Chapitre 6

Formes de normalisation

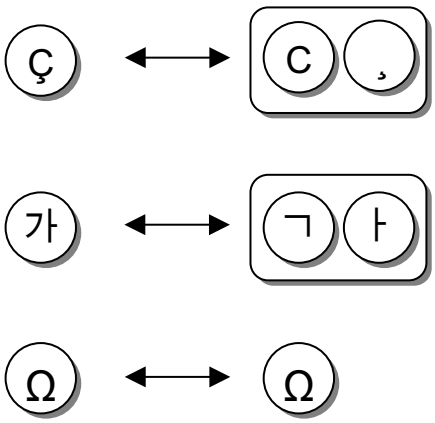
6.1 Introduction

Le standard Unicode définit deux formes d'équivalences entre caractères : l'équivalence canonique et l'équivalence de compatibilité. L'équivalence canonique établit une relation d'égalité fondamentale entre caractères ou suites de caractères. La *Figure 6-1b* décrit cette équivalence :

Figure 6-1. Équivalence canonique

- Équivalence fondamentale
- Indifférenciable aux yeux de l'utilisateur, si rendu correctement
- Comprend :
  - hangûl
  - suite de diacritiques
  - singleton[1]

Figure 6-1b. Exemples d'équivalence



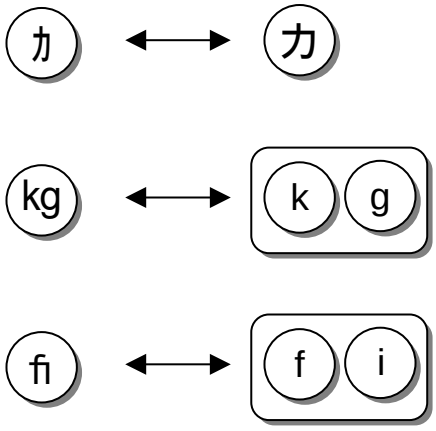
Afin de garantir une compatibilité aller-retour entre les normes préexistantes et Unicode, plusieurs codes d'Unicode représentent des entités qui ne sont que des variantes d'un même caractère générique. Les représentations visuelles qui illustrent ce caractère générique forment d'ordinaire un sous-ensemble de toutes ses représentations visuelles possibles. Unicode définit pour ces caractères des décompositions de compatibilité. L'apparence de ces caractères étant quelque peu différente ; les remplacer par un seul caractère entraînerait une perte potentielle d'information de formatage, à moins d'ajouter du balisage complémentaire. La *Figure 6-2* fournit quelques exemples d'équivalences de compatibilité.

[1] Caractères précomposés dont la décomposition consiste en un seul caractère et non une suite de caractères comme c'est le plus souvent le cas.

Les chapitres 2 et 3 exposent de manière plus détaillée ces deux types d'équivalence. En outre, la section *5.7, Normalisation*, décrit également plusieurs formes de normalisation. Unicode définit ces formes de normalisation afin de pouvoir produire à partir de toute chaîne une seule forme normalisée. La section *3.6, Décomposition*, définit de manière précise deux de ces formes. Elle définit notamment la décomposition canonique qui peut être utilisée dans l'échange normalisé de textes. Cette forme permet d'effectuer une comparaison binaire tout en conservant une équivalence canonique avec le texte non normalisé d'origine.

Figure 6-2. Équivalences de compatibilité

- Différences de formatage
  - Variante de fonte (ℋ)
  - Différence de sécabilité (-)
  - Forme cursive (٩ ھ ڪ ه)
  - Cerclé (⑩)
  - Chasse, force, rotation (ｶ & ﹀)
  - Exposant, indice ($_9{}^9$)
  - Disposé en carré (㌵)
  - Fraction (¾)
  - Autres (dž)



La section 3.6 définit également la *décomposition de compatibilité*. Celle-ci permet d'effectuer une comparaison binaire tout en conservant cette-fois-ci une équivalence de compatibilité avec le texte non normalisé d'origine. Cette dernière forme peut s'avérer utile en maintes occasions puisqu'elle permet d'éliminer des différences qui ne sont pas toujours pertinentes. Les caractères *katakana* à pleine chasse et à demi-chasse ont les mêmes décompositions de compatibilité et sont donc compatibles ; ils ne sont toutefois pas canoniquement équivalents.

Ces deux formes normalisent vers des caractères décomposés. Bien que la section *3.6, Décomposition,* mentionne également l'existence d'une normalisation vers des caractères composites[2], elle n'en précise pas la forme. La nature des formes précomposées dans le standard Unicode permet d'envisager plusieurs normalisations à base de caractères composites. Ce chapitre donne une définition unique de la normalisation et identifie chacune des formes normalisées.

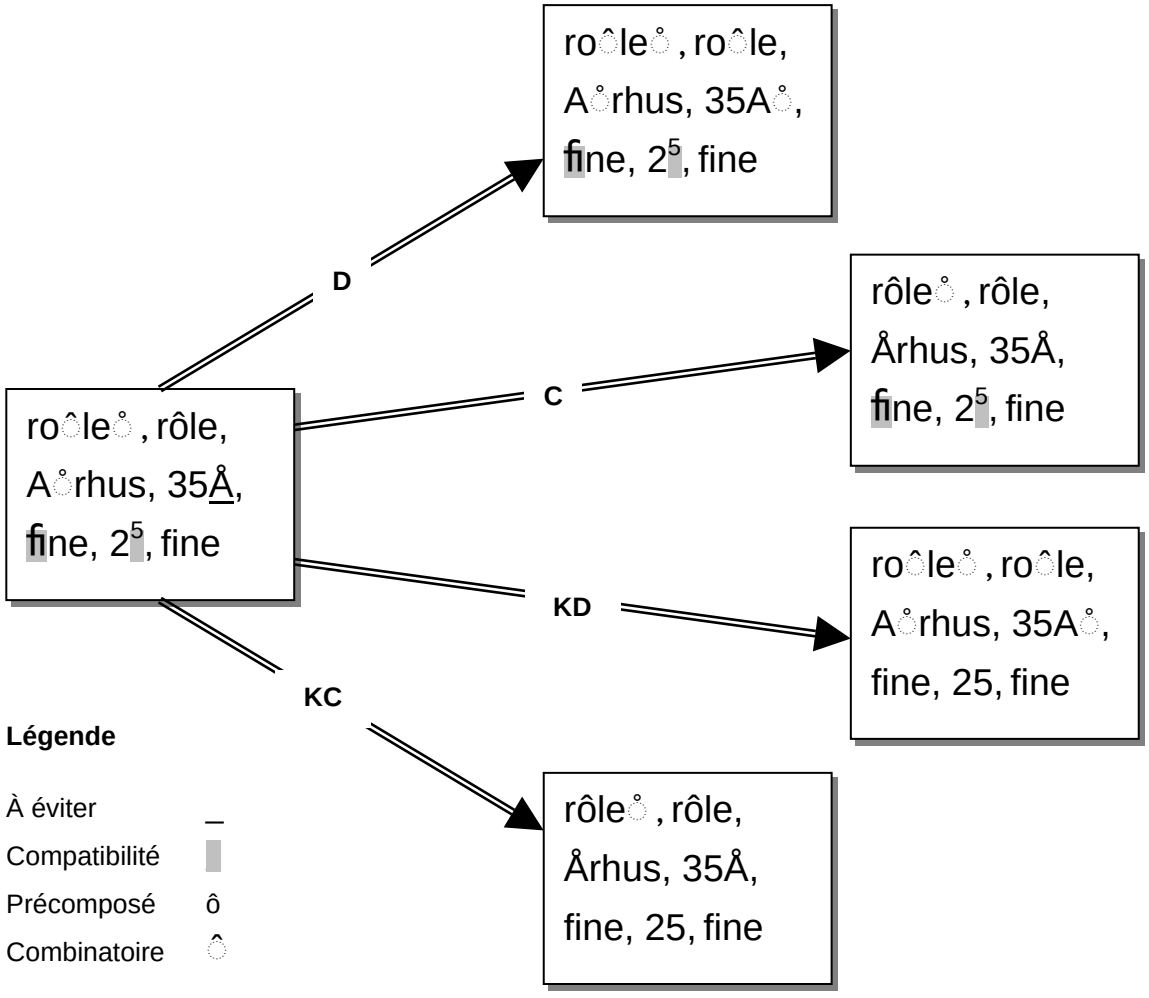
Le *Tableau 6-1* désigne les quatre formes de normalisation. Tout comme pour la décomposition, il existe deux formes normalisées vers les caractères précomposés : C et KC. Le résultat de l'une est l'équivalent canonique du texte non normalisé d'origine, le résultat de l'autre est équivalent de compatibilité. Le K de KD et KC représente le mot *compatibilité* (suggéré par l'allemand *kompatibel*) alors que le C signifie *canonique*. Les deux types de normalisation peuvent s'avérer utiles. Le D lui se réfère à la décomposition.

