# I
# Qu'est ce que ConTEXt ?
# Comment travailler avec ?

# Chapitre 1
# ConTEXt : une vue d'ensemble



## 1.1 Qu'est-ce que ConTEXt ?

ConTEXt est un *système de composition*, c'est à dire un ensemble complet d'outils visant à donner à l'utilisateur un contrôle le plus complet précis possible sur la présentation et l'apparence d'un document électronique destiné à être imprimé sur papier ou affiché à l'écran. Ce chapitre expliquera ce que cela signifie. Mais d'abord, mettons en évidence certains des caractéristiques de ConTEXt.

- Il existe deux versions de ConTEXt appelées respectivement Mark II et Mark IV. ConTEXt Mark II est *gelé*, c'est-à-dire qu'il est considéré comme finalisé, qu'aucun changement ou nouvelle fonctionnalité ne devrait y être introduit. Une nouvelle version ne sera publiée que si un bogue est détecté et doit être corrigé. ConTEXt Mark IV, en revanche, est toujours en développement, de nouvelles versions intègrent périodiquement des améliorations et fonctionnalité supplémentaires. Mais, bien que toujours en cours de développement, c'est un langage très mature, dans lequel les changements introduits par les nouvelles versions sont très subtils et n'affectent que le fonctionnement de bas niveau du système. Pour l'utilisateur moyen

ces changements sont totalement transparents, c'est comme s'ils n'existaient pas. Pour finir, bien que les deux versions aient beaucoup en commun, il y a aussi quelques incompatibilités entre elles.[1] Cette introduction se concentre donc uniquement sur le ConTEXt Mark IV.

- ConTEXt est un logiciel libre. Le programme lui-même (c'est-à-dire l'ensemble des outils logiciels qui composent ConTEXt) est distribué sous la *Licence Publique Générale GNU*. La documentation est fournie sous une licence *Creative Commons* qui vous permet de la copier et de la distribuer librement.

- ConTEXt n'est pas un programme de traitement de texte ou d'édition de texte, mais un ensemble d'outils conçus pour *transformer* un texte que nous avons précédemment écrit avec notre éditeur de texte préféré. Par conséquent, lorsque nous travaillons avec ConTEXt :

  – Nous commençons par écrire un ou plusieurs fichiers texte avec n'importe quel éditeur de texte.

  – Dans ces fichiers, en plus du texte qui constitue le contenu réel du document, il y a une série d'instructions, instructions qui indiquent à ConTEXt à quoi doit ressembler le document final généré à partir des fichiers texte originaux. L'ensemble complet des instructions ConTEXt constitue en fait un *langage* ; et puisque ce langage permet de *programmer* la transformation typographique d'un texte, on peut dire que ConTEXt est un *langage de programmation typographique*.

  – Une fois les fichiers sources écrits, ils seront traités par un programme (également appelé « `context` »[1]), qui, à partir de ceux-ci, générera un fichier PDF prêt à être envoyé à une imprimante ou à être affiché à l'écran.

- Dans ConTEXt, nous devons donc faire la différence entre le document que nous rédigeons et le document que ConTEXt génère. Pour éviter tout doute, dans cette introduction, j'appellerai *fichier source*

[1] ConTEXt désigne donc à la fois un langage et un programme (ainsi qu'un ensemble d'autres outils formant le système complet). Par conséquent, dans un texte comme cette introduction, se pose le problème de devoir parfois faire la distinction entre les deux aspects. J'ai donc adopté la convention typographique consistant à écrire «ConTEXt» avec son logo (ConTEXt) lorsque je veux me référer exclusivement à la langue, ou indistinctement à la langue et au programme. Et lorsque je veux me référer exclusivement au programme j'écrirai « `context` » tout en minuscules et avec une police de caractères à espacement constant, typique des terminaux d'ordinateur et des machines à écrire, que j'utiliserai aussi pour les exemples et pour faire référence aux instructions du langage.

le document texte qui contient les instructions de formatage, et *document final* le fichier PDF généré par ConTEXt à partir du fichier source.

Dans la suite, les points fondamentaux ci-dessus seront développés un peu plus.

## 1.2 La composition typographique de document

Ecrire un document (livre, article, chapitre, brochure, dépliant, imprimé, affiche...) et le mettre en page, dit également le composer, sont deux activités très différentes.

L'écriture du document c'est sa rédaction, qui est effectuée par l'auteur, qui décide de son contenu et de sa structure. Le document créé directement par l'auteur, tel qu'il l'a écrit, s'appelle un *manuscrit*. Le manuscrit, par sa nature même, n'est accessible qu'à l'auteur et aux personnes à qui l'auteur permet de le lire. Sa diffusion au-delà de ce cercle intime nécessite que le manuscrit soit *publié*. De nos jours, publier quelque chose — au sens étymologique de «rendre accessible au public» — est aussi simple que de le mettre sur l'internet, à la disposition de quiconque peut le localiser et souhaite le lire. Mais jusqu'à une date relativement récente, l'édition était un processus qui impliquait des coûts, dépendait de certains professionnels spécialisés dans ce domaine et n'était accessible qu'aux manuscrits qui, en raison de leur contenu ou de leur auteur, étaient considérés comme particulièrement intéressants. Et aujourd'hui encore, nous avons tendance à réserver le mot *publication* au type de *publication professionnelle* par lequel le manuscrit subit une série de transformations de son apparence visant à améliorer la *lisibilité du document*. C'est cette série de transformations qui est appelée *composition* ou encore *composition typographique*.

L'objectif de la composition est —- en général, et en laissant de côté les textes publicitaires qui cherchent à attirer l'attention du lecteur — de réaliser des documents avec une *lisibilité* maximale, entendue comme la qualité d'un texte imprimé qui invite à la lecture, ou qui la facilite, et qui

fait que le lecteur s'y sente à l'aise. De nombreux aspects y contribuent ; certains, bien sûr, sont liés au *contenu* du document (qualité, clarté, standardisation...), mais d'autres dépendent de questions telles que le[1] type et la taille de la police utilisée, la répartition des espaces blancs sur la page, la séparation visuelle entre les paragraphes, etc., on encore d'autres outils, moins graphiques ou visuels, telles que l'existence ou non dans le document de certaines aides à la lecture comme les en-têtes ou les pieds de page, les index, les glossaires, les caractères gras, les titres dans les marges, etc. On pourrait appeler «art de la composition» ou «art de l'impression» la connaissance et la manipulation correcte de toutes les ressources dont dispose un compositeur, un imprimeur.

[1] Par une convention assez ancienne, on fait une distinction entre les logiciels d'édition de texte et les logiciels de *traitement de texte*. Les premiers manipulent des fichiers de texte brut, et les seconds des fichiers de texte au format binaire permettant une plus grande complexité.

Historiquement, et jusqu'à l'avènement des ordinateurs, les tâches et les rôles du rédacteur et du compositeur sont restés clairement différenciés. L'auteur écrivait à la main ou, depuis le milieu du XIX$^{e}$ siècle, sur une «machine à écrire» dont les ressources typographiques étaient encore plus limitées que celles de l'écriture manuscrite ; puis il remettait ses originaux à l'éditeur ou à l'imprimeur qui se chargeait de les transformer pour en tirer le document imprimé.

De nos jours, grâce à l'informatique, il est plus facile pour l'auteur lui-même de définir la composition jusqu'aux moindres détails. Mais pour autant les qualités d'un bon auteur ne sont pas les mêmes que celles d'un bon compositeur. L'auteur doit avoir une bonne connaissance du sujet traité, savoir le structurer, l'exposer avec clarté, avec créativité, avec un rythme. etc. Le compositeur typographe doit avoir une bonne connaissance de l'environnement graphique et conceptuel à sa disposition, un goût de l'esthétique pour les utiliser harmonieusement, de façon cohérente avec le sujet traité, avec les tendances du moment.

Avec un bon logiciel de traitement de texte [1], il est possible d'obtenir une composition raisonnablement bonne. Mais les traitements de texte ne sont généralement pas conçus pour la composition et leurs résultats, même s'ils sont corrects, ne sont pas comparables à ceux obtenus avec d'autres outils spécifiquement conçus pour contrôler la composition des documents. En fait, les traitements de texte sont l'évolution des machines à écrire, et leur utilisation, dans la mesure où ces outils masquent la différence entre la rédaction du texte (la paternité) et

sa composition, tend à produire des textes parfois moins structurés et moins bien optimisés typographiquement.

Au contraire, les outils tels que ConTEXt sont l'évolution de l'imprimerie ; ils offrent beaucoup plus de possibilités de composition et, surtout, il n'est pas possible d'apprendre à les utiliser sans acquérir également, en cours de route, de nombreuses notions liées à la composition, contrairement aux traitements de texte, qui peuvent être utilisés pendant de nombreuses années sans apprendre un seul mot de typographie.

## 1.3 Les langages de balisage

Avant l'arrivée de l'informatique, comme je l'ai déjà dit, l'auteur préparait son manuscrit à la main ou à la machine à écrire et le remettait à l'éditeur ou à l'imprimeur, qui était chargé de transformer le manuscrit en texte final imprimé. Bien que l'auteur soit relativement peu intervenu dans cette transformation, il l'a fait dans une certaine mesure, par exemple en indiquant que certaines lignes du manuscrit étaient les titres de ses différentes parties (chapitres, sections...) ; ou en indiquant que certains fragments devaient être mis en valeur typographiquement d'une certaine manière. Ces indications étaient faites par l'auteur dans le manuscrit lui-même, parfois expressément, et d'autres fois au moyen de certaines conventions qui, avec le temps, se sont développées ; ainsi, par exemple, les chapitres commençaient toujours sur une nouvelle page, en insérant plusieurs lignes vierges avant le titre, en le soulignant, en l'écrivant en majuscules ; ou en encadrant le texte à mettre en valeur entre deux soulignements, en augmentant l'indentation d'un paragraphe, etc.

