs de filtre de clés.

`/pgf/key filter handlers/append filtered to=` $\{\langle$ *macro* $\rangle\}$ (sans défaut)

On met en place ce manipulateur pour que les options non-traitées soient ajoutées à la $\{\langle$ *macro* $\rangle\}$.

(A1 : a1)(A2 : a2)Remaining options : '/my group/B=b,/my group/C=c,/tikz/color=blue'.

```
\pgfkeys{/pgf/key filter handlers/append filtered to/.install key filter handler=\remainingoptions}
\def\remainingoptions{}
\pgfkeysfiltered{/my group/A1=a1, /my group/A2=a2,
/my group/B=b, /my group/C=c, /tikz/color=blue}
Remaining options: '\remainingoptions'.
```

Cet exemple utilise les clés telles que définies dans la section d'introduction 6.1.

`/pgf/key filter handlers/ignore` (sans valeur)

On met en place ce manipulateur pour que les options non-traitées soient simplement ignorées. C'est le manipulateur par défaut.

`/pgf/key filter handlers/log` (sans valeur)

On met en place ce manipulateur pour qu'une mention soit écrite dans le fichier de `log` — et sur le terminal — pour chaque option non-traitée.

6.4 Gestion des familles

PGF s'appuie aussi sur le concept de **famille** (*family*) : chaque option peut être rattachée à — au plus — une famille. Les familles sont des groupes vagues de clés, indépendants de la hiérarchie des clés. P. ex., `/mon arbre/clef` peut appartenir à la famille `/tikz`.

On peut « activer » ou « désactiver » une famille et, de plus, ne régler que les clés qui appartiennent à une famille active à l'aide de manipulateurs idoines.

La gestion des familles est rapide : s'il y a N options dans une liste de paires et K familles actives, le temps d'exécution de `\pgfkeysfiltered` est un $O(N+K)$ — activation de chaque famille $O(K)$, vérification de chaque option $O(N)$, désactivation de chaque famille $O(K)$.

Manipulateur \langle *clef* \rangle `/.is family`

Définit une nouvelle famille. Cette option est décrite dans 4.1, page 13.

Manipulateur \langle *clef* \rangle `/.activate family`

Active une famille. Il faut que la famille ait été définie sinon l'erreur `/errors/family unknown` est produite.

L'activation revient à mettre sur « vrai » un booléen T_EXien qui indique que la famille doit être traitée.

On peut utiliser `\pgfkeysactivatefamily` $\{\langle$ *clef complète* $\rangle\}$ pour obtenir la même chose. De plus, on peut activer plusieurs familles avec `\pgfkeysactivatefamilies` $\{\langle$ *liste de familles* $\rangle\}$ $\{\langle$ *nom de macro pour la désactivation* $\rangle\}$ — voir 6.6, page 28.

Manipulateur `<clef>/deactivate family`

Désactive une famille. Il faut que la famille ait été définie sinon l'erreur `/errors/family unknown` est produite.

On peut utiliser `\pgfkeysdeactivatefamily{<clef complète>}` pour obtenir la même chose.

Manipulateur `<clef>/belongs to family={<nom de famille>}`

Associe l'option en cours avec la famille nommée `<nom de famille>`, nom qui doit être un chemin complet.

```
\pgfkeys{/foo/bar/.is family}
\pgfkeys{
  /foo/a/.belongs to family=/foo/bar,
  /foo/b/.belongs to family=/foo/bar
}
```

Chaque option ne peut avoir qu'au plus une famille, `/.belongs to family` écrase les réglages précédents.

`/pgf/key filters/active families` (sans valeur)

On met ce filtre en place si l'on veut que `\pgfkeysfiltered` ne traite que les familles actives. Si une clé n'appartient à aucune famille, elle n'est pas traitée. Si une clé est complètement inconnue dans le chemin par défaut, le manipulateur normal de clé inconnue lié à `\pgfkeys` est invoqué.

`/pgf/key filters/active families or no family={<filtre 1>}{<filtre 2>}` (sans défaut)

Ce filtre de clés configure `\pgfkeysfiltered` pour qu'il fonctionne comme suit :

1. si la clé en cours appartient à une famille, `\ifpgfkeysfiltercontinue` est réglé sur « vrai » si, et seulement si, la famille est active ;
2. si la clé en cours n'appartient à aucune famille, `\ifpgfkeysfiltercontinue` est réglé sur le résultat de `<filtre 1>` ;
3. si la clé en cours est inconnue, `\ifpgfkeysfiltercontinue` est réglé sur le résultat de `<filtre 2>`.