[2] Également connus sous le nom de caractères précomposés ou décomposables.

Tableau 6-1. Quatre formes de normalisation

Nom	Description	Définition
Forme de normalisation D (FND)	Décomposition canonique	Sections *3.6, Décomposition, 3.10, Mise en ordre canonique,* et *3.11, Comportement des jamos jointifs.*
Forme de normalisation C (FNC)	Décomposition canonique suivie d'une composition canonique	*Section 6.5, Définition.*
Forme de normalisation KD (FNKD)	Décomposition de compatibilité	Sections *3.6, Décomposition, 3.10, Mise en ordre canonique,* et *3.11, Comportement des jamos jointifs.*
Forme de normalisation KC (FNKC)	Décomposition de comptabilité suivie d'une composition canonique	*Section 6.5, Définition*.

Figure 6-3. Application des quatre formes de normalisation



La *Figure 6-3* ci-dessus illustre l'effet de l'application de différentes formes de normalisation sur un texte non normalisé. Le tableau illustre les transformations qui affectent différents types de caractère.

Quelle que soit la forme de normalisation, on remplace les caractères singletons (ceux qui possèdent une transformation canonique vers un seul caractère). Les formes D et C conservent les caractères de compatibilité (ceux qui possèdent une transformation de compatibilité), alors que les formes KD et KC les remplacent par le résultat de cette transformation. Ces remplacements peuvent parfois entraîner la perte d'information importante, en l'absence de balisage complémentaire.

Les formes D et KD transforment les caractères précomposés en leurs décompositions canoniques, alors que les formes C et KC transforment les caractères combinatoires en caractères précomposés. Remarquons qu'en l'absence d'un caractère précomposé pour e-rond (dans ro◌̂le◌̊ de la *Figure 6-3*), les formes C et KC le laissent sous sa forme décomposée.

Toutes ces définitions découlent des règles d'équivalence et de composition exposées au chapitre *3, Conformité,* et des listes de décompositions de la *Base de données de caractères Unicode.*

> **Remarque :** les parties de texte qui ne comprennent que des caractères ASCII (U+0000 à U+007F) ne sont pas affectées par ces différentes formes de normalisation. Ceci est particulièrement important pour les langages informatiques (voir 6.12, *Identificateurs dans les langages informatiques*).

La forme de normalisation C utilise dans la mesure du possible des caractères canoniques précomposés et préserve la distinction entre caractères équivalents en terme de compatibilité. D'ordinaire, les chaînes de caractères accentués précomposés Unicode sont déjà sous la forme de normalisation C. Les mises en œuvre qui se bornent à utiliser un répertoire qui ne reprend aucun signe combinatoire (comme celles qui disent implanter le niveau 1 défini par l'ISO/CEI 10646-1) utilisent aussi, habituellement, la forme de normalisation C. Les mises en œuvre de versions ultérieures de l'ISO 10646 doivent considérer les problèmes potentiels liés aux différences de versions (voir *6.3, Versions et stabilité*).

Le modèle de caractères du W3C[3] utilise la forme de normalisation C en XML et dans les normes connexes à celui-ci (ce document n'est pas encore final, mais ce choix de forme de normalisation devrait rester).

La forme de normalisation KC élimine également les différences entre les caractères compatibles[PA14] que l'on distingue souvent à mauvais escient. C'est pourquoi les caractères *katakana* de demi-chasse et de pleine chasse sont normalisés vers une même valeur ; il en va de même pour les chiffres romains et leurs lettres correspondantes[4]. La section *6.7, Exemples et tableaux,* fournit de nombreux autres exemples complets.

On veillera à ne pas appliquer les formes de normalisations KC et KD sans discernement à n'importe quel texte. Ces formes rendent impossible la conversion aller-retour entre Unicode et de nombreux jeux de caractères préexistants, car elles éliminent bon nombre de différences de formatage. À moins qu'elles ne suppléent à cette suppression par du balisage de formatage complémentaire, elles risquent donc d'éliminer des distinctions cruciales à la compréhension du texte.

[3] <http://www.w3.org/TR/WD-charmod>

[4] Par exemple, I en chiffre romain ≈ la lettre i majuscule, V en chiffre romain ≈ la lettre v majuscule

La meilleure façon d'appréhender ces formes de normalisation est sans doute de les considérer comme analogue à un passage en majuscules ou en minuscules : utile parfois pour indiquer le sens, mais parfois une modification de texte inappropriée. On peut les utiliser avec plus de liberté pour des applications aux jeux de caractères restreints, comme à la section *6.12*, *Identificateurs dans les langages informatiques*.

En résumé, voici l'effet des formes de normalisation sur les caractères précomposés de compatibilité présents dans le texte-source :

- les formes D et C conservent les précomposés de compatibilité ;
- ni KD ni KC ne conservent les précomposés de compatibilité ;
- aucune de ces formes n'introduit de précomposés de compatibilité absents du texte-source.

Remarque : la forme de normalisation KC *ne* tente *pas* de transformer les suites de caractères vers des composés de compatibilité. Ainsi, une composition de compatibilité de « suffit » *ne* produit *pas* "su\uFB03t" bien que '\uFB03' soit un caractère équivalent de compatibilité des trois caractères « ffi ».

La concaténation des chaînes n'est fermée dans aucunes des formes de normalisation. Considérons les exemples suivants :

Tableau 6-2. La concaténation n'est pas fermée

Forme	Chaîne1	Chaîne2	Concaténation	Normalisation correcte
C	"a"	"^"	"a"+"^"	"â"
D	"a"+"^"	"." (point souscrit)	"a"+"^" + "."	"a" + "." +"^"

À moins de limiter le répertoire, il est impossible de concevoir une forme de normalisation dans laquelle la concaténation simple soit fermée. Au besoin, on peut cependant écrire une fonction spécialisée qui produit une concaténation normalisée. Par contre, la sélection de sous-chaînes est fermée dans l'ensemble des chaînes normalisées, quelle que soit la forme de normalisation utilisée.

6.2 Notation

Toutes les définitions de ce chapitre reposent sur les règles d'équivalence et de décomposition énoncées au chapitre *3, Conformité*, et sur les décompositions et les classes combinatoires définies dans la *Base de données de caractères Unicode*. Les décompositions *doivent* respecter ces règles. On doit, notamment, appliquer récursivement les décompositions définies dans la *Base de données de caractères Unicode* ; la chaîne-résultat doit ensuite être mise en ordre canonique selon les classes combinatoires des caractères qui la composent. Pour plus de concision, on utilise la notation suivante.