L'auteur, en somme, *indiquait* dans le texte original quelques éléments concernant la composition typographique du texte. L'éditeur ensuite inscrivait à son tour de nouvelles indication pour l'imprimeur, comme par exemple la police et la taille de caractères.

Aujourd'hui, dans un monde informatisé, nous pouvons continuer à faire de même pour la génération de documents électroniques, au moyen de ce que l'on appelle un *langage de balisage*. Dans ce type de

langage, on utilise une série de marques ou d'indications ou encore de *balises* que le programme traitant le fichier qui les contient sait interpréter. Le langage de balisage le plus connu au monde aujourd'hui est sans doute le HTML, car la plupart des pages web sont basées sur ce langage. Un fichier HTML contient le texte d'une page web, ainsi qu'une série de marques qui indiquent au programme de navigation avec lequel la page est chargée, comment elle doit être affichée. L'ensemble des balises HTML compréhensibles par les navigateurs web, ainsi que les instructions sur la manière et l'endroit où les utiliser, est appelé «langage HTML», qui c'est un langage de balisage. Mais en plus du HTML, il existe de nombreux autres langages de balisage ; en fait, ceux-ci sont en plein essor et ainsi, le XML, qui est le langage de balisage par excellence, est aujourd'hui absolument omniprésent et est utilisé pour presque tout : pour la conception de bases de données, pour la création de langages spécifiques, pour la transmission de données structurées, pour les fichiers de configuration d'applications, et ainsi de suite. Il existe également des langages de balisage conçus pour la conception graphique (SVG, TikZ ou MetaPost), les formules mathématiques (MathML), la musique (Lilypond et MusicXML), la finance, la géographie, etc. Il y a aussi, bien sûr, ceux destinés à la transformation typographique des textes, et parmi eux se distinguent TEX et ses dérivés.

En ce qui concerne les balises *typographiques*, qui indiquent l'apparence que doit avoir un texte, il en existe deux types, que nous pourrions distinguer comme d'un côté les *balises purement typographique* (ou encore graphiques) et de l'autre les *balises sémantiques* (ou encore conceptuelles, logiques). Les balises purement typographiques se limitent à indiquer précisément quelles ressources typographiques doivent être utilisées pour afficher un certain texte ; par exemple, lorsque nous indiquons qu'un certain texte doit être en gras ou en italique, de telle ou telle couleur. Le balisage sémantique, quant à lui, indique la fonction d'un texte donné dans l'ensemble du document, par exemple lorsque nous indiquons qu'il s'agit d'un titre, d'un sous-titre, d'une citation. En général, les documents qui utilisent de préférence ce deuxième type de balisage sont plus cohérents et plus faciles à composer, car la différence

entre la paternité et la composition y est à nouveau claire : l'auteur indique que cette ligne est un titre, ou que ce fragment est un avertissement, ou une citation ; et le compositeur décide comment mettre en valeur typographiquement tous les titres, avertissements ou citations ; ainsi, d'une part, la cohérence est garantie, puisque tous les fragments remplissant la même fonction auront la même apparence, et, d'autre part, on gagne du temps, puisque le format de chaque type de fragment ne doit être indiqué qu'une seule fois.

## 1.4 TEX et ses dérivés

TEX a été développé à la fin des années 1970 par Donald E. Knuth, professeur de théorie de la programmation à l'université de Stanford, qui l'a utilisé pour composer ses propres publications et ainsi que pour donner un exemple de *programmation littéraire*, une approche de la programmation où le code source du logiciel est systématique commenté et documenté. Avec TEX, Knuth a également développé un langage de programmation supplémentaire appelé METAFONT, pour la conception de caractères typographiques, avec lequel il a créé une police qu'il a nommée *Computer Modern*, qui, en plus des caractères habituels de toute police, comprenait également un ensemble complet de «glyphes» [1] pour l'écriture des mathématiques. Il a ajouté à tout cela quelques utilitaires supplémentaires et c'est ainsi qu'est né le système de composition appelé TEX, qui, pour sa puissance, la qualité de ses résultats, sa flexibilité d'utilisation et ses vastes possibilités, est considéré comme l'un des meilleurs systèmes informatiques pour la composition de textes. Il a été pensé pour des textes dans lesquels il y avait beaucoup de mathématiques, mais on a vite vu que les possibilités du système le rendaient adapté à tous les types de textes.

[1] En typographie, un glyphe est une représentation graphique d'un caractère, de plusieurs caractères ou d'une partie d'un caractère et est l'équivalent actuel du type d'impression (la pièce mobile en bois ou en plomb qui portait la gravure de la lettre).

En interne, il fonctionne comme la machine à écrire d'une presse à imprimer, car tout y est *boîte*. Les lettres sont contenues dans des boîtes, les blancs sont aussi des boîtes. Un mot est une boîte enfermant les boîtes de ses lettres. Une ligne est une boîte enfermant les boîtes de ses mots et des blancs entre ces mots. Un paragraphe est une boîte contenant l'ensemble des boîtes de ses lignes. Et ainsi de suite. Tout cela avec une précision extraordinaire apportée au traitement des mesures. Il suffit

de penser que la plus petite unité que TEX traite est 65,536 fois plus petite que le point typographique, avec lequel on mesure les caractères et les lignes, qui est généralement la plus petite unité traitée par la plupart des programmes de traitement de texte. Cette plus petite unité traité par TEX est d'environ 0,000005356 millimètre.

Le nom TEX vient de la racine du mot grec $\tau\acute{\epsilon}\chi\nu\eta$, écrit en lettres capitales (TÉXNH). Par conséquent, comme la dernière lettre du nom n'est pas un « X » latin, mais le « $\chi$ » grec, il faut prononcer «Tec». Ce mot grec, quant à lui, signifiait à la fois «art» et «technique», c'est pourquoi KNUTH l'a choisi comme nom pour son système. Le but de ce nom, écrit-il, «est de rappeler qu'il s'occupe principalement de manuscrits techniques de haute qualité. Elle met l'accent sur l'art et la technologie, tout comme le mot grec sous-jacent». Par convention établie par Knuth, le nom de est à écrire :

- Dans des textes formatés typographiquement, comme le présent texte, en utilisant le logo que j'ai utilisé jusqu'à présent : Les trois lettres sont en majuscules, avec le « E » central légèrement décalé vers le bas pour faciliter un rapprochement entre le « T » et le « X » ; c'est-à-dire : « TEX » . Pour rendre plus facile l'écriture d'un tel logo, Knuth a inclus dans une instruction qui l'inscrit dans le document final : TeXTeX.

  > Pour rendre plus facile l'écriture d'un tel logo, Knuth a inclus dans une instruction qui l'inscrit dans le document final : `\TeX`.

- Dans un texte non formaté (tel qu'un e-mail ou un fichier texte), le « T » et le « X » sont en majuscules, et le « e » du milieu est en minuscules ; par exemple : « TeX ».

Cette convention est suivie dans tous les dérivés de TEX qui l'incluent dans leur propre nom, comme par exemple ConTEXt, qui lorsqu'il est écrit en mode texte doit être écrit « ConTeXt ».

### 1.4.1 Les moteurs TEX

Le programme TEX est un logiciel libre : son code source est à la disposition du public et chacun peut l'utiliser ou le modifier à sa guise, à la seule condition que, si des modifications sont introduites, le résultat

ne puisse être appelé « TEX ». C'est la raison pour laquelle, au fil du temps, certaines adaptations du programme sont apparues, qui lui ont apporté différentes améliorations, et qui sont généralement appelées *moteurs TEX* (engine en anglais). En dehors du programme original, les principaux moteurs TEX sont, par ordre chronologique d'apparition pdfTEX, ε-TEX, XETEX et LuaTEX. Chacun d'entre eux est censé intégrer les améliorations de son prédécesseurs. Ces améliorations, en revanche, jusqu'à l'apparition de LuaTEX, n'ont pas affecté le langage lui-même, mais seulement les fichiers d'entrée, les fichiers de sortie, la gestion des polices et le fonctionnement de bas niveau des macros.

> La question du choix du moteur TEX à utiliser fait l'objet d'un débat animé dans l'univers TEX. Je ne m'y attarderai pas ici, car ConTEXt Mark IV ne fonctionne qu'avec LuaTEX. En fait, dans le monde de ConTEXt la discussion sur les moteurs devient une discussion sur l'utilisation de Mark II (qui fonctionne avec pdfTEXet XETEX) ou Mark IV (qui fonctionne avec LuaTEX).

### 1.4.2 Les formats dérivés de TEX

Le noyau, ou cœur, de TEX contient seulement un ensemble d'environ 300 instructions, appelées *primitives*, qui conviennent aux opérations de composition et aux fonctions de programmation très basiques. Ces instructions sont pour la plupart de très *bas niveau*, ce qui, en terminologie informatique, signifie qu'elles se rapprochent des opérations élémentaires de l'ordinateur, dans un langage machine peu approprié aux êtres humains, du type « déplacer ce caractère de 0,000725 millimètre vers le haut ».

Pour cette raison, KNUTH a rendu TEX extensible, c'est-à-dire disposant d'un mécanisme permettant de définir des instructions de plus haut niveau, dans un langage plus facilement compréhensibles par les êtres humains. Ces instructions, qui au moment de l'exécution sont décomposées en instructions élémentaires, sont appelées *macros*. Par exemple, l'instruction TEX qui imprime votre logo (`\TeX`), est décomposée lors de son exécution en :

```
T
\kern -.1667em
\lower .5ex
\hbox {E}
\kern -.125em
X
```


```
T
\kern -.1667em
\lower .5ex
\hbox {E}
\kern -.125em
X
```


On comprend là qu'il est beaucoup plus facile pour un être humain de comprendre et mémoriser la simple commande « `\TeX` » dont l'exécution effectue l'ensemble des opérations typographiques nécessaires à l'impression du logo.

> La différence entre les *macros* et les *primitives* n'est vraiment importante que du point de vue du développeur de TEX. Du point de vue de l'utilisateur, ce sont toutes des *instructions* ou, si vous préférez, des *commandes*. Knuth les appelait des *séquences de contrôle*.