Les arguments `<filtre 1>` et `<filtre 2>` sont d'autres filtres de clés — éventuellement avec options — et permettent un contrôle plus fin du procédé de filtrage.

```
\pgfkeysinstallkeyfilter
  {/pgf/key filters/active families or no family}
  {{/pgf/key filters/is descendant of=/tikz}% pour les clefs sans famille
  {/pgf/key filters/false}% pour les clefs inconnues
  }%
```

Ce filtre retournera `vrai` pour chaque option dont la famille est active. Si une option est sans famille, la valeur de retour sera `vrai` si et seulement si cette option appartient à `/tikz`. Si l'option est inconnue, la valeur de retour sera `faux` et aucun manipulateur d'inconnu ne sera appelé.

`/pgf/key filters/active families or no family DEBUG={<filtre 1>}{<filtre 2>}` (sans défaut)

Variante de `active families or no family` qui trace chaque action sur le terminal et le fichier `log`.

`/pgf/key filters/active families and known` (sans valeur)

Un alias rapide pour

```
/pgf/key filters/active families or no family=
  {/pgf/keys filters/false}
  {/pgf/keys filters/false}.
```

`/pgf/key filters/active families or descendants of={<prefixe de chemin>}` (sans défaut)

Un alias rapide pour

```
/pgf/key filters/active families or no family=
  {/pgf/keys filters/is descendant of={<prefixe de chemin>}}
  {/pgf/keys filters/false}.
```

`\pgfkeysactivatefamiliesandfilteroptions{<liste de familles>}{<liste de paires>}`

Un simple raccourci qui active chaque famille de `<liste de familles>`, invoque `\pgfkeysfiltered{<liste de paires>}` puis désactive les familles.

On notera qu'il faut avoir mis en place un filtre de famille sinon l'activation de la famille n'aura pas d'effet.

`\pgfqkeysactivatefamiliesandfilteroptions`{*{liste de familles}*}{*{chemin par défaut}*}{*{liste de paires}*}

Variante rapide avec chemin par défaut de `\pgfqkeysactivatefamiliesandfilteroptions`.

`\pgfkeysactivatesinglefamilyandfilteroptions`{*{nom de famille}*}{*{liste de paires}*}

Un simple raccourci qui active une seule famille, invoque `\pgfkeysfiltered`{*{liste de paires}*} puis désactive la famille.

On notera qu'il faut avoir mis en place un filtre sur cette famille sinon l'activation de la famille n'aura pas d'effet.

`\pgfqkeysactivatesinglefamilyandfilteroptions`{*{nom de famille}*}{*{chemin par défaut}*}{*{liste de paires}*}

Variante rapide avec chemin par défaut de `\pgfkeysactivatesinglefamilyandfilteroptions`.

6.5 Autres filtres de clés

Il existe encore quelques filtres qui n'ont rien à voir avec les familles.

`/pgf/key filters/is descendant of`={*{chemin}*} (sans défaut)

On met en place ce filtre pour ne traiter que les clés appartenant au *{chemin}*. Il retourne `vrai` pour les clés de *{chemin}* et, aussi, pour les clés inconnues c.-à-d. qu'elles sont traitées par les manipulateurs d'erreurs standards de PGF.

```
(A : a)(B : b) \pgfkeys{
  /group 1/A/.code={A: #1},
  /group 1/foo/bar/B/.code={B: #1},
  /group 2/C/.code={C: #1},
  /pgf/key filters/is descendant of/.install key filter=/group 1}
\pgfkeysfiltered{/group 1/A=a,/group 1/foo/bar/B=b,/group 2/C=c}
```

`/pgf/key filters>equals`={*{clef complete}*} (sans défaut)

On met en place ce filtre pour ne traiter que la *{clef complete}*. Le filtre retourne `vrai` si la clé est inconnue ou égale à *{clef complete}*.

```
(A : a) \pgfkeys{
  /group 1/A/.code={A: #1},
  /group 1/B/.code={B: #1},
  /pgf/key filters>equals/.install key filter=/group 1/A}
\pgfqkeysfiltered{/group 1}{A=a,B=b}
```