- On abrège les noms ISO 10646 de la manière suivante:

E-grave	=	U+00C8	È	LETTRE MAJUSCULE LATINE E ACCENT GRAVE
ka	=	U+30AB	カ	SYLLABE KATAKANA KA
ka_dc	=	U+FF76	ｶ	SYLLABE KATAKANA KA DEMI-CHASSE
ten	=	U+3099	◌゙	DIACRITIQUE KATAKANA-HIRAGANA SON VOISÉ
ten_dc	=	U+FF9E	ﾞ	MARQUE KATAKANA DE SON VOISÉ DEMI-CHASSE

- On peut désigner la classe combinatoire d'un caractère X de la manière suivante : classeCombinatoire(X).
- On peut représenter une suite de caractères en séparant les différents noms de caractère à l'aide de « + » ou en utilisant la notation des chaînes.
- "...\uxxxx..." représente le caractère Unicode U+xxxx au sein d'une chaîne de caractères.
- Un caractère unique équivalent à la suite de caractères B + C peut s'écrire B-C.
- On abrège les différentes formes de normalisation d'une chaîne C de la manière suivante : FND(C), FNKD(C), FNC(C) et FNKD(C). FNX(C) représente n'importe quelle forme de normalisation.
- On représente les divers types de jamos jointifs (initiaux, médiaux, finaux) à l'aide de lettres souscrites comme dans k_i, a_m et k_f.
- Il est permis d'utiliser des accents avec chasse (sans cercle en pointillé) pour représenter des accents sans chasse : « c, » pour désigner *c* suivi d'une *cédille sans chasse*.

6.3 Versions et stabilité

Il est crucial que les formes de normalisation demeurent stables au fil du temps. Ceci signifie que, si l'on normalise une chaîne (qui ne contient pas de caractères non attribués) sous une version d'Unicode, cette chaîne doit rester normalisée dans les versions ultérieures d'Unicode. C'est ce qu'on appelle l'exigence de rétrocompatibilité. Afin de la respecter, on désigne une *version fixe* pour le processus de composition. On la nomme la *version de composition.*

La version de composition correspond à la version 3.1.0 de la *Base de données de caractères Unicode*. Afin de mieux comprendre l'importance de la version de composition, supposons que la version 4.0 d'Unicode ajoute le précomposé Q-caron. Pour les mises en œuvre qui utilisent la version 4.0, les chaînes sous les formes de normalisation C et KC continueront à contenir la suite Q + caron et *non* le nouveau caractère Q-caron, puisque la composition canonique pour Q-caron n'avait pas été définie pour la version de composition. Pour plus d'informations, voir la section *6.6, Tableau des compositions exclues*.

> Remarque : il est possible que de nouvelles compositions soient ajoutées dans des versions ultérieures d'Unicode, pour autant que l'exigence de rétrocompatibilié soit satisfaite. Ceci signifie que, pour toute nouvelle composition $XY \rightarrow Z$, X ou Y doivent être un nouveau caractère, ainsi que Z. Toutefois, Unicode décourage fortement la définition de nouvelles compositions, même quand elles respectent les restrictions énoncées ci-dessus.

Il ne faut pas seulement établir la version de composition pour les prochaines versions d'Unicode, il est également nécessaire de limiter le genre de modifications qui pourront être apportées aux propriétés de caractère. C'est pourquoi, le consortium Unicode a établi une politique claire qui garantit la stabilité des formes de normalisation. Pour de plus amples renseignements, veuillez vous référer aux *Lignes de conduite Unicode*[5].

> Remarque : la version 3.1 constitue une exception à cette ligne de conduite. Voir la *Section 6.15*, *Corrigendum relatif à la normalisation 3.1.*

[5] <http://www.unicode.org/unicode/standard/policies.html>

6.4 Conformité

C1. Tout processus qui produit du texte Unicode qui se veut dans une forme normalisée doit le faire conformément aux prescriptions de ce document.

C2. Tout processus qui désire établir si un texte Unicode se présente sous une forme normalisée doit le faire conformément aux prescriptions de ce document.

C3. Tout processus qui désire transformer du texte en une forme normalisée doit satisfaire au test de conformité de normalisation[6]*.*

> **Remarque :** la définition des formes de normalisation est décrite du point de vue d'un processus qui produit une décomposition ou une composition à partir d'une chaîne Unicode arbitraire. Il s'agit d'une description *logique* – certaines mises en œuvre peuvent adopter des mécanismes plus efficaces pour autant qu'elles produisent le même résultat. De même, il n'est pas nécessaire pour établir l'utilisation d'une forme de normalisation particulière d'employer le processus de normalisation, il suffit que le processus utilisé produise un résultat équivalent à une normalisation suivie d'un test d'identité binaire.

6.5 Définition

Cette section définit les formes de normalisation C et KC. Elle utilise les quatre définitions suivantes (D1, D2, D3 et D4) ainsi que deux règles (R1 et R2).

Toutes les suites de caractères combinatoires commencent par un caractère de classe combinatoire zéro. Afin de simplifier, on définit le terme suivant pour ces caractères :

D1. On dit d'un caractère T qu'il est une *tête* si sa classe combinatoire dans la *Base de données de caractères* est zéro.

Compte tenu de la définition de l'équivalence canonique, l'ordre des caractères combinatoires de même classe combinatoire importe. Ainsi, *a-macron-brève* n'est-il pas équivalent à *a-brève-macron.* On ne peut donc composer des caractères s'il en résultait une modification de l'ordre canonique des caractères combinatoires.

D2. Pour toute suite de caractères qui commence par une tête T, on dit qu'un caractère K est *isolé* de T si et seulement s'il existe un caractère B entre T et K tel que B est également une tête ou B appartient à la même classe combinatoire que K.

> **Remarque :** Quand B *isole* K, intervertir B et K produit une suite de caractères qui *n'*est *pas* canoniquement équivalente à la suite originelle. Voir la section *3.9, Mise en ordre canonique.*

> **Remarque :** Si une suite de caractères combinatoires se présente en ordre canonique, il suffit d'inspecter le caractère qui précède immédiatement un caractère K pour établir si K est isolé ou non.

[6] À partir d'Unicode 3.0.1, le fichier à utiliser pour le test de conformité de normalisation se trouve sur le disque optique qui accompagne cet ouvrage ; une version tenue à jour est disponible à l'adresse suivante <http://www.unicode.org/Public/UNIDATA/NormalizationTest.txt>. Ce fichier comprend une série de champs. L'application des formes de normalisation aux différents champs devra produire un résultat correspondant à celui déclaré dans l'en-tête de ce fichier.

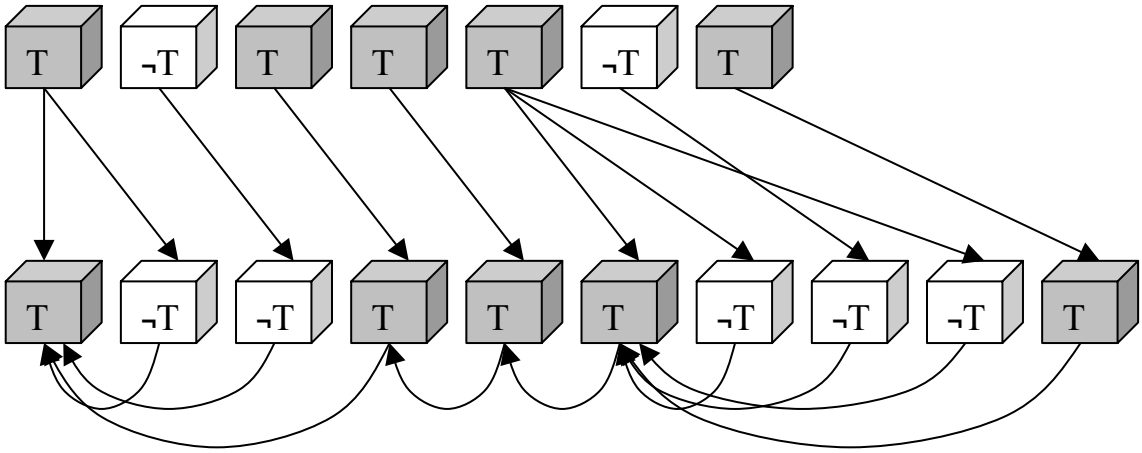
La formation d'une composition sous la forme normalisée C ou KC implique :

1. la décomposition de la chaîne en question conformément aux transformations canoniques (ou de compatibilité) indiquées dans la Base de données de caractères qui correspond à la dernière version d'Unicode supportée par la mise en œuvre, puis

2. la composition de la chaîne résultante conformément aux transformations *canoniques* de la version de composition[7] et cela par compositions successives de chaque caractère non isolé avec la tête qui le précède.