Cette possibilité d'étendre le langage par le biais de *macros* est l'une des caractéristiques qui ont fait de TEX un outil si puissant. En fait, Knuth lui-même a conçu environ 600 macros qui, avec les 300 primitives, constituent le format appelé « Plain TEX ». Il est assez courant de confondre TEX lui-même avec Plain TEX et, en fait, presque tout ce qui est dit ou écrit sur TEX, se réfère en fait à Plain TEX. Les livres qui prétendent être sur TEX (y compris le livre fondateur « *The TEXBook* »), font en fait référence à Plain TEX ; et ceux qui pensent qu'ils manipulent directement TEX manipulent en fait Plain TEX.

Plain TEX est ce que l'on appelle dans la terminologie TEX un *format*, constitué d'un vaste ensemble de macros, ainsi que de certaines règles syntaxiques sur la manière et la façon de les utiliser. En plus de Plain TEX, d'autres formats ont été développés au fil du temps, notamment LATEX un vaste ensemble de macros pour TEX conçu en 1985 par Leslie Lamport, qui est probablement le dérivé de TEX le plus utilisé dans le

monde universitaire, technologique et mathématique. ConTEXt est (ou a commencé à être), de même que LATEX un format dérivé de TEX.

Normalement, ces *formats* sont accompagnés d'un programme qui charge dans la mémoire de l'ordinateur les macros qui les composent avant d'appeler « `tex` » (ou l'un des autres moteurs précédemment listés) pour traiter le fichier source. Mais bien que tous ces formats exécute finalement TEX, comme chacun possède ses instructions et ses règles syntaxiques spécifiques, du point de vue de l'utilisateur, nous pouvons les considérer comme des *langages différents*. Ils sont tous inspirés de TEX, mais différents de TEX, et différents les uns des autres.

## 1.5 ConTEXt

En fait, si ConTEXt a commencé comme un *format* de TEX, aujourd'hui il est beaucoup plus que cela. ConTEXtcomprend :

1. Un très large ensemble de macros de TEX. Si Plain TEX se compose d'environ 900 instructions, il en compte près de 3500 ; et si l'on ajoute les noms des différentes options que ces commandes prennent en charge, on parle d'un vocabulaire d'environ 4000 mots. Le vocabulaire est aussi large car la stratégie de ConTEXtpour faciliter son apprentissage est d'inclure de nombreux synonymes des commandes et des options.

   > Ce qui est prévu, pour obtenir un certain effet, c'est de fournir à l'utilisateur l'ensemble des façons dont celui ci pourrait chercher à appeler cet effet. Par exemple, pour obtenir simultanément un caractère gras (en anglais *bold*) et italique (en anglais *italic*), ConTEXtpropose trois instructions identiques en terme de résultat : `\bi`, `\italicbold` y `\bolditalic`.

2. Un autre ensemble assez complet de macros pour MetaPost, un langage de programmation graphique dérivé de METAFONT, qui, lui-même, est le langage de conception de caractères que KNUTH a co-développé avec TEX.

3. Plusieurs *scripts* développés en Perl (les plus anciens), Ruby (certains également anciens et d'autres moins) et Lua (les plus récents).

4. Une interface qui intègre TEX, MetaPost, Lua et XML, permettant d'écrire et de traiter des documents dans n'importe lequel de ces langages, ou qui mélangent des éléments de certains d'entre eux.

   Vous n'avez pas compris grand-chose à l'explication ci-dessus ? Ne vous inquiétez pas. J'ai utilisé beaucoup de jargon informatique et mentionné beaucoup de programmes et de langages. Mais il n'est pas nécessaire de savoir d'où viennent les différents composants pour les utiliser. L'important, à ce stade de l'apprentissage, est de garder à l'esprit qu'il intègre de nombreux outils d'origines diverses qui forment un *système de composition typographique*.

C'est en raison de cette intégration d'outils d'origines différentes que l'on caractérise ConTEXt de « technologie hybride » dédié à la composition typographique de documents. C'est également ce qui fait de ConTEXt un système extraordinairement avancé et puissant.

Mais bien que ConTEXt soit bien plus qu'un ensemble de macros pour TEX, ses fondamentaux restent basés sur TEX, et donc ce document, qui se veut n'être qu'une *introduction*, se concentre sur cet aspect.

ConTEXt en revanche est beaucoup plus moderne que TEX. Lorsque TEX a été conçu, l'informatique commençait à peine à émerger, et on était encore loin d'entrevoir ce que serait (ce qui allait devenir) l'Internet et le monde du multimédia. En ce sens, ConTEXt intègre naturellement certains éléments qui ont toujours constitué une sorte de corps étranger, tels que l'inclusion de graphiques externes, le traitement des couleurs, les hyperliens dans les documents électroniques, l'hypothèse d'un format de papier adapté d'un affichage sur écran, etc.

### 1.5.1 Une brève histoire de ConTEXt

ConTEXt est né vers 1991. Il a été créé par Hans Hagen et Ton Otten au sein d'une société néerlandaise de conception et de composition de documents appelée « *Pragma Advanced Document Engineering* », souvent

abrégée en Pragma ADE. Il s'agissait au départ d'un ensemble de macros TEX en néerlandais, officieusement connu sous le nom de *Pragmatex*, et destiné aux employés non techniques de l'entreprise, qui devaient gérer les nombreux détails de la mise en page des documents à éditer, et qui n'étaient pas habitués à utiliser des langages de balisage et des interfaces dans une autre langue que le néerlandais.

La première version de ConTEXt a donc été écrite en néerlandais. L'idée était de créer un nombre suffisant de macros avec une interface uniforme et cohérente. Vers 1994, le *paquet* était suffisamment stable pour qu'un manuel d'utilisation soit écrit en néerlandais, et en 1996, à l'initiative de Hans Hagen, le nom « ConTEXt » a été utilisé pour s'y référer. Ce nom est censé signifier « Texte avec TEX » (en utilisant la préposition latine "con" qui a la même signification qu'en espagnol), mais il joue en même temps avec le terme « Contexte », qui en néerlandais (comme en anglais) s'écrit « context ». Derrière ce nom, il y a donc un triple jeu de mots entre « TEX », « texte » et « contexte ».

> Par conséquent, bien que ConTEXt soit dérivé de TEX (prononcé « Tec »), il ne devrait pas être prononcé « Contect » afin de ne pas perdre ce jeu de mots.

L'interface a commencé à être traduite en anglais vers 2005, donnant lieu à la version connue sous le nom de ConTEXt Mark II, où le « II » s'explique par le fait que dans l'esprit des développeurs, la version « I » est la version précédente en néerlandais, même si elle n'a jamais vraiment été appelée ainsi. Après la traduction de l'interface en anglais, le système a commencé à être utilisé en dehors des Pays-Bas, et l'interface a été traduite dans d'autres langues européennes comme le français, l'allemand, l'italien et le roumain. La documentatino « officielle » de ConTEXt est généralement écrite sur la version anglaise, et c'est donc sur cette version que nous travaillons dans ce document, même si l'auteur de ce document (c'est-à-dire moi) est plus à l'aise en espagnol qu'en anglais.

Dans sa version initiale, ConTEXt Mark II fonctionnait avec le *moteur TEX* pdfTEX. Plus tard, lorsque le nouveau moteur XƎTEX est apparu, ConTEXt Mark II a été modifié pour en permettre l'utilisation, qui présentait de nombreux avantages par rapport à pdfTEX. Des années plus tard encore, lorsque le moteur LuaTEXa été développé, il a été décidé

de reconfigurer le fonctionnement interne de ConTEXt Mark II pour intégrer toutes les nouvelles possibilités offertes par ce dernier moteur. C'est ainsi qu'est né ConTEXt Mark IV, qui a été présenté en 2007, immédiatement après la présentation de LuaTEX. La décision d'adapter ConTEXt à LuaTEXa très probablement été influencée par le fait que deux des trois principaux développeurs de ConTEXt, Hans Hagen et Taco Hoekwater, font également partie de l'équipe de développement de LuaTEX. Par conséquent, ConTEXt Mark IV et LuaTEX sont nés simultanément et ont été développés à l'unisson. Il existe une synergie entre ConTEXt et LuaTEX et qui n'existe avec aucun autre dérivé de TEX ; et je ne pense pas qu'aucun d'entre eux ne profite des possibilités de LuaTEX comme ConTEXt le fait.

Entre Mark II et Mark IV, il existe de nombreuses différences, bien que la plupart d'entre elles soient *internes*, c'est-à-dire qu'elles concernent le fonctionnement de la macro à un bas niveau, de sorte que du point de vue de l'utilisateur, la différence n'est pas perceptible : le nom et les paramètres de la macro sont les mêmes. Il existe cependant quelques différences qui affectent l'interface et vous obligent à faire les choses différemment selon la version avec laquelle vous travaillez. Ces différences sont relativement peu nombreuses, mais elles affectent des aspects très importants comme, par exemple, l'encodage du fichier d'entrée, ou la gestion des polices installées dans le système.

Cependant, il serait apprécié qu'il existe quelque part un document expliquant (ou listant) les différences significatives entre Mark II et Mark IV. Dans le wiki de ConTEXt, par exemple, il existe parfois *deux syntaxes* (souvent identiques) pour chaque commande. Je suppose que l'une est la version Mark II et l'autre la version Mark IV ; et à deviner, je suppose également que la première version est la version Mark II. Mais en pratique rien n'est indiqué à ce sujet sur le wiki.

Le fait que, pour les utilisateur, les différences au niveau du langage soient relativement peu nombreuses, signifie que dans de nombreux cas, plutôt que de parler de deux versions, nous parlons de deux « saveurs » de ConTEXt. Mais qu'on les appelle d'une manière ou d'une autre, le fait est qu'un document préparé pour Mark II peut ne pas être compatible d'une compilation avec Mark IV et vice versa ; et si le document mélange les deux versions, il est fort probable qu'il ne se compilera bien avec aucune d'entre elles ; ce qui signifie que l'auteur du

fichier source doit commencer par décider s'il l'écrit pour Mark II ou Mark IV.