`/pgf/key filters/not`={*{filtre}*} (sans défaut)

Ce filtre inverse la valeur logique de *{filtre}*.

```
(C : c) \pgfkeys{
  /group 1/A/.code={A: #1},
  /group 1/foo/bar/B/.code={B: #1},
  /group 2/C/.code={C: #1},
  /pgf/key filters/not/.install key filter=
    {/pgf/key filters/is descendant of=/group 1}
\pgfkeysfiltered{/group 1/A=a,/group 1/foo/bar/B=b,/group 2/C=c}
```

On notera que les clés inconnues seront traitées par les manipulateurs habituels.

`/pgf/key filters/and`={*{filtre 1}*}{*{filtre 2}*} (sans défaut)

Ce filtre ne retourne `vrai` que si les deux filtres retournent `vrai`.

`/pgf/key filters/or`={*{filtre 1}*}{*{filtre 2}*} (sans défaut)

Ce filtre retourne `vrai` dès que l'un des deux filtres retourne `vrai`.

`/pgf/key filters/true` (sans valeur)

Ce filtre retourne toujours `vrai`.

`/pgf/key filters/false` (sans valeur)

Ce filtre retourne toujours faux même pour des clés inconnues.

`/pgf/key filters/defined` (sans valeur)

Ce filtre retourne faux si la clé en cours est inconnue, ce qui évite d'appeler un manipulateur de clé inconnue.

6.6 Interface de programmation

`\pgfkeysinterruptkeyfilter`

<environment contents>

`\endpgfkeysinterruptkeyfilter`

Désactive temporairement le filtrage de clés et n'a aucun effet si aucun filtrage n'est actif.

On notera que cela ne produit aucun groupe au sens de T_EX.

`\pgfkeyssavekeyfilterstateto{<macro>}`

Crée la *<macro>* qui contient les commandes pour réactiver le filtrage et le manipulateur de filtre courants. On peut s'en servir pour met en place temporairement le filtrage.

`\pgfkeysinstallkeyfilter{<full key>}{<optional arguments>}`

La commande `\pgfkeysinstallkeyfilter{<full key>}{<optional arguments>}` a le même effet que `\pgfkeys{<full key>/install key filter={<optional arguments>}}`.

`\pgfkeysinstallkeyfilterhandler{<full key>}{<optional arguments>}`

La commande `\pgfkeysinstallkeyfilterhandler{<full key>}{<optional arguments>}` a le même effet que `\pgfkeys{<full key>/install key filter handler={<optional arguments>}}`.

`\pgfkeysactivatefamily{<nom de famille>}`

Équivaut à `\pgfkeys{<nom de famille>/activate family}`.

`\pgfkeysdeactivatefamily{<nom de famille>}`

Équivaut à `\pgfkeys{<nom de famille>/deactivate family}`.

`\pgfkeysactivatefamilies{<liste de familles>}{<macro>}`

Active toutes les familles de la *<liste de familles>* et crée la *<macro>* pour les désactiver.

```
\pgfkeysactivatefamilies{/family 1./family 2./family 3}{\deactivatename}
\pgfkeysfiltered{foo,bar}
\deactivatename
```

`\pgfkeysiffamilydefined{<famille>}{<sivrai>}{<sifaux>}`

Regarde si la clé complète *<famille>* est bien une famille puis exécute *<sivrai>* ou *<sifaux>* suivant les cas.

`\pgfkeysisfamilyactive{<famille>}`

Règle le booléen T_EXien `\ifpgfkeysfiltercontinue` suivant que *<famille>* est active ou pas.

`\pgfkeysgetfamily{<clef>}{<macro>}`

Place dans la *<macro>* la famille à laquelle est attachée la *<clef>* complète.

`\pgfkeyssetfamily{<clef>}{<famille>}`

La commande `\pgfkeyssetfamily{<clef complete>}{<famille>}` a le même effet que `\pgfkeys{<clef complete>/belongs to family={<famille>}}`.