La figure 6-4 ci-dessous illustre le processus. Les cubes gris représentent les têtes, les cubes blancs les autres caractères. La première étape consiste à décomposer complètement la chaîne (les flèches vers le bas) et à la réordonner. Lors de la seconde étape, représentée par les arcs fléchés du bas de la figure, on compare chaque caractère au caractère précédent (autre qu'une tête) et on le combine à la tête précédente si toutes les conditions sont remplies. Voir les sections *6.7, Exemples et tableaux* et *6.10, Exemples de mise en œuvre*.

Figure 6-4. Processus de composition



Il faut également préciser quand il est permis de composer une tête et un caractère non isolé. Pour ce faire, on introduit les deux définitions suivantes :

D3. On dit d'un caractère qu'il est un *caractère précomposé principal* s'il fait l'objet d'une décomposition canonique dans la *Base de données de caractères Unicode* (ou s'il s'agit d'une syllabe hangûl canoniquement décomposable) mais n'apparaît pas dans le *Tableau des compositions exclues. Voir section 6.6, Tableau des compositions exclues*.

> **Remarque :** La décomposition d'une syllabe hangûl est considérée comme une décomposition canonique.

D4. On dit d'un caractère X qu'il est recomposable avec un caractère Y si, et seulement si, il existe un caractère précomposé principal Z canoniquement équivalent à la suite <X, Y>.

À partir de ces définitions, les règles suivantes définissent les formes de normalisation C et KC.

[7] C'est-à-dire la version de composition de la *Base de données de caractères Unicode*.

R1. FORME DE NORMALISATION C

On obtient la forme de normalisation C d'une chaîne de caractères C par l'application du processus suivant ou de tout autre processus produisant le même résultat :

1. Produire la décomposition *canonique* de la chaîne d'origine C conformément aux décompositions répertoriées dans la dernière *version supportée* de la *Base de données de caractères Unicode.*

2. Pour chaque caractère K de cette décomposition, en commençant par le premier, si rien n'isole K de la dernière tête D et si K est recomposable avec D, remplacer D par le précomposé D-K et supprimer K.

Le résultat de ce processus forme une nouvelle chaîne C' normalisée sous la forme normalisée C.

R2. FORME DE NORMALISATION KC

On obtient la forme de normalisation KC d'une chaîne de caractères C par l'application du processus suivant ou de tout autre processus produisant le même résultat :

1. Produire la décomposition de *compatibilité* de la chaîne d'origine C conformément aux décompositions répertoriées dans la dernière *version supportée* de la *Base de données de caractères Unicode.*

2. Pour chaque caractère K de cette décomposition, en commençant par le premier, si rien n'isole K de la dernière tête D et si K est recomposable avec D, remplacer D par le précomposé D-K et supprimer K.

Le résultat de ce processus forme une nouvelle chaîne normalisée C' sous la forme KC.

6.6 Tableau des compositions exclues

Il existe quatre classes de caractères exclues de la composition.

1. **Propres à une écriture :** caractères précomposés qui ne constituent pas généralement les formes privilégiées d'une écriture particulière.
   - On *ne* peut déterminer ces caractères à partir de l'information de la *Base de données de caractères Unicode*.

2. **Postérieurs à la version de composition :** caractères précomposés rajoutés à la version 3.0 d'Unicode. Cet ensemble sera mis à jour après chaque nouvelle version d'Unicode. Voir la section *6.3, Versions et stabilité.*
   - On *ne* peut déterminer ces caractères à partir de l'information fournie dans la *Base de données de caractères Unicode*.

3. **Singletons :** caractères précomposés dont la décomposition consiste en un seul caractère (voir la description ci-dessous).
   - On calcule ces caractères à partir de l'information fournie dans la *Base de données de caractères Unicode.*

4. **Décompositions sans tête :** caractères précomposés dont les décompositions ne commencent pas par une tête.
   - On calcule ces caractères à partir de l'information fournie dans la *Base de données de caractères Unicode.*

La *Base de données de caractères Unicode* peut fournir pour deux caractères distincts une même décomposition canonique. Voir l'exemple ci-dessous.

Sources	Même décomposition
U+212B Å SYMBOLE ANGSTRÖM	U+0041 A LETTRE MAJUSCULE LATINE A + U+030A ◌̊ DIACRITIQUE ROND EN CHEF
U+00C5 Å LETTRE MAJUSCULE LATINE A ROND EN CHEF	

La *Base de données de caractères Unicode* décomposera d'abord un des caractères en l'autre avant de décomposer ce caractère à son tour. En d'autres termes, un des caractères (ici U+212B Å SYMBOLE ANGSTRÖM) a une décomposition singleton. Unicode comprend des caractères à décomposition singleton essentiellement pour des raisons de compatibilité avec des normes et standards préexistants. Ces décompositions singleton ne font pas partie des compositions principales.

On trouvera un tableau des compositions exclues lisibles mécaniquement à l'adresse suivante <http://www.unicode.org/Public/UNIDATA/CompositionExclusions.txt. Remarquer que cette version corrige l'omission du YOD HIRIK. Voir la *Section 6.15*, *Corrigendum relatif à la normalisation 3.1 pour plus de détails.*

Les quatre différentes classes de caractères sont incluses dans ce fichier, bien que les singletons et les décompositions sans tête sont mis en commentaires.

6.7 Exemples et tableaux

Cette section énumère quelques exemples de chaque forme de normalisation.

Exemples communs

Les exemples suivants représentent des cas où les formes D et KD sont identiques et les formes C et KC sont identiques

	Original	Formes D, KD	Formes C, KC	Remarques
a	D-point_en_chef	D + point_en_chef	D-point_en_chef	Les équivalents canoniques précomposées et décomposées produisent le même résultat.
b	D + point_en_chef	D + point_en_chef	D-point_en_chef	
c	D-point_souscrit + point_en_chef	D + point_souscrit + point_en_chef	D-point_souscrit + point_en_chef	Au moment où l'on atteint le *point_en_chef*, on ne peut plus l'adjoindre au caractère de base. Il se peut que des signes combinatoires supplémentaires se glissent dans la suite (voir **f**) pour autant que le résultat de la combinaison soit canoniquement équivalente.
d	D-point_en_chef + point_souscrit	D + point_souscrit + point_en_chef	D-point_souscrit + point_en_chef	
e	D + point_en_chef + point_souscrit	D + point_souscrit + point_en_chef	D-point_souscrit + point_en_chef	
f	D + point_en_chef + cornu + point_souscrit	D + cornu + point_souscrit + point_en_chef	D-point_souscrit + cornu + point_en_chef	
g	E-macron-grave	E + macron + grave	E-macron-grave	Adjonction de plusieurs caractères combinatoires au caractère de base.
h	E-macron + grave	E + macron + grave	E-macron-grave	

i	E-grave + macron	E + grave + macron	E-grave + macron	Le macron ne se combine pas, puisque E-grave-macron n'existe pas et E-macron-grave n'est pas canoniquement équivalent.
j	signe_angstrom	A + rond	A-rond	On utilise la forme Å (A-rond) pour les deux caractères, car il s'agit du précomposé recommandé.
k	A-rond	A + rond	A-rond	

Exemples des formes de normalisation D et C

Ces exemples de formes D et C illustrent leurs différences respectives avec les formes KD et KC.