> Si nous avons a travailler avec différentes versions de ConTEXt, une bonne astuce pour facilement distinguer les versions des fichiers sources consiste à utiliser une extension différente dans le nom des fichiers. Ainsi, par exemple, mes fichiers écrits pour Mark II sont nommés « `.mkii` » et ceux écrits pour Mark IV sont nommés « `.mkiv` ». Il est vrai que ConTEXt s'attend à ce que tous les fichiers sources aient l'extension « `.tex` », mais vous pouvez changer l'extension tant que lorsque vous invoquez un fichier, vous indiquez explicitement son extension, si elle n'est pas celle par défaut.

La distribution de ConTEXt que vous installez à partir de leur wiki, « ConTEXt Standalone », inclut les deux versions, et pour éviter toute confusion —je suppose— propose une commande distincte pour compiler avec chacune d'entre elles. Mark II compile avec la commande « `texexec` » et Mark IV avec la commande « `context` ».

> En réalité, aussi bien « `context` » que « `texexec` » sont des *scripts* qui lancent, avec différentes options, « `mtxrun` » qui, à son tour, est un *script* Lua.

A ce jour, Mark II est gelé et Mark IV est toujours en cours de développement, ce qui signifie que les nouvelles versions de Mark II ne sont publiées que lorsque des bogues ou des erreurs sont détectés, tandis que les nouvelles versions de Mark IV sont publiées régulièrement ; parfois même deux ou trois par mois ; bien que dans la plupart des cas, ces « nouvelles versions » n'introduisent pas de changements notables dans le langage, et se limitent à améliorer d'une manière ou d'une autre l'implémentation de bas niveau d'une commande, ou à mettre à jour l'un des nombreux manuels qui sont inclus dans la distribution. Néanmoins, si nous avons installé la version de développement — qui est celle que je recommande et celle qui est installée par défaut avec « ConTEXt Standalone » —, il est approprié de mettre à jour notre installation de temps en temps (voir l'annexe ?? concernant la mise à jour de la version installée de « ConTEXt Standalone »).

### LMTX et autres implémentations alternatives de Mark IV

> Les développeurs de ConTEXt sont soucieux de la qualité du logiciel et n'ont cessé de faire évoluer Mark IV ; de nouvelles versions sont testées et expérimentées. Celles-ci, en général, ne diffèrent de Mark IV que sur

très peu de points, et ne présentent pas d'incompatibilité de compilation comme cela existe entre Mark IV et Mark II, ce qui traduit la maturité du langage du point de vue utilisateur.

Ainsi, quelques variantes de Mark IV ont été développées, appelées respectivement Mark VI, Mark IX et Mark XI. Je n'ai pu trouver qu'une petite référence à Mark VI dans le wiki de ConTEXt où il est indiqué que sa seule différence avec Mark IV est la possibilité de définir des commandes en assignant aux paramètres non pas un nombre, comme c'est traditionnel dans TEX, mais un nom, comme cela se fait habituellement dans presque tous les langages de programmation.

Plus important que ces petites variantes —je pense— est l'apparition dans l'univers de ConTEXt d'une nouvelle version, appelée LMTX, nom qui est un acronyme pour luametaTEX : un nouveau *moteur* de TEX qui est une version simplifiée et optimisée de LuaTEX, développé en vue d'économiser les ressources informatiques et d'offrir une solution TEX aussi minimaliste que possible ; c'est-à-dire que LMTX nécessite moins de place sur disque dur, moins de mémoire et moins de puissance de traitement que ConTEXt Mark IV.

LMTX a été présenté au printemps 2019 et l'on suppose qu'il n'impliquera aucune altération externe du langage Mark IV. Pour l'auteur du document, il n'y aura aucune différence dans la conception ; mais au moment de la compilation, vous pouvez choisir entre compiler avec LuaTEX, ou compiler avec luametaTEX. Une procédure pour attribuer un nom de commande différent à chacune des installations (section ??) est expliquée dans l'annexe ??, relative à l'installation de ConTEXt.

### 1.5.2 ConTEXt versus LATEX

Comme le format dérivé de TEX le plus populaire est LATEX la comparaison entre celui-ci et ConTEXt est inévitable.

Il s'agit bien sûr de langages différents mais, d'une certaine manière, liés par leur origine commune TEX ; la parenté est donc similaire à celle qui existe entre, par exemple, l'espagnol et le français : des langues qui partagent une origine commune (le latin) qui utilise des syntaxes *similaires* et de nombreux mots se correspondants assez directement. Mais au-delà de cet air de famille, LATEX et ConTEXt diffèrent dans leur philosophie et leur mise en œuvre, puisque les objectifs initiaux de l'un et de l'autre sont, en quelque sorte, contradictoires.

LATEX vise à faciliter l'utilisation de TEX, en éloignant l'auteur des détails typographiques spécifiques pour l'inciter à se concentrer sur le contenu, et laisser les détails de la composition entre les mains de LATEX. En d'autres termes, la simplification de l'utilisation de TEX est obtenue au prix d'une limitation de son immense flexibilité, par la prédéfinition de nombreux formats de base et la réduction du nombre de choix typographiques que l'auteur doit déterminer.

A l'opposé de cette philosophie, ConTEXt est ne au sein d'une entreprise dédiée à la composition de documents. Par conséquent, loin d'essayer d'isoler l'auteur des détails de la composition, ce qu'il tente de faire, c'est de lui donner un contrôle absolu et complet sur ceux-ci. Pour ce faire, ConTEXt fournit une interface homogène et cohérente qui reste beaucoup plus proche de l'esprit original TEX que LATEX.

Cette différence de philosophie et d'objectifs fondamentaux se traduit à son tour par une différence de mise en œuvre. Parce que LATEX, qui tend à simplifier au maximum, n'a pas besoin d'utiliser toutes les ressources de TEX. Son cœur est, d'une certaine manière, assez simple. Par conséquent, lorsque vous souhaitez étendre ses possibilités, vous devez construire un *paquet*. Cet *ensemble de paquets* associé à LATEXest à la fois une force et une faiblesse : une force, car l'énorme popularité de LATEX, ainsi que la générosité de ses utilisateurs, impliquent que pratiquement tous les besoins qui se présentent ont déjà été soulevés par quelqu'un, et qu'il existe un paquet qui y réponde ; mais aussi une faiblesse, car ces paquets sont souvent incompatibles entre eux, et leur syntaxe n'est pas toujours homogène, ce qui signifie que l'utilisation de LATEX exige une plongée continue dans les milliers de paquets existants pour trouver ceux dont nous avons besoin et les faire fonctionner ensemble.

Contrairement à la simplicité du noyau de LATEX et son extensibilité par le biais de paquets, ConTEXt est conçu pour intégrer et rendre accessibles toutes — ou presque toutes — les possibilités typographiques de TEX, de sorte que sa conception est beaucoup plus monolithique, mais, en même temps, il est aussi plus modulaire : le noyau ConTEXt permet de faire presque tout et il est garanti qu'il n'y aura pas d'incompatibilités entre les différentes commandes, il n'y a pas besoin de rechercher

les extensions dont vous avez besoin (elles sont déjà présentes), et la syntaxe du langage est homogène entre les différents commande.

Il est vrai que ConTEXt propose des *modules* d'extension dont on pourrait considérer qu'ils ont une fonction similaire à celle des paquets de LATEX, mais la vérité est que la fonction des deux est très différente : les modules de ConTEXt sont conçus exclusivement pour accueillir des fonctionnalités supplémentaires qui, parce qu'ils sont en phase expérimentale, n'ont pas encore été incorporés dans le noyau, ou pour permettre à des développeurs en dehors de l'équipe de développement de ConTEXt de les proposer.

Je ne pense pas que l'une de ces deux *philosophies* puisse être considérée comme préférable à l'autre. La réponse dépend plutôt du profil de l'utilisateur et de ce qu'il souhaite. Si l'utilisateur ne veut pas s'occuper de questions typographiques, mais simplement produire des documents standardisés de très haute qualité, il serait probablement préférable pour lui d'opter pour un système comme LATEX ; au contraire, l'utilisateur qui aime expérimenter, ou qui a besoin de contrôler chaque détail de ses documents, ou qui doit concevoir un design spécial pour un certain document, ferait probablement mieux d'utiliser un système comme ConTEXt, où l'auteur dispose de tous les contrôle ; avec le risque, bien sûr, qu'il ne sache pas correctement l'utiliser.

### 1.5.3 La logique de travail avec ConTEXt

Lorsque nous travaillons avec ConTEXt, nous commençons toujours par écrire un fichier texte (que nous appellerons *fichier source*), dans lequel nous inclurons, en plus du contenu de notre document final à proprement parler, les instructions (en langage ConTEXt) qui indiquent exactement comment nous voulons que le document soit composé : quel aspect général nous voulons donner à ses pages et paragraphes, quelles marges nous souhaitons appliquer à certains paragraphes spéciaux, quelles types de police doit être utilisé, quels fragments souhaitons nous afficher avec une police différente, etc. Une fois que nous avons écrit le fichier source, depuis un terminal, nous executerons le programme « `context` », qui le traitera et, à partir de celui-ci, générera un fichier différent, dans lequel le contenu de notre document aura été formaté selon les instructions qui étaient incluses dans le fichier

source. Ce nouveau fichier peut être envoyé à l'imprimante, affiché à l'écran, hébergé sur Internet ou distribué à nos contacts, amis, clients, professeurs, étudiants... bref, à tous ceux pour qui nous avons écrit le document.