6.7 Définir ses propres filtres et manipulateurs de filtres

Le code du filtre sera invoqué pendant l'exécution de `\pgfkeysfiltered`. À ce moment-là, le chemin complet de la clé est entreposé dans `\pgfkeyscurrentkey`, dans `\pgfkeyscurrentkeyRAW` le nom de la clé avant que l'on tienne compte du chemin par défaut et la valeur dans `\pgfkeyscurrentvalue`.

De plus `\pgfkeysnumber` contient le type de la clé sous forme d'un nombre entier.

<1> La clé est une clé de commande — c.-à-d. que `/.@cmd` existe.

⟨2⟩ La clé contient sa valeur directement.

⟨3⟩ La clé est manipulée — par `/.code` ou `/.cd` p. ex. .

Dans ce cas, `\pgfkeyscurrentname` et `\pgfkeyscurrentpath` valent, respectivement, le nom et le chemin du manipulateur. Il faut employer `\pgfkeyssplitpath{}` pour extraire ces renseignements quand la clé n'est pas manipulée.

⟨0⟩ La clé est inconnue.

Tout filtre et tout manipulateur de filtre a accès à ces variables. Les filtres sont sensés régler le booléen `\ifpgfkeysfiltercontinue` selon que la clé en cours doit être traitée ou non.

`\pgfkeysevalkeyfilterwith{⟨filtre complet⟩}=⟨arguments de filtre⟩`

Évalue un *⟨filtre⟩* complètement qualifié — c.-à-d. avec un chemin absolu — avec les *⟨arguments de filtre⟩*.

```
\pgfkeysevalkeyfilterwith{/pgf/key filters>equals=tikz}
```

Index

Cet index ne contient que des entrées créés automatiquement. Un bon index devrait comprendre également des mots-clés soigneusement choisis. Cet index n'est pas un bon index.

- `<signification du caractere>` clef, 8
- `.activate family` manipulateur, 26
- `active families` clef, 27
- `active families and known` clef, 27
- `active families or descendants of` clef, 27
- `active families or no family` clef, 27
- `active families or no family DEBUG` clef, 27
- `.add` manipulateur, 18
- `.add code` manipulateur, 16
- `add exception` clef, 13
- `.add style` manipulateur, 17
- `and` clef, 28
- `.append` manipulateur, 18
- `.append code` manipulateur, 16
- `append filtered to` clef, 26
- `.append style` manipulateur, 18
- `.belongs to family` manipulateur, 27
- `boolean expected` clef, 23
- `.cd` manipulateur, 14
- `.code` manipulateur, 15
- `.code 2 args` manipulateur, 15
- `.code args` manipulateur, 16
- `.code n args` manipulateur, 16
- `.deactivate family` manipulateur, 27
- `.default` manipulateur, 14
- `defined` clef, 29
- `.ecode` manipulateur, 15
- `.ecode 2 args` manipulateur, 16
- `.ecode args` manipulateur, 16
- `.ecode n args` manipulateur, 16
- Environments
 - `pgfkeysinterruptkeyfilter`, 29
- `equals` clef, 28
- `/errors/`
 - `boolean expected`, 23
 - `unknown choice value`, 23
 - `unknown key`, 23
 - `value forbidden`, 23
 - `value required`, 23
- `.estore in` manipulateur, 19
- `.estyle` manipulateur, 17
- `.estyle 2 args` manipulateur, 17
- `.estyle args` manipulateur, 17
- `exec` clef, 23
- `.expand once` manipulateur, 20
- `.expand twice` manipulateur, 20
- `.expanded` manipulateur, 20
- Extensions et fichiers
 - `pgfkeys`, 4
- `false` clef, 29
- Fichier, *voir* Extensions et fichiers
- `first char syntax` clef, 8
- `.forward to` manipulateur, 20
- `.get` manipulateur, 18
- `/handler config/`
 - `only existing/`
 - `add exception`, 13
- `handler config`, 12
- `handler config` clef, 12
- `/handlers/`
 - `first char syntax/`
 - `<signification du caractere>`, 8
 - `first char syntax`, 8
- `ignore` clef, 26
- `.initial` manipulateur, 18
- `.install key filter` manipulateur, 25
- `.install key filter handler` manipulateur, 26
- `.is choice` manipulateur, 19
- `is descendant of` clef, 28
- `.is family` manipulateur, 14, 26
- `.is if` manipulateur, 19
- `.lastretry` manipulateur, 23
- `.link` manipulateur, 18
- `.list` manipulateur, 20
- `log` clef, 26
- Manipulateurs
 - `.activate family`, 26
 - `.add`, 18
 - `.add code`, 16
 - `.add style`, 17
 - `.append`, 18
 - `.append code`, 16
 - `.append style`, 18
 - `.belongs to family`, 27
 - `.cd`, 14
 - `.code`, 15
 - `.code 2 args`, 15
 - `.code args`, 16
 - `.code n args`, 16
 - `.deactivate family`, 27
 - `.default`, 14
 - `.ecode`, 15
 - `.ecode 2 args`, 16
 - `.ecode args`, 16
 - `.ecode n args`, 16
 - `.estore in`, 19
 - `.estyle`, 17
 - `.estyle 2 args`, 17
 - `.estyle args`, 17
 - `.expand once`, 20
 - `.expand twice`, 20
 - `.expanded`, 20
 - `.forward to`, 20
 - `.get`, 18
 - `.initial`, 18