	Original	Forme D	Forme C	Remarques
l	"Äffin"	"A\u0308ffin"	"Äffin"	La *ligature_ffi* (U+FB03) *n'*est *pas* décomposée, car si elle possède une décomposition de compatibilité, elle n'en a pas de canonique. (Voir ci-dessous *Exemples des formes de normalisation KD et KC.*)
m	"Ä\uFB03n"	"A\u0308\uFB03n"	"Ä\uFB03n"	
n	"Henri IV"	"Henri IV"	"Henri IV"	De même, le CHIFFRE ROMAIN IV (U+2163) n'est pas décomposé.
o	"Henri \u2163"	"Henri \u2163"	"Henri \u2163"	
p	ga	ka + ten	ga	Les différents équivalents de compatibilité d'un même caractère japonais *ne* produisent *pas* la même chaîne normalisée sous la forme C.
q	ka + ten	ka + ten	ga	
r	ka_dc + ten_dc	ka_dc + ten_dc	ka_dc + ten_dc	
s	ka + ten_dc	ka + ten_dc	ka + ten_dc	
t	ka_dc + ten	ka_dc + ten	ka_dc + ten	
u	kaks	$k_i + a_m + ks_f$	kaks	Les normalisations conservent les syllabes hangûl.

Exemples des formes de normalisation KD et KC

Ces exemples de formes KD et KC illustrent leurs différences respectives avec les formes D et C.

	Original	Forme KD	Forme KC	Remarques
l'	"Äffin"	"A\u0308ffin"	"Äffin"	La *ligature_ffi* (U+FB03) est décomposée pour la forme de normalisation KC (alors qu'elle ne l'est pas dans la forme de normalisation C).
m'	"Ä\uFB03n"	"A\u0308\ffin"	"Äffin"	

n'	"Henri IV"	"Henri IV"	"Henri IV"	Ici également, les chaînes normalisées sont identiques pour la forme de normalisation KC.
o'	"Henri \u2163"	"Henri IV"	"Henri IV"	
p'	ga	ka + ten	ga	Les différents équivalents de compatibilité d'un même caractère japonais produisent la même chaîne normalisée sous la forme KC.
q'	ka + ten	ka + ten	ga	
r'	ka_dc + ten_dc	ka + ten	ga	
s'	ka + ten_dc	ka + ten	ga	
t'	ka_dc + ten	ka + ten	ga	
u'	kaks	$k_i + a_m + ks_f$	kaks	Les normalisations conservent les syllabes hangûl.

* Dans les premières versions d'Unicode, les caractères jamos composés comme ks_f avaient une décomposition de compatibilité de type $k_f + s_f$. La version 2.1.9 d'Unicode a éliminé ces correspondances afin de préserver les syllabes hangûl.

6.8 Objectifs de conception

On trouvera ci-dessous les objectifs qui sous-tendent la conception des formes de normalisation.

Objectif 1 : Unicité

L'unicité est le premier, et de loin le plus important, objectif de conception des formes de normalisation. Ceci signifie que deux chaînes équivalentes doivent avoir précisément la même forme normalisée. De façon explicite :

1. Si deux chaînes *x* et *y* sont canoniquement équivalentes, alors
   - C(x) = C(y)
   - D(x) = D(y)

2. Si deux chaînes *x* et *y* sont des équivalents de compatibilité, alors
   - KC(x) = KC(y)
   - KD(x) = KD(y)

Objectif 2 : Stabilité

Le deuxième objectif de conception est la préservation des caractères qui n'interviennent pas dans le processus de composition ou de décomposition.

1. Si X contient un caractère ayant une décomposition de compatibilité, alors D(X) et C(X) contiennent toujours ce caractère.
2. Dans la mesure du possible, si X ne comprend pas de caractère combinatoire alors C(X) = X.
3. Les signes combinatoires non pertinents ne doivent pas influer sur le résultat des compositions. Voir l'exemple **f** à la section *6.7*, *Exemples et tableaux*, où le caractère *cornu* n'a pas d'incidence sur le résultat des compositions.

Remarque : Les seuls caractères pour lesquels l'objectif 2.2 ne se vérifie pas sont ceux énumérés dans la section *6.6*, *Tableau des compositions exclues*.

Objectif 3 : Efficacité

Le troisième objectif de conception est de permettre des mises en œuvre efficaces.

1. Il est possible de produire les formes de normalisation à l'aide de code efficace. Plus particulièrement, on doit être en mesure de produire très rapidement la forme de normalisation C à partir des chaînes qui sont déjà en forme de normalisation C ou D.
2. Les formes de compositions ne doivent pas nécessairement produire les résultats les plus concis car cela peut entraîner des calculs excessifs.

6.9 Notes de mise en œuvre

On peut optimiser de diverses façons les programmes qui produisent la forme de normalisation C. Plutôt que de commencer par la décomposition du texte au complet, on peut, par exemple, vérifier rapidement chaque caractère. Si celui-ci est déjà dans la forme précomposée adéquate alors on passe au caractère suivant sans autre procès. On n'invoque du code plus complexe que dans le cas où le caractère courant est combinatoire ou s'il appartient au *Tableau des compositions exclues* (voir la section 6.6). Dans ce cas, il faudra vérifier les caractères précédents jusqu'à la dernière tête. Voir la section *6.13*, *Détection des formes de normalisation* pour un complément d'information.

Lors de la composition, la majorité du temps de calcul se passe à extraire les données appropriées. La consultation des données pour la forme de normalisation C peut être mise en œuvre de manière très efficace car elle n'implique que des paires de caractères et non des chaînes arbitraires. On utilise d'abord un tableau à plusieurs niveaux (également appelé un *trie*, voir le *chapitre 5*) pour faire correspondre au caractère *c* un petit entier *i* appartenant à un intervalle continu de 0 à *n*. Le code pour ce faire ressemble à :

```
i = données[indexe[c >> GLISSEMENTBLOC] + (c & MASQUEBLOC)];
```

Ensuite, à l'aide d'une matrice, on fait correspond à une paire de ces petits entiers la valeur résultat. Cet algorithme est bien plus efficace que ceux qui utilisent des méthodes universelles comme les tables de hachage.

Les compositions et décompositions hangûl sont algorithmiques, on peut donc réduire radicalement les besoins en mémoire si l'on code les opérations correspondantes à l'aide d'instructions plutôt que de grandes matrices Voir la section *6.14*, *Hangûl*, pour plus d'informations.

> **Remarque** : il est important de s'assurer que la mise en œuvre produit bien des résultats conformes. À cette fin, elle doit réussir les tests de conformité de normalisation prévus : <http://www.unicode.org/Public/UNIDATA/NormalizationTest.txt>.

6.10 Exemples de mise en œuvre

Le consortium Unicode fournit des programmes-modèles pour chacune des quatre formes de normalisation. Pour plus de clarté, ces modèles ne sont pas optimisés. Dans les cas des formes C et KC, ils transforment une chaîne en deux passes : une décomposition suivie de recompositions successives de chacun des caractères non isolés avec la dernière tête.

Pour certaines mises en œuvre, il se peut que les caractères en entrée ou en sortie soient fournis par petits paquets. Il est alors nécessaire de stocker dans un tampon le texte jusqu'à la dernière tête. On peut vider le tampon dès qu'on désire insérer une seconde tête.

Les modèles sont écrits en Java, toutefois – afin d'être plus accessibles – ils n'emploient pas de techniques orientées objet. Le code de ces modèles et une démonstration interactive sont disponibles[8]. Le W3C fournit sur son site du code Perl équivalent[9].