C'est-à-dire que lorsqu'il travaille avec ConTEXt, l'auteur agit sur un fichier dont l'apparence n'a rien à voir avec celle du document final : le fichier avec lequel l'auteur travaille directement est un fichier texte qui n'est pas formaté typographiquement. À cet égard, son fonctionnement est très différent de celui des programmes dits de *traitement de texte*, qui affichent l'aspect final du document édité au fur et à mesure de sa saisie. Pour ceux qui sont habitués aux *traitements de texte*, le fonctionnement de l'application peut sembler étrange au début, et il peut même falloir un certain temps pour s'y habituer. Cependant, une fois que vous vous y serez habitué, vous comprendrez que cette autre façon de travailler, faisant la différence entre le fichier de travail et le résultat final, est en fait un avantage pour de nombreuses raisons, parmi lesquelles je soulignerai, sans ordre particulier, les suivantes :

1. car les fichiers texte sont plus « légers » à manipuler que les fichiers binaires des traitements de texte et que leur édition nécessite moins de ressources informatiques ; ils sont moins sujets à la corruption et ne deviennent pas illibles si la version du programme avec lequel ils ont été créés change. Ils sont également compatibles avec n'importe quel système d'exploitation et peuvent être édités avec de nombreux éditeurs de texte, de sorte que pour travailler avec eux, il n'est pas nécessaire de disposer d'un logiciel d'édition particulier : n'importe quel autre programme d'édition de texte fera l'affaire, et chaque système d'exploitation informatique propose un voire des programmes d'édition de texte.

2. car la différenciation entre le document de travail et le document final permet de distinguer ce qui est le contenu réel du document de ce qui sera son apparence, permettant à l'auteur de se concentrer sur le contenu dans la phase de création, et sur l'apparence dans la phase de composition.

3. car il vous permet de modifier très rapidement et très précisément l'apparence du document, puisque celle-ci est déterminée par des commandes facilement identifiables.

4. car cette facilité à changer l'apparence, d'autre part, permet de générer facilement plusieurs versions différentes à partir d'un seul contenu : par exemple une version optimisée pour l'impression sur papier, et une autre pour l'affichage sur écran, ajustée à la taille de celui-ci et, peut-être, incluant des hyperliens qui n'ont pas d'utilité dans un document imprimé sur papier.

5. car il est également facile d'éviter les erreurs typographiques courantes dans les traitements de texte comme, par exemple, l'extension de l'italique au-delà du dernier caractère à utiliser, les erreurs d'application de style..

6. car, puisque le fichier de travail ne sera pas distribué et qu'il est « pour nos yeux seulement », il est possible d'incorporer des annotations et des observations, des commentaires et des avertissements pour nous-mêmes, pour des révisions ou des versions futures, avec la tranquillité d'esprit de savoir qu'ils n'apparaîtront pas dans le fichier formaté qui sera distribué.

7. car la qualité que l'on peut obtenir en traitant simultanément l'ensemble du document est bien supérieure à celle que l'on peut obtenir avec un programme qui doit prendre des décisions typographiques à la volée, au fur et à mesure de la rédaction du document.

8. etcétéra.

Tout cela signifie que, d'une part, lorsque l'on travaille avec ConTEXt, une fois que l'on a pris le coup de main, on est plus efficace et productif, et que, d'autre part, la qualité typographique que l'on obtiendra est bien supérieure à celle que l'on obtiendrait avec les *logiciels de traitement de texte*. Et s'il est vrai que, en comparaison, ces derniers sont plus faciles à utiliser, en réalité ils ne le sont *pas beaucoup*. Car s'il est vrai que ConTEXtse compose, comme je l'ai déjà dit, d'environ 3500 instructions, un utilisateur normal n'a pas à toutes les connaître. Pour faire ce que l'on fait habituellement avec les traitements de texte, il suffira de connaître les instructions qui permettent d'indiquer la structure du document, quelques instructions relatives aux ressources typographiques courantes, comme le gras ou l'italique, et, éventuellement, comment générer une liste, ou une note de bas de page. Au total, pas plus de 15

ou 20 instructions nous permettront de faire presque toutes les choses que l'on fait avec un traitement de texte. Le reste des instructions nous permet de faire différentes choses qui, normalement, sont très difficiles voire impossibles à faire avec un logiciel de traitement de texte. Ainsi, si l'apprentissage de ConTEXt est plus difficile que celui d'un logiciel de traitement de texte, c'est parce que l'on peut faire beaucoup plus de choses avec.

### 1.5.4 Obtenir de l'aide sur ConTEXt

Tant que nous sommes des débutants, le meilleur endroit pour trouver de l'aide sur ConTEXt est sans aucun doute son wiki, qui regorge d'exemples et dispose d'un bon moteur de recherche, même s'il nécessite bien sûr de bien comprendre l'anglais. Nous pouvons aussi chercher de l'aide sur Internet, mais ici le jeu de mots sur lequel repose ConTEXt nous jouera un sale tour car une recherche d'informations sur « contexte » renverrait des millions de résultats et la plupart d'entre eux n'auraient aucun rapport avec ce que nous recherchons. Pour rechercher des informations sur ConTEXt, vous devez ajouter quelque chose au nom « context » ; par exemple, « tex », « luatex », « Mark IV » , « Hans Hagen » (un des créateurs de ConTEXt), « Pragma ADE », ou quelque chose de similaire (par exemple une autre commande souvent utilisée dans le cas de figure qui vous préoccupe). Il peut également être utile de rechercher des informations par le nom wiki : « contextgarden ».

Après en avoir appris un peu plus sur ConTEXt, et si l'on maîtrise bien l'anglais, on peut consulter l'un des nombreux documents inclus dans « ConTEXt Standalone » ou demander de l'aide :

- soit sur TeX – LaTeX Stack Exchange et en particulier les questions tagguées « ConTEXt »
- soit sur la liste de diffusion propre à ConTEXt NTG-context et son moteur de recherche.

Cette dernière liste diffusion implique les personnes les plus compétents sur ConTEXt, mais les règles d'une bonne éducation de « cybercitoyen » exigent qu'avant de poser une question, on ait essayé par tous les moyens de trouver la réponse par soi-même dans les documentations déjà existantes.

# Chapitre 2
# Notre premier fichier source

**Table of Contents:** 

Ce chapitre est consacré à la mise en oeuvre de notre première expérience. Il expliquera la structure de base d'un document ConTEXt ainsi que les meilleures stratégies pour faire face aux éventuelles erreurs.

## 2.1 Préparation de l'expérience outils nécessaires

Pour écrire et compiler un premier fichier source, nous devrons avoir les outils suivants installés sur notre système.

1. **un éditeur de texte** pour écrire notre fichier de test. Il existe de nombreux éditeurs de texte et il est difficilement concevable qu'un système informatique n'en ait pas déjà un d'installé. Nous pouvons utiliser n'importe lequel d'entre eux : il existe des systèmes simples, d'autres complexes, des puissants, d'autres simples, des payants, des gratuits, des spécialisés pour TEX, des généralistes, etc. Si nous avons l'habitude d'utiliser un éditeur spécifique, il est préférable de poursuivre avec lui ; si nous n'avons pas l'habitude de travailler avec des éditeurs de texte, mon conseil est, dans un premier temps, de choisir un éditeur simple, afin de ne pas ajouter à la difficulté de

l'apprentissage de ConTEXt la difficulté d'apprendre à utiliser l'éditeur. Bien qu'il soit également vrai que, souvent, les programmes les plus difficiles à maîtriser sont aussi les plus puissants.

J'ai écrit ce texte avec GNU Emacs, qui est l'un des éditeurs généralistes les plus puissants et les plus polyvalents qui existent ; il est vrai qu'il a ses particularités et aussi ses détracteurs, mais en général il y a plus de « *Emacs Lovers* » que de « *Emacs Haters* ». Pour travailler avec des fichiers TEX ou l'un de ses dérivés, il existe une extension pour GNU Emacs, appelée AucTeX, qui fournit à l'éditeur quelques fonctionnalités supplémentaires très intéressantes, même si AucTeX est plus développé pour LATEX que pour ConTEXt. GNU Emacs en combinaison avec AucTeX peut être une bonne option si l'on ne sait pas quel éditeur choisir ; tous deux sont des programmes à code source ouvert, et ils sont disponibles pour tous les systèmes d'exploitation. En fait, dire que GNU Emacs est un *logiciel libre* est un euphémisme, car ce programme incarne mieux que tout autre l'esprit de ce qu'est et signifie le *logiciel libre*. Après tout, son principal développeur était Richard Stallman, fondateur et idéologue du projet GNU et de la *Free Software Foundation*.

En plus de GNU Emacs + AucTeX, *Scite* et *TexWorks* sont d'autres bonnes options si vous ne savez pas quel éditeur choisir. Le premier, bien qu'il s'agisse d'un éditeur à usage généraliste, non conçu spécifiquement pour travailler avec des fichiers ConTEXt, est facilement personnalisable et, comme c'est l'éditeur généralement utilisé par les développeurs de ConTEXt « ConTEXt Standalone » contient les fichiers de configuration de cet éditeur conçus et utilisés par Hans Hagen lui-même. *TexWorks* est, quant à lui, un éditeur de texte rapide, spécialisé dans le traitement des fichiers TEX et de ses langages dérivés. Il est assez facile à configurer pour fonctionner avec ConTEXt et « ConTEXt Standalone » prévoit également de fournir des fichiers configurations.

Qu'il s'agisse d'un éditeur ou d'un autre, ce qu'il ne faut pas faire, c'est utiliser, comme éditeur de texte, un *logiciel de traitement de texte* tel que, par exemple, OpenOffice Writer ou Microsoft Word. Ces programmes, qui sont à mon avis trop lents et trop lourds, peuvent certes enregistrer un fichier en « texte pur », mais ils ne sont pas

conçus pour cela et nous finirions probablement par enregistrer notre fichier dans un format binaire incompatible avec ConTEXt.

2. **Une distribution ConTEXt** pour traiter notre fichier de test. S'il existe déjà une installation TEX (ou LATEX) sur votre système, il est possible qu'une version de ConTEXt soit déjà installée. Pour le vérifier, il suffit d'ouvrir un terminal et de taper dans celui-ci

```
1 $ context --version
```

```
$ context --version
```

> **NOTA** ceux pour qui l'utilisation du terminal est nouvelle, les deux premiers caractères que j'ai indiqué ( « $ ») n'ont pas à être tappé dans le terminal par l'utilisateur. Je les utilise pour représenter ce qu'on appelle l'*invite* du terminal (le prompt en anglais), qui indique que le terminal attend nos instructions.