- .install key filter, 25
- .install key filter handler, 26
- .is choice, 19
- .is family, 14, 26
- .is if, 19
- .lastretry, 23
- .link, 18
- .list, 20
- .prefix, 18
- .prefix code, 16
- .prefix style, 18
- .retry, 22
- .search also, 21
- .show code, 23
- .show value, 23
- .store in, 19
- .style, 17
- .style 2 args, 17
- .style args, 17
- .style n args, 17
- .try, 22
- .value forbidden, 15
- .value required, 15

Manipulateurs de clés, *voir* Manipulateurs

not clef, 28

or clef, 28

/pgf/

- key filter handlers/
 - append filtered to, 26
 - ignore, 26
 - log, 26
- key filters/
 - active families, 27
 - active families and known, 27
 - active families or descendants of, 27
 - active families or no family, 27
 - active families or no family DEBUG, 27
 - and, 28
 - defined, 29
 - equals, 28
 - false, 29
 - is descendant of, 28
 - not, 28
 - or, 28
 - true, 28

\pgfkeys, 7

pgfkeys extension, 4

\pgfkeysactivatefamilies, 29

\pgfkeysactivatefamiliesandfilteroptions, 27

\pgfkeysactivatefamily, 29

\pgfkeysactivatesinglefamilyandfilteroptions, 28

\pgfkeysalso, 7

\pgfkeysalsofiltered, 25

\pgfkeysalsofilteredfrom, 25

\pgfkeysalsofrom, 25

\pgfkeysdeactivatefamily, 29

\pgfkeysdef, 10

\pgfkeysdefargs, 11

\pgfkeysdefnargs, 10

\pgfkeysedef, 10

\pgfkeysedefargs, 11

\pgfkeysedefnargs, 11

\pgfkeysevalkeyfilterwith, 30

\pgfkeysfiltered, 25

\pgfkeysgetfamily, 29

\pgfkeysgetvalue, 6

\pgfkeysifdefined, 7

\pgfkeysiffamilydefined, 29

\pgfkeysinstallkeyfilter, 29

\pgfkeysinstallkeyfilterhandler, 29

pgfkeysinterruptkeyfilter environment, 29

\pgfkeysisfamilyactive, 29

\pgfkeyslet, 6

\pgfkeyssavekeyfilterstateto, 29

\pgfkeyssetfamily, 29

\pgfkeyssetvalue, 6

\pgfkeysvalueof, 6

\pgfqkeys, 7

\pgfqkeysactivatefamiliesandfilteroptions, 28

\pgfqkeysactivatesinglefamilyandfilteroptions, 28

\pgfqkeysalso, 8

\pgfqkeysfiltered, 25

- .prefix manipulateur, 18
- .prefix code manipulateur, 16
- .prefix style manipulateur, 18

.retry manipulateur, 22

- .search also manipulateur, 21
- .show code manipulateur, 23
- .show value manipulateur, 23
- .store in manipulateur, 19
- .style manipulateur, 17
- .style 2 args manipulateur, 17
- .style args manipulateur, 17
- .style n args manipulateur, 17

true clef, 28

.try manipulateur, 22

unknown choice value clef, 23

unknown key clef, 23

/utils/

- exec, 23

.value forbidden manipulateur, 15

value forbidden clef, 23

.value required manipulateur, 15

value required clef, 23