6.11 Codages préexistants

Bien que les formes de normalisation soient destinées à qualifier du texte Unicode, elles peuvent également être utilisées pour les codages de caractères non-Unicode, en d'autres mots les codages préexistants. Pour ce faire, on effectue une correspondance aller-retour entre le jeu de caractères préexistant et Unicode en utilisant les définitions D5 et D6.

D5. Un *transcodage inversible* T pour un jeu de caractères préexistant P est une relation bijective entre les caractères codés de P et les caractères d'Unicode telle que la relation inverse T^{-1}, appliquée à toute chaîne C de P, vérifie l'équation $T^{-1}(T(C)) = C$.

> **Remarque 1 :** On admet habituellement un seul transcodage pour un jeu de caractères préexistant donné. Il se peut cependant que plusieurs transcodages existent, c'est le cas du Shift-JIS auquel on peut associer deux transformations, l'une pour conserver la signification « / » de $2F_{16,}$ l'autre pour conserver la signification « ¥ ».
>
> **Remarque 2 :** La position des caractères dans la chaîne codée à l'aide du jeu préexistant peut être très différente de celle des caractères équivalents en Unicode. Ainsi, si la chaîne préexistante utilise un codage visuel de l'hébreu, le premier caractère peut devenir le dernier caractère de la chaîne Unicode.

Il est préférable que les routines de transcodage destinées aux jeux de caractères préexistants produisent, dans la mesure du possible, des chaînes normalisées sous la forme C. Pour plus d'informations, veuillez vous référer au rapport technique Unicode n° 22, *Character Mapping Tables*.

D6. Soit une chaîne C codée en P et un transcodage inversible T pour P, on définit *la forme de normalisation X de C sous T* comme étant le résultat des opérations suivantes : transformation vers Unicode, normalisation vers la forme X et transformation inverse vers le codage de caractères d'origine, c'est à dire $T^{-1}(FNX(T(C)))$. Quand on n'admet qu'un seul transcodage inversible pour un jeu de caractères donné, on peut alors simplement parler de la forme de normalisation X de C.

On classe les jeux de caractères préexistants en trois catégories selon leur comportement lors de la normalisation avec les transcodeurs acceptés.

[8] <http://www.unicode.org/unicode/reports/tr15/Normalizer.html>.

[9] <http://www.w3.org/International/charlint>

- *Prénormalisé :* toute chaîne dans le jeu de caractères est déjà en forme normalisée X.
    - Exemple : ISO 8859-1 est prénormalisée en forme normalisée C.
- *Normalisable :* bien que le jeu de caractères ne soit pas prénormalisé, toute chaîne du jeu *peut* être normalisée sous la forme X.
    - Exemple : l'ISO 2022 (avec une dose d'ISO 5426 et d'ISO 8859-1) est normalisable.
- *Non normalisable* : Certaines chaînes formées à partir du jeu de caractères ne peuvent être normalisées sous la forme X.
    - Exemple : l'ISO 5426 n'est pas normalisable sous la forme C par les transcodeurs ordinaires car elle contient des signes combinatoires sans en contenir des formes précomposées.

6.12 Identificateurs dans les langages informatiques

Cette section aborde les problèmes liés à la normalisation des identificateurs dans les langages de programmation ou de script informatiques. Le standard Unicode recommande une syntaxe pour les identificateurs de langage informatique qui permet l'utilisation de caractères non-ASCII au sein de ces identificateurs. Il s'agit d'un prolongement naturel de la syntaxe des identificateurs utilisée pour le C et plusieurs autres langages informatiques.

```
<identificateur> ::= <début_identificateur> (<début_identificateur> |
                     <suite_identificateur> )*

<début_identificateur> ::= [{Lu}{Ll}{Lt}{Lm}{Lo}{Nl}]

<suite_identificateur> ::= [{Mn}{Mc}{Nd}{Pc}{Cf}]
```

En d'autres mots, le premier caractère d'un identificateur peut être une *lettre majuscule*, *minuscule*, en *casse de titre*, une *lettre modificative*, une *autre lettre* ou une *lettre numérale*. Les caractères subséquents d'un identificateur peuvent appartenir à une des catégories susmentionnées ou encore aux *signes combinatoires avec ou sans chasse*, aux *chiffres décimaux*, à la *ponctuation connective* et aux *codes de formatage* (par exemple les *marques gauche-à-droite*). En règle générale, on éliminera les codes de formatage avant toute comparaison ou stockage.

On peut utiliser les formes de normalisation décrites dans ce chapitre pour éviter que des identificateurs apparemment identiques ne soient pas traités de manière équivalente. De tels problèmes peuvent surgir lors de la compilation ou de l'édition des liens, plus particulièrement lorsqu'on utilise différents langages de programmation. Afin de prévenir de tels désagréments, les compilateurs devraient normaliser les identificateurs avant de les stocker ou de les comparer. Le plus souvent, si un langage de programmation distingue la casse de ses identificateurs, on utilisera alors la forme de normalisation C (FNC). Dans le cas contraire, on adoptera la forme de normalisation KC (FNKC) généralement plus adaptée.

Si un langage de programmation utilise la FNKC pour uniformiser (« plier ») les différences entre caractères, il faut alors modifier légèrement la syntaxe des identificateurs fournie ci-dessus afin de traiter correctement les particularismes d'un petit nombre de caractères. Ces caractères se classent en trois catégories :

1. **Point médian.** La plupart des textes catalans préexistants sont codés en Latin-1, il faut donc admettre U+00B7 • POINT MÉDIAN dans la <suite_identificateur>. (Si le langage de programmation utilise un point comme opérateur, il faut alors distinguer son utilisation et plutôt utiliser U+2219 ∙ OPÉRATEUR PUCE ou U+22C5 ⋅ OPÉRATEUR POINT. Il faut toutefois traiter U+00B7 • POINT MÉDIAN avec beaucoup de précautions puisque de nombreux processus pourraient le considérer comme une marque de ponctuation plutôt qu'un modificateur de digramme[10]).

2. **Caractères combinatoïdes.** Certains caractères, bien qu'ils ne soient pas à strictement parler des caractères combinatoires, se comportent comme s'ils en étaient. En principe, ces caractères ne devraient pas faire partie de <début_identificateur>, mais bien de la <suite_identificateur> en compagnie des autres caractères combinatoires. Dans la plupart des cas, la mauvaise affectation de ces caractères ne pose pas problème. Cependant, quand ces caractères possèdent des décompositions de compatibilité, il se peut alors que la normalisation KC de ne soit plus fermée dans l'ensemble des identificateurs. Il faut plus particulièrement s'assurer que les quatre caractères suivants fassent partie de <suite_identificateur> et non de <début_identificateur> :

   - 0E33 ◌ำ LETTRE THAÏE SARA AM[11]
   - 0EB3 ◌ຳ VOYELLE DIACRITIQUE LAOTIENNE AM
   - FF9E ◌ﾞ MARQUE KATAKANA DE SON VOISÉ DEMI-CHASSE
   - FF9F ◌ﾟ MARQUE KATAKANA DE SON SEMI-VOISÉ DEMI-CHASSE

3. **Caractères à décomposition irrégulière.** U+037A ◌ͺ CARACTÈRE GREC IOTA SOUSCRIT et certaines formes de présentation arabes ont des décompositions de compatibilité irrégulière[12]. Il faut exclure ces caractères de <début_identificateur> et <suite_identificateur>. On préconise d'ailleurs d'exclure des identificateurs toutes les formes de présentation arabes même s'il suffit d'en exclure quelques-unes pour s'assurer que la normalisation des identificateurs sera fermée.

Avec ces amendements apportés à la syntaxe des identificateurs, toutes les formes de normalisation sont fermées pour tous les identificateurs. Ceci signifie donc que, pour toute chaîne C,

```
estIdentificateur(C) implique
                        estIdentificateur(FND(C))
                        estIdentificateur(FNC(C))
                        estIdentificateur(FNKD(C))
                        estIdentificateur(FNKC(C))
```

Les modifications de casse sont également fermées pour tous (à une exception près) les identificateurs, de telle sorte que pour toute chaîne C,

```
estIdentificateur(C) implique
                        estIdentificateur(enMajuscules(C))
```

[10] Le point médian (« punt volat » en catalan) apparaît dans le digramme du l géminé (`l·l`) et le distingue du digramme du l mouillé (`ll`).