Si une version de ConTEXt est déjà installée, vous devriez obtenir un résultat similaire au suivant :

```
1 $ context --version
2 mtx-context     | ConTeXt Process Management 1.03
3 mtx-context     |
4 mtx-context     | main context file: /usr/share/texmf/tex/context/base/mkiv/context.mkiv
5 mtx-context     | current version: 2020.03.10 14:44
6 mtx-context     | main context file: /usr/share/texmf/tex/context/base/mkiv/context.mkxl
7 mtx-context     | current version: 2020.03.10 14:44
```

```
$ context --version
mtx-context     | ConTeXt Process Management 1.03
mtx-context     |
mtx-context     | main context file: /usr/share/texmf/tex/context/base/mkiv/context.mkiv
mtx-context     | current version: 2020.03.10 14:44
mtx-context     | main context file: /usr/share/texmf/tex/context/base/mkiv/context.mkxl
mtx-context     | current version: 2020.03.10 14:44
```

Dans la dernière ligne, nous sommes informés de la date à laquelle la version installée a été publiée. Si elle est très ancienne, nous devons le mettre à jour ou installer une nouvelle version. Je recommande d'installer la distribution appelée « ConTEXt Standalone »

dont les instructions d'installation se trouvent sur le wiki de ConTEXt. Les indications sont également incluses dans l'annexe ??.

3. **Un programme de visualisation de fichier PDF** afin de visualiser le résultat de notre expérience à l'écran. Sur les systèmes Windows et Mac OS, la visionneuse omniprésente est Adobe Acrobat Reader. Il n'est pas installé par défaut (ou ne l'était pas lorsque j'ai cessé d'utiliser Microsoft Windows, il y a plus de 15 ans). L'installation se fait la première fois que vous essayez d'ouvrir un fichier PDF, il est donc généralement déjà installé. Sur les systèmes Linux/Unix, il n'y a pas de version mise à jour d'Acrobat Reader, mais il n'est pas nécessaire non plus, car il existe littéralement des dizaines de très bons visualisateurs de PDF gratuits. De plus, dans ces systèmes, il y en a presque toujours un installé par défaut. Mon préféré, pour sa vitesse et sa facilité d'utilisation, est MuPDF ; bien qu'il ait quelques inconvénients comme, par exemple, de ne pas montrer l'index des signets, de ne pas permettre les recherches de texte qui incluent des caractères inexistants dans l'alphabet anglais (comme les voyelles accentuées ou les eñes) ou de ne pas permettre de sélectionner le texte, d'envoyer le document à l'imprimante ; c'est juste un visualisateur, mais très rapide et très confortable. Lorsque j'ai besoin de certains de ces fonctionnalités absentes de MuPDF, j'utilise généralement Okular ou qPdfView. Mais, encore une fois, la question est une affaire de goût : chacun peut choisir celui qu'il préfère.

Nous pouvons choisir l'éditeur, nous pouvons choisir le visualisateur de PDF, nous pouvons choisir la distribution ConTEXt... Bienvenue dans le monde du *logiciel libre* !

## 2.2 L'experience elle-même

### Rédaction du fichier source

Si nous disposons déjà des outils mentionnés dans la section précédente, nous devons ouvrir notre éditeur de texte et créer un fichier appelé « la-maison-sur-le-port.tex » pour notre exemple. Comme contenu du fichier, nous allons écrire ce qui suit :

```
% Première ligne du document

\mainlanguage[fr]    % Langue français

\setuppapersize[S5]  % Format du papier

\setupbodyfont       % Police = Latin Modern, 12 points
  [modern,18pt]

\setuphead           % Format des titres de chapitre
  [chapter]
  [style=\bfc]

\starttext           % Début du contenu du document

\startchapter
  [title=La maison sur le port]

Il y avait des    chansons
Les hommes        venaient y boire et rêver
Dans la maison    sur le port
Où les filles     riaient fort
Où le vin faisait chanter chanter chanter

Les pêcheurs      vous le diront
Ils y venaient    sans façon
Avant de partir   retirer leurs filets
Ils venaient      se réchauffer près de nous
Dans la maison    sur le port

\stopchapter

\stoptext            % Fin du document
```

Durant l'écriture, certains aspects n'ont aucune importance, notamment si vous ajoutez ou supprimez des espaces blancs ou des sauts de ligne. Ce qui est important, c'est que chaque mot suivant le caractère « \ » soit écrit très exactement de la même façon qu'il l'est dans l'exemple, ainsi que le contenu des crochets. Il peut y avoir des variations dans le reste.

## Encodage du fichier

Une fois le texte précédent écrit, nous enregistrons le fichier sur le disque. Cela n'est dorénavant qu'une vérification à faire, mais il faut nous assurer que l'encodage du fichier est bien UTF-8. Cet encodage est aujourd'hui la norme et constitue l'encodage par défaut sur la plupart des systèmes Linux/Unix. Néanmois, je ne sais pas si c'est la même

```
% Première ligne du document

\mainlanguage[fr]    % Langue français

\setuppapersize[S5]  % Format du papier

\setupbodyfont       % Police = Latin Modern, 12 points
  [modern,18pt]

\setuphead           % Format des titres de chapitre
  [chapter]
  [style=\bfc]

\starttext           % Début du contenu du document

\startchapter
  [title=La maison sur le port]

Il y avait des    chansons
Les hommes        venaient y boire et rêver
Dans la maison    sur le port
Où les filles     riaient fort
Où le vin faisait chanter chanter chanter

Les pêcheurs      vous le diront
Ils y venaient    sans façon
Avant de partir   retirer leurs filets
Ils venaient      se réchauffer près de nous
Dans la maison    sur le port

\stopchapter

\stoptext            % Fin du document
```

chose sous Mac OS ou Windows et il est encore tout à fait possible que l'encodage ANSI soit utilisé. En tout cas, si nous ne sommes pas sûrs, depuis l'éditeur de texte lui-même, nous pouvons voir avec quel encodage le fichier sera enregistré et, si nécessaire, le modifier. La manière de procéder dépend, bien entendu, de l'éditeur avec lequel nous travaillons. Dans GNU Emacs, par exemple, en appuyant simultanément sur les touches CTRL-X puis Return suivi de « f », dans la dernière ligne de la fenêtre (que GNU Emacs appelle mini-buffer) un message apparaîtra nous demandant un nouvel encodage et nous informant de l'encodage actuel. Dans les autres éditeurs, nous pouvons généralement accéder à l'encodage dans le menu « Enregistrer sous ».

Après avoir vérifié que l'encodage est correct et enregistré le fichier sur le disque, nous fermerons l'éditeur pour nous concentrer sur l'analyse de ce que nous avons écrit.

## Regardons le contenu de notre premier fichier source pour ConTEXt

La première ligne commence par le caractère « % ». C'est un caractère réservé qui indique à ConTEXt de ne pas traiter le texte qui le suit et ce jusqu'à la fin de la ligne sur laquelle il se trouve. Cette fonctionnalité est utilisée pour écrire des commentaires dans le fichier source que seul l'auteur pourra lire, car ils ne seront pas incorporés au document final. Dans cet exemple, je l'ai par exemple utilisé pour attirer l'attention sur certaines lignes, en expliquant ce qu'elles font.

Les lignes suivantes commencent par le caractère « \ » qui est un autre des caractères réservés de ConTEXt et indique que ce qui suit est le nom d'une commande. L'exemple comprend plusieurs commandes couramment utilisée dans un fichier source ConTEXt : la langue dans laquelle le document est écrit, le format du papier, la police à utiliser dans le document et la mise en forme appliquée aux titres de chapitres. Plus tard, dans d'autres chapitres, nous détailleront ces commandes, pour le moment je veux juste que le lecteur voit à quoi elles ressemblent : elles commencent toujours par le caractère « \ », suivi du nom de la commande, et ensuite, entre parenthèses ou accolades, selon le cas, les données dont la commande a besoin pour produire ses effets. Entre le nom de la commande et les crochets ou accolades qui l'accompagnent, il peut y avoir des espaces vides ou des sauts de ligne. C'est à l'auteur de choisir la façon dont le code source est le plus clair et lisible.

Sur la 9ème ligne de notre exemple (je ne compte que les lignes qui ont du texte) se trouve la commande importante `\starttext` : elle indique à ConTEXt que le contenu du document commence à partir de cet endroit; et à la dernière ligne de notre exemple, nous voyons la commande `\stoptext` qui indique la fin du contenu. Tout ce qui suit cette dernière commande ne sera pas traité. Ce sont deux commandes très importantes sur lesquelles je reviendrai très bientôt. Entre les deux se trouve donc le contenu à proprement parler de notre document qui, dans notre exemple, consiste en la première strophe de la chanson "« La maison sur le port », dont les paroles sont de Amalia Rodrigues, et qui a été reprise notamment par Sanseverino. Je l'ai écrit en prose afin de mieux observer le formatage des paragraphes effectué par ConTEXt.

## Traitement du document source

Pour l'étape suivante, après s'être assuré que ConTEXt a été correctement installé dans notre système, nous devons ouvrir un terminal dans le répertoire où se trouve notre fichier « la-maison-sur-le-port.tex ».

> De nombreux éditeurs de texte vous permettent de compiler le document sur lequel vous travaillez sans ouvrir un terminal. Cependant, la procédure *canonique* pour traiter un document avec ConTEXt implique de le faire à partir d'un terminal, en exécutant directement le programme. Je vais procéder de cette manière (ou supposer qu'il en est ainsi) tout au long de ce document pour plusieurs raisons ; la première est que je n'ai aucun moyen de savoir avec quel éditeur chaque lecteur travaille. Mais le plus important est que, depuis le terminal, nous aurons accès à la *sortie* de « `context` », c'est à dire les messages émis par le programme.

Si la distribution ConTEXt que nous avons installée est « ConTEXt Standalone », nous devons tout d'abord exécuter le *script* qui indique au terminal les chemins et l'emplacement des fichiers dont ConTEXt a besoin pour fonctionner. Sur les systèmes Linux/Unix, cela se fait en tapant la commande suivante :

```
1 $ source ~/context/tex/setuptex
```


```
$ source ~/context/tex/setuptex
```


en supposant que nous avons installé ConTEXt dans un répertoire appelé « `context` ».

> En ce qui concerne l'exécution du *script* qui vient d'être mentionné, voir ce qui est dit dans l'annexe ?? concernant l'installation de « ConTEXt Standalone ».

Une fois que les variables nécessaires à l'exécution de « `context` » ont été chargées en mémoire, nous pouvons l'exécuter. Cela se fait en tapant dans le terminal :

```
1 $ context la-maison-sur-le-port
```

```
$ context la-maison-sur-le-port
```

Notez que bien que le fichier source s'appelle « la-maison-sur-le-port.tex » dans l'appel à « `context` » nous avons omis l'extension du fichier. Si nous avions appelé au fichier source, par exemple, « `la-maison-sur-le-port.mkiv` » (ce que je fais habituellement pour savoir que ce fichier est écrit pour Mark IV), il aurait fallu indiquer expressément l'extension du fichier à compiler en tapant « `context la-maison-sur-le-port.mkiv` ».

Après avoir exécuté « `context` » dans le terminal, plusieurs dizaines de lignes s'affichent à l'écran, informant de ce que fait ConTEXt. Les informations s'affichent à une vitesse impossible à suivre par un être humain, mais ne vous inquiétez pas, car en plus de l'écran, ces informations sont également stockées dans un fichier auxiliaire, avec l'extension « `.log` » qui est généré avec la compilation et que nous pourrons consulter tranquililement plus tard si nécessaire.

Après quelques secondes, si nous avons bien écrit le texte de notre fichier source, sans faire d'erreur grave, l'émission de messages vers le terminal se terminera. Le dernier des messages nous informera du temps nécessaire à la compilation. La première fois qu'un document est compilé, cela prend toujours un peu plus de temps, car ConTEXt doit construire à partir de zéro certains fichiers contenant les informations de notre document. Par la suite ils seront juste réutilisés et complétés pour les compilations suivantes qui iront plus vite. Le message du temps passé indique que la compilation est terminée. Si tout s'est bien passé, trois fichiers supplémentaires apparaîtront dans le répertoire où nous avons exécuté « `context` » :

- la-maison-sur-le-port.pdf
- la-maison-sur-le-port.log
- la-maison-sur-le-port.tuc

Le premier est le résultat de notre traitement, c'est-à-dire : le fichier PDF déjà formaté. Le deuxième est le fichier dans lequel sont stockées toutes les informations qui ont été affichées à l'écran pendant la compilation ; le troisième est un fichier auxiliaire que ConTEXt génère pendant la compilation et qui est utilisé pour construire les index et les

références croisées. Pour le moment, si tout a fonctionné comme prévu, nous pouvons supprimer les deux fichiers (« `la-maison-sur-le-port.log` » et « `la-maison-sur-le-port.tuc` »). S'il y a eu un problème, les informations contenues dans ces fichiers peuvent nous aider à localiser la source du problème et à déterminer comment le résoudre.

Si nous n'avons pas obtenu ces résultats, c'est probablement dû à un ou plusieurs de points qui suivent :

- soit nous n'avons pas installé correctement notre distribution ConTEXt, auquel cas en tapant la commande « `context` » dans le terminal, un message « commande inconnue » sera apparu.
- soit notre fichier n'a pas été encodé en UTF-8 et cela a généré une erreur de compilation.
- soit peut-être que la version de ConTEXt installée sur notre système est Mark II. Dans cette version, vous ne pouvez pas utiliser l'encodage UTF-8 sans l'indiquer explicitement dans le fichier source lui-même. Nous pourrions corriger le fichier source pour qu'il compile bien, mais, puisque cette introduction se réfère à Mark IV, cela n'a pas de sens de continuer à travailler avec Mark II : la meilleure chose à faire est d'installer « ConTEXt Standalone ».
- Soit nous avons fait une erreur en écrivant dans le fichier source le nom de certaines commandes ou leurs données associées.

  Si après l'exécution de « `context` », le terminal a commencé à émettre des messages, mais s'est ensuite arrêté sans que le *prompt* ne réapparaisse, avant de continuer, il faut appuyer sur CTRL-X pour interrompre l'exécution de ConTEXt qui a été interrompue par l'erreur.

En cas de problème, nous devrons donc vérifier chacune de ces possibilités, et les corriger, jusqu'à ce que la compilation se déroule correctement.

La figure 2.1 montre le contenu de « la-maison-sur-le-port.pdf ». Nous pouvons voir que ConTEXt a numéroté la page, numéroté le chapitre et écrit le texte dans la police indiquée. Il a également réparti le mot « venaient » entre la sixième et la septième ligne, ainsi que le mot « maison » la septième et la guitième ligne. ConTEXt, par défaut, active la césure (division syllabique) des mots afin de répartir les blancs (les espaces

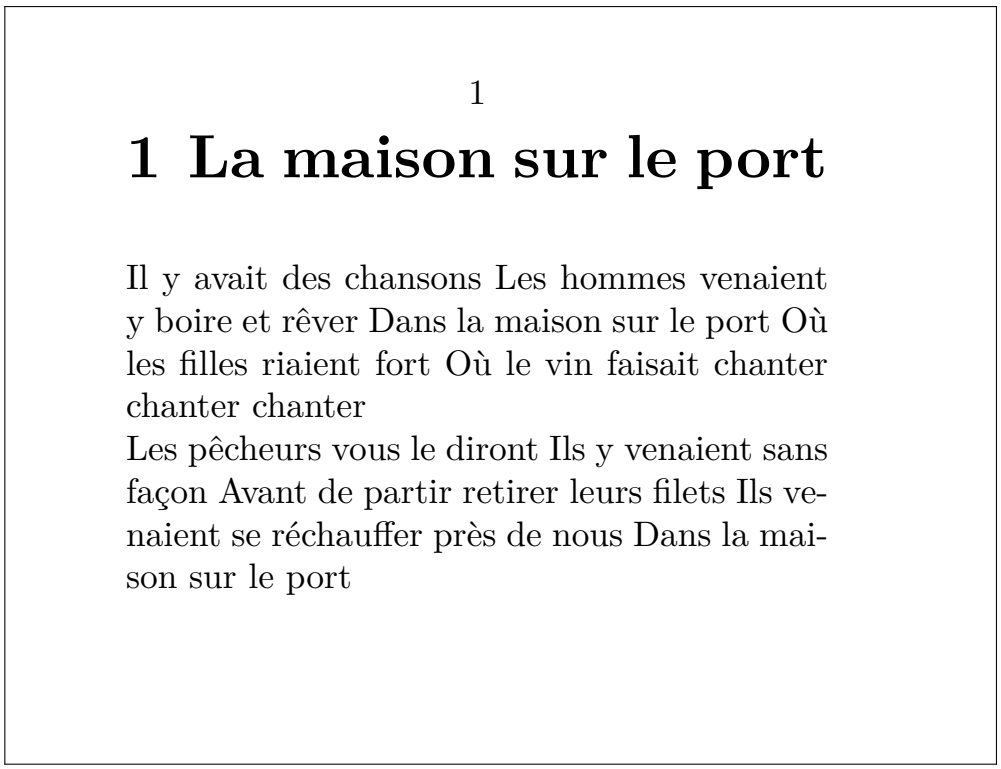


**Figure 2.1** La maison sur le port

vides entre les mots) de façon la plus homogène possible. C'est pourquoi il est si important d'informer ConTEXt de la langue du document, car les modèles de césure varient selon la langue. Dans notre exemple, c'est l'objectif de la première commande du fichier source (`\mainlanguage[fr]`).

En bref : ConTEXt a transformé le fichier source et a généré un fichier dans lequel nous avons un document formaté selon les instructions qui étaient incluses dans le fichier source. Les commentaires en ont disparu et, en ce qui concerne les commandes, ce que nous avons maintenant n'est pas leur nom, mais le résultat de leur application par ConTEXt.

## 2.3 La structure de notre fichier d'exemple

Dans un projet aussi simple que notre exemple, développé dans un seul fichier source, la structure de celui-ci est très simple et est marquée par les commandes `\starttext` ... `\stoptext`. Tout ce qui se trouve entre

la première ligne du fichier et la commande `\starttext` constitue le *préambule*. Le contenu du document lui-même est inséré entre les commandes `\starttext` et `\stoptext`. Dans notre exemple, le préambule comprend quatre commandes de configuration globale : une pour indiquer la langue de notre document (`\mainlanguage`), une autre pour indiquer la taille des pages (`\setuppapersize`) qui dans notre cas est « S5 », représentant les proportions d'un écran d'ordinateur, une troisième commande (`\setupbodyfont`) qui nous permet d'indiquer la police de caractère et sa taille, et une quatrième (`\setuphead`) qui nous permet de configurer l'apparence des titres des chapitres.

Le corps du document est encadré par les commandes `\starttext` et `\stoptext`. Ces commandes indiquent, respectivement, le point de départ et le point final du texte à traiter : entre elles, nous devons inclure tout le texte que nous voulons que ConTEXt traite, ainsi que les commandes qui ne doivent pas affecter le document entier mais seulement des fragments de celui-ci. Pour le moment, nous devons supposer que les commandes `\starttext` et `\stoptext` sont obligatoires dans tout document ConTEXt, bien que plus tard, en parlant des projets multifichiers (section ??) nous verrons qu'il y a quelques exceptions.