[11] La classe de U+0E33 = {Lo}, classe valable pour un <début_identificateur>, alors que le premier caractère de sa décomposition de compatibilité (U+0E4D ◌ิ LETTRE THAÏE NIKHAHIT) a pour classe {Mn} qui ne peut pas faire partie de <début identificateur>.

[12] U+037A est une lettre modificative {Lm} dont la décomposition de compatibilité est une espace {Zs} suivie d'un iota souscrit sans chasse {Mn}. L'espace ne fait partie des caractères admis dans les identificateurs.

```
estIdentificateur(enMinuscules(C))
estIdentificateur(enCassePliée(C))
```

L'exception mentionnée est le caractère U+0345 ◌ͅ DIACRITIQUE GREC IOTA SOUSCRIT. Dans les cas extrêmement rares où l'U+0345 ◌ͅ DIACRITIQUE GREC IOTA SOUSCRIT est le premier caractère de C, U+0345 ne fait pas partie de <début_identitificateur>, mais sa forme majuscule[13] et à casse « pliée » si. En pratique, cela ne pose cependant pas de problème étant donné la manière dont on utilise la normalisation avec les identificateurs.

> Remarque : Les langages de programmation aux identificateurs insensibles à la casse doivent utiliser les transformations de casse définies dans le rapport technique Unicode n° 21, *Case Mappings*, afin de produire une forme normalisée insensible à la casse.

Il faut analyser le programme source à la recherche d'identificateurs avant d'uniformiser (de « plier ») les différences par l'application des transformations de casse ou de la forme normalisée KC. Bien sûr, il ne faut effectuer ce pliage ni sur des chaînes littérales ni sur des commentaires présents dans le texte du programme.

> Remarque : Unicode 3.1 prévoit des propriétés dérivées que les mises en œuvre peuvent utiliser pour analyser des identificateurs, normalisés ou non. Il s'agit des propriétés `ID_Start`, `ID_Continue`, `XID_Start`, `XID_Continue`. Unicode 3.1 facilite également la prise en charge du « pliage » de casse dans le cadre de la normalisation : on peut utiliser la propriété FNC lors du pliage de casse de telle sorte que le résultat du pliage de casse d'une chaîne FNKC est également une chaîne normalisée. DerivedProperties.html[14] précise l'emplacement de ces propriétés et les fichiers qui les contiennent.

6.13 Détection des formes de normalisation

Le tableau de données du fichier *NormalizationQuickCheck*[15] peut servir à déterminer rapidement si une chaîne se trouve sous une forme normalisée particulière. Ces données ne sont qu'informatives car on peut les reproduire à partir de la *Base de données de caractères Unicode*. Pour chaque forme de normalisation, le tableau attribue à chaque numéro de caractère une variable ternaire. Cette variable peut avoir les valeurs suivantes :

- NON indique que ce point de code ne peut se présenter pour cette forme de normalisation.
- OUI indique que ce point de code peut se présenter, pour peu qu'il soit en ordre canonique, mais sans aucune autre contrainte.
- PEUTÊTRE indique que ce point de code peut se présenter, pour peu qu'il soit en ordre canonique et que quelques contraintes supplémentaires soient respectées. En d'autres mots, le texte n'est peut-être pas sous cette forme normalisée, si le point de code en question est précédé par certains autres caractères.

[13] La lettre iota majuscule.

[14] <http://www.unicode.org/Public/UNIDATA/DerivedProperties.html >, les fichiers les plus importants sont : <http://www.unicode.org/Public/UNIDATA/DerivedNormalizationProperties.txt> et <http://www.unicode.org/Public/UNIDATA/DerivedCoreProperties.txt>.

[15] <http://www.unicode.org/unicode/reports/tr15/NormalizationQuickCheck.html>

Les programmes qui utilisent ces propriétés peuvent rapidement inspecter la chaîne en question lors d'une première passe pour vérifier la forme de normalisation. Le résultat de cette inspection peut être NON, OUI ou PEUTÊTRE. Pour OUI et NON, la question est tranchée. Dans le cas de PEUTÊTRE, il faut effectuer un examen plus approfondi. En règle générale, on normalise alors toute la chaîne sous la forme considérée et on la compare ensuite à l'original.

Cette vérification est nettement plus rapide que l'application directe de l'algorithme de normalisation, puisqu'elle n'alloue aucune mémoire et n'effectue aucune copie. Pour la grande majorité des chaînes, la réponse sera OUI ou NON. Un faible pourcentage nécessite donc un supplément de travail. Le code ci-dessous est en Java, bien que – pour des raisons de lisibilité – il n'emploie pas de techniques orientées objet.

```
public int vérifRapide (String source) {
    short dernièreClasseCanonique = 0;
    int résultat = OUI;
    for (int i = 0; i < source.length(); ++i) {
        char car = source.charAt(i);
        short classeCanonique = obtenirClasseCanonique (car);
        if (dernièreClasseCanonique > classeCanonique && classeCanonique != 0){
            return NON;
        }
        int vérif = estPermis(car);
        if (vérif == NON) return NON;
        if (vérif == PEUTÊTRE) résultat = PEUTÊTRE;
    }
    return résultat;
}

public static final int NON = 0, OUI = 1, PEUTÊTRE = -1;
```

L'appel `estPermis()` devrait accéder aux données[16] qui correspondent à la forme de normalisation en question. Pour plus d'informations, veuillez vous référer au fichier DerivedProperties.html faisant partie de la *Base de données de caractères Unicode* pour Unicode 3.1. En guise d'exemple, voici un extrait de ces données pour FNC :

```
...
0338       ; NFC_MAYBE # Mn DIACRITIQUE BARRE OBLIQUE LONGUE COUVRANTE
...
F900..FA0D ; NFC_NO    # Lo [270]
                       # IDÉOGRAMME DE COMPATIBILITÉ CJC-F900
                       # IDÉOGRAMME DE COMPATIBILITÉ CJC -FA0D
...
```

Ces lignes affectent la valeur NFC_MAYBE (PEUTÊTRE) au numéro de caractère U+0338 et la valeur NFC_NO (NON) aux numéros de caractère appartenant à l'intervalle U+F900 .. U+FA0D. Remarquon qu'on ne trouve aucune valeur PEUTÊTRE pour les FND et FNKD : la fonction `vérifRapide` doit donc toujours produire un résultat décisif pour ces formes de normalisation. La valeur implicite pour tous les caractères absents des fichiers est OUI.

Les données nécessaires à la mise en œuvre de `estPermis()` peuvent être accédées à l'aide d'une table de hachage ou d'un « trie » (un tableau multiétape, voir la *Section 6.9*, *Notes de mise en œuvre*). Cette dernière méthode est la plus rapide.

[16] <http://www.unicode.org/Public/UNIDATA/DerivedNormalizationProperties.txt>

6.14 Hangûl

Les compositions et décompositions hangûl étant algorithmiques, on peut réduire substantiellement les besoins de mémoire en effectuant les opérations correspondantes grâce à du code supplémentaire plutôt qu'en stockant les données dans des tableaux On trouvera ci-dessous un programme qui illustre la composition et la décomposition canoniques hangûl conformément à la *section 3.11, Comportement des jamos jointifs.* Bien que cet exemple soit codé en Java, on peut utiliser la même structure dans d'autres langages de programmation.

Constantes communes

```
static final int
    BaseS = 0xAC00, BaseI = 0x1100, BaseV = 0x1161, BaseF = 0x11A7,
    CompteI = 19, CompteV = 21, CompteF = 28,
    CompteN = CompteV * CompteF,   // 588
    CompteS = CompteI * CompteN;   // 11172
```

Décomposition hangûl

```
public static String décomposerHangûl(char s) {
    int IndexS = s - BaseS;
    if (IndexS < 0 || IndexS >= CompteS) {
        return String.valueOf(s);
    }
    StringBuffer résultat = new StringBuffer();
    int I = BaseI + IndexS / CompteN;
    int V = BaseV + (IndexS % CompteN) / CompteF;
    int F = BaseF + IndexS % CompteF;
    résultat.append((char)I);
    résultat.append((char)V);
    if (F != BaseF) résultat.append((char)F);
    return résultat.toString();
}
```

Composition hangûl

Remarquez une caractéristique importante de la composition hangûl : lorsque la chaîne d'origine n'est pas sous la forme normalisée D, il est insuffisant de détecter uniquement les suites de caractères de type <I, V> ou <I, V, F>. Il faut également déceler les suites de type <IV, F>. Afin de garantir l'unicité, il faut également composer ces suites. L'étape 2 ci-dessous illustre cet aspect de la composition.

```
public static String composerHangûl(String source) {
    int lon = source.length();
    if (lon == 0) return "";
    StringBuffer résultat = new StringBuffer();
    char dernier = source.charAt(0);            // copier premier car
    résultat.append(dernier);

    for (int i = 1; i < lon; ++i) {
        char car = source.charAt(i);

        // 1. Vérifier si les deux caractères courants sont I et V

        int IndexI = dernier - BaseI;
        if (0 <= IndexI && IndexI < CompteI) {
            int IndexV = car - BaseV;
            if (0 <= IndexV && IndexV < CompteV) {

                // former une syllabe de type IV
                dernier = (char)(BaseS + (IndexI * CompteV + IndexV) *
                                 CompteF);
```

```
                résultat.setCharAt(résultat.length()-1, dernier);
                                    // réinitialiser dernier
                continue; // ignorer car
            }
        }

        // 2. Vérifier si les deux caractères courants sont IV et F

        int IndexS = dernier - BaseS;

        if (0 <= IndexS && IndexS < CompteS && (IndexS % CompteF) == 0) {
            int IndexF = car - BaseF;
            if (0 <= IndexF && IndexF <= CompteF) {

                  // former une syllabe de type IVF

                dernier += IndexF;
                résultat.setCharAt(résultat.length()-1, dernier);
                                    // réinitialiser dernier
                continue; // ignorer car
            }
        }

        // Si aucun des deux cas, simplement ajouter le caractère
        dernier = car;
        résultat.append(car);

    }
    return résultat.toString();
}
```

On peut, pour différentes raisons, effectuer des transformations supplémentaires sur les suites de jamos. Ainsi, on peut insérer des bourres *tch'ôsong* et *djoungsong* pour rendre ces suites de jamos hangûl régulières selon la définition[17] du chapitre 3. Pour les suites saisies au clavier, il se peut qu'il faille effectuer d'autres compositions. On peut par exemple combiner la suite de consonnes finales k_f + s_f en ks_f. De surcroît, certaines méthodes d'entrée hangûl ne forcent pas l'utilisateur à distinguer les formes initiales et finales des consonnes et les modifient selon le contexte. Ainsi, dans la suite saisie au clavier m_i + $é_m$ + n_i + s_i + a_m, la méthode d'entrée interprète la consonne n_i comme n_f, car il n'existe pas de syllabe *nsa*. On aura donc comme résultat deux syllabes : *mén* et *sa*. Il faut cependant souligner qu'aucune de ces transformations supplémentaires ne fait partie des formes de normalisation Unicode.

Noms des caractères hangûl

On utilise également la décomposition hangûl pour déterminer le nom des caractères qui représentent les syllabes hangûl. Bien que le code ci-dessus ne fasse pas à proprement parler partie de la normalisation, il vaut la peine de l'inclure car il s'apparente énormément au code de la décomposition.

```
public static String obtenirNomHangûl (char s) {
    int IndexS = s - BaseS;
    if (0 > IndexS || IndexS >= CompteS) {
        throw new IllegalArgumentException("Pas une syllabe hangûl :" + s);
    }
    StringBuffer résultat = new StringBuffer();
    int IndexI = IndexS / CompteN;
    int IndexV = (IndexS % CompteN) / CompteF;
```

[17] Le texte anglais de la norme Unicode 2.0 nomme ces suites régulières de jamos des « suites normalisées ». Toutefois celles-ci n'ont rien à voir avec la composition ou la décomposition canonique.

```
        int IndexF = IndexS % CompteF;
        return "SYLLABE HANGÛL "    + TABLEAU_JAMOS_I[IndexI]
          + TABLEAU_JAMOS_V[IndexV] + TABLEAU_JAMOS_F[IndexF];
    }

    static private String[] TABLEAU_JAMOS_I = {
        "K", "KK", "N", "T", "TT", "L", "M", "P", "PP",
        "S", "SS", "", "TCH", "TCHTCH", "TCH'", "K'", "T'", "P'", "H"
    };

    static private String[] TABLEAU_JAMOS_V = {
        "A", "È", "YA", "YÈ", "O", "É", "YO", "YÉ", "Ô",
        "WA", "WÈ", "EU", "YÔ", "OU", "WO", "WÉ", "WI",
        "YOU", "Û", "ÛI", "I"
    };

    static private String[] TABLEAU_JAMOS_F = {
        "", "K", "KK", "KS", "N", "NTCH", "NH", "T", "L", "LK", "LM",
        "LP", "LS", "LT'", "LP'", "LH", "M", "P", "PS",
        "S", "SS", "NG", "TCH", "TCH'", "K'", "T'", "P'", "H"
    };
```

6.15 Corrigendum relatif à la normalisation 3.1

Lors de la production des tableaux de normalisation pour Unicode 3.0, U+FB1D ִ LETTRE HÉBRAÏQUE YOD HIRIK a par erreur été omis du fichier des compositions exclues. Pendant la période de révision publique, cette erreur fut signalée sans être prise en considération. Unicode 3.1 inclut désormais ce caractère parmi les compositions exclues.

> *Cette modification n'affecte pas la rétrocompatibilité des formes de normalisation KC et C des chaînes qui comprennent ce caractère. Il est conseillé à toutes les mises en œuvre de passer à la version 3.1 des tableaux de données Unicode.*

Lignes de conduites. Le comité technique Unicode a autorisé qu'on apporte une modification modifie aux *Compositions exclues* afin de remédier à cette omission. Le comité directeur du consortium Unicode a également approuvé ce changement. Les raisons de cette décision exceptionnelle sont les suivantes :

- Cette omission avait été signalée pendant la période de révision publique pour Unicode 3.0.
- Selon nos organes de liaison (particulièrement l'IETF et le W3C), aucun texte normatif ne se référait à la Normalisation Unicode 3.0 ; bien que de prochaines normes devraient sous peu faire référence à Unicode 3.1.
- YOD HIRIK appartient à une classe de caractères (les formes de présentation hébraïques situées dans l'intervalle U+FB1D .. U+FB4E) qui ont fait l'objet d'un traitement identique par le comité technique d'Unicode pendant toutes les discussions et examens relatifs à la Normalisation. Tous les autres caractères de cette classe se sont vus inclus sans exception parmi les Compositions exclues.
- YOD HIRIK est un caractère extrêmement rare. La proportion de données qui incluent actuellement ce caractère est infime par rapport à tous les textes informatisés. Même si la mise à niveau des mises en œuvre pouvait prendre quelque temps, ce changement n'entraîne dans la pratique aucun problème significatif de rétrocompatibilité.

À l'avenir, il ne sera plus admis aucun autre changement aux normalisations qui pourrait affecter la rétrocompatibilité, car aucun autre caractère ne remplira ces conditions.