## 2.4 Quelques détails supplémentaires sur la façon d'exécuter « `context` »

La commande « `context` » avec laquelle nous avons procédé au traitement de notre premier fichier source est, en fait, un *script* Lua, c'est-à-dire : un petit programme Lua qui, après avoir effectué quelques vérifications et opérations, appelle LuaTEXpour traiter le fichier source.

Nous pouvons appeler « `context` » avec plusieurs options. Les options sont saisies immédiatement après le nom de la commande, précédées de deux traits d'union. Si nous voulons saisir plus d'une option, nous les séparons par un espace blanc. L'option « `help` » nous donne une liste de toutes les options, avec une brève explication de chacune d'elles :

```
1 $ context --help
```

```
$ context --help
```

Parmi les options les plus intéressantes, citons les suivantes :

**interface** Comme je l'ai dit dans le chapitre d'introduction, l'interface de ConTEXt est traduite en plusieurs langues. Par défaut, c'est l'interface anglaise qui est utilisée, mais cette option nous permet de lui demander d'utiliser la version néerlandaise (nl), française (fr), italienne (it), allemande (de) ou roumaine (ro).

**purge, purgeall** Supprime les fichiers auxiliaires générés pendant le traitement.

**result=Name** indique le nom que doit porter le fichier PDF résultant. Par défaut, ce sera le même que le fichier source à traiter, avec l'extension « `.pdf` ».

**usemodule=list** Charge les modules qui sont indiqués avant d'exécuter ConTEXt (un module est une extension de ConTEXt, qui ne fait pas partie de son noyau, et qui lui fournit une utilité supplémentaire).

**useenvironment=list** Charge les fichiers d'environnement qui sont spécifiés avant de lancer ConTEXt (un fichier d'environnement est un fichier contenant des instructions de configuration).

**version** indique la version de ConTEXt.

**help** affiche des informations d'aide sur les options du programme.

**noconsole** Supprime l'envoi de messages à l'écran pendant la compilation. Toutefois, ces messages seront toujours enregistrés dans le fichier « `.log` ».

**nonstopmode** Exécute la compilation sans s'arrêter sur les erreurs. Cela ne signifie pas que l'erreur ne se produira pas, mais que lorsque ConTEXt rencontre une erreur, même si elle est récupérable, il continuera la compilation jusqu'à ce qu'elle se termine ou jusqu'à ce qu'il rencontre une erreur irrécupérable.

`batchmode` Il s'agit d'une combinaison des deux options précédentes. Il fonctionne sans interruption et omet les messages à l'écran.

Pour les premières utilisation et pour l'apprentissage de ConTEXt, je ne pense pas que ce soit une bonne idée d'utiliser les trois dernières options, car lorsqu'une erreur se produit, nous n'aurons aucune idée de l'endroit où elle se trouve ou de ce qui l'a produite. Et, croyez-moi chers lecteurs, tôt ou tard, une erreur de compilation se produira.

## 2.5 Traitement des erreurs

En travaillant avec ConTEXt, il est inévitable que, tôt ou tard, des erreurs se produisent dans la compilation. En gros, nous pouvons regrouper les erreurs dans l'une des quatre catégories suivantes :

1. **Erreurs de frappe**. Elles se produisent lorsque nous orthographions mal le nom d'une commande. Dans ce cas, nous envoyons au compilateur une commande qu'il ne comprend pas. Par exemple, lorsque, au lieu d'écrire la commande `\TeX`, nous écrivons `\Tex` avec le « X » final en minuscule, puisque ConTEXt fait la différence entre les majuscules et les minuscules et considère donc que « TeX » et « Tex » sont des mots différents ; ou si les options utilisée pour une commande, au lieu de les mettre entre crochets, sont mises entre accolades, ou si nous essayons d'utiliser un des caractères réservés comme s'il s'agissait d'un caractère normal, etc.

2. **Erreurs par omission**. Dans ConTEXt il y a des instructions qui démarrent une tâche, dont il faut indiquer explicitement quand la fermer ; comme le caractère réservé « $ » qui active le mode mathématique, qui est maintenu jusqu'à ce qu'on le désactive, et si on oublie de le désactiver, une erreur sera générée dès qu'on trouvera un texte ou une instruction qui n'a pas de sens dans le mode mathématique. Il en va de même si nous commençons un bloc de texte au moyen du caractère réservé « { » ou d'une commande `\startUnTruc` et que, par la suite, la fermeture explicite n'est pas trouvée (« } » ou `\stopUnTruc`).

3. **Erreurs de conception**. J'appelle ainsi les erreurs qui se produisent lorsque vous appelez une commande qui nécessite certains arguments, sans les fournir, ou lorsque la syntaxe d'appel de la commande n'est pas correcte.

4. **Erreurs situationnelles**. Certaines commandes sont destinées à ne fonctionner que dans certains contextes ou environnements, et sont donc inconnues en dehors de ceux-ci. Cela se produit, en particulier, avec le mode mathématique : certaines commandes ConTEXt ne fonctionnent que lors de l'écriture de formules mathématiques et si elles sont appelées dans d'autres contextes, elles génèrent une erreur.

Que faire lorsque « `context` » nous avertit, pendant la compilation, qu'une erreur s'est produite ? La première chose est, évidemment, d'identifier quelle est l'erreur. Pour ce faire, nous devrons parfois analyser le fichier « `.log` » généré pendant la compilation ; mais encore plus souvent il suffira de remonter dans les messages produits par « `context` » dans le terminal où il est exécuté.

Par exemple, si dans notre fichier de test, « `la-maison-sur-le-port.tex` », par erreur, au lieu de `\starttext` nous avions écrit `\startext` (avec un seul « t »), ce qui, par ailleurs, est une erreur très courante, lors de l'exécution de « `context la-maison-sur-le-port` », lorsque la compilation était arrêtée, dans l'écran du terminal nous pouvions voir l'information montrée dans la figure 2.2.

Nous pouvons y voir les lignes de notre fichier source numérotées, et à l'une d'entre elles, dans notre cas la ligne 14, entre le numéro et le texte de la ligne le compilateur a ajouté « >> » pour indiquer que c'est dans cette ligne qu'il a trouvé l'erreur. Le numéro de la ligne est également indiqué plus haut, avnat l'affichage des lignes, dans une ligne commençant par « `tex error` ». Le fichier « `la-maison-sur-le-port.log` » nous donnera plus d'indices. Dans notre exemple, il ne s'agit pas d'un très gros fichier, car la source que nous compilions est très petite ; dans d'autres cas, il peut contenir une quantité écrasante d'informations. Mais nous devons nous y plonger. Si nous ouvrons « `la-maison-sur-le-port.log` » avec un éditeur de texte, nous verrons que ce fichier enregistre tout ce que fait ConTEXt. Nous devons y chercher une ligne

```
tex error       > tex error on line 14 in file la-maison-sur-le-port_bug.tex:
! Undefined control sequence

l.14 \startext
                      % Début du contenu du document

 4
 5     \setuppapersize[S5]  % Format du papier
 6
 7     \setupbodyfont       % Police = Latin Modern, 12 points
 8       [modern,18pt]
 9
10     \setuphead           % Format des titres de chapitre
11       [chapter]
12       [style=\bfc]
13
14 >>  \startext           % Début du contenu du document
15
16     \startchapter
17       [title=La maison sur le port]
18
19     Il y avait des    chansons
20     Les hommes        venaient y boire et rêver
21     Dans la maison    sur le port
22     Où les filles     riaient fort
23     Où le vin faisait chanter chanter chanter
24

mtx-context     | fatal error: return code: 256
```

**Figure 2.2** Sortie d'affichage en cas d'erreur de compilation

qui commence par un avertissement d'erreur « `tex error` », pour cela nous pouvons utiliser la fonction de recherche de texte de l'éditeur. Nous trouveerons les lignes d'erreur suivantes :

```
tex error       > tex error on line 14 in file la-maison-sur-le-port_bug.tex:
! Undefined control sequence

l.14 \startext
                      % Début du contenu du document
```

> **Note :** La première ligne informant de l'erreur, dans le fichier « `la-maison-sur-le-port.log` » est très longue. Pour que cela soit présentable ici, en tenant compte de la largeur de la page, j'ai supprimé une partie du chemin indiquant l'emplacement du fichier.

Si nous prêtons attention aux trois lignes du message d'erreur, nous voyons que la première nous indique à quel numéro de ligne l'erreur s'est produite (ligne 14) et de quel type d'erreur il s'agit : « Undefined

```
tex error       > tex error on line 14 in file la-maison-sur-le-port_bug.tex:
! Undefined control sequence

l.14 \startext
                      % Début du contenu du document

 4
 5     \setuppapersize[S5]  % Format du papier
 6
 7     \setupbodyfont       % Police = Latin Modern, 12 points
 8       [modern,18pt]
 9
10     \setuphead           % Format des titres de chapitre
11       [chapter]
12       [style=\bfc]
13
14 >>  \startext           % Début du contenu du document
15
16     \startchapter
17       [title=La maison sur le port]
18
19     Il y avait des    chansons
20     Les hommes        venaient y boire et rêver
21     Dans la maison    sur le port
22     Où les filles     riaient fort
23     Où le vin faisait chanter chanter chanter
24

mtx-context     | fatal error: return code: 256
```

control sequence », ou, ce qui revient au même : « Unknown control sequence », c'est-à-dire une commande inconnue. Les deux lignes suivantes du fichier journal nous montrent la ligne 14, qui commence à l'endroit où l'erreur s'est produite. Donc il n'y a pas de doute, l'erreur est dans `\startext`. Nous le lirons attentivement et, avec de l'attention et de l'expérience, nous nous rendrons compte que nous avons écrit « startext » et non « starttext » (avec un double « t »).

Pensez que les ordinateurs sont très bons et très rapides pour exécuter des instructions, mais très maladroits pour lire nos pensées, et que le mot « startext » n'est pas le même que « starttext ». Dans le second cas, le programme sait comment l'exécuter ; dans le premier cas, il ne sait pas quoi faire.

D'autres fois, la localisation de l'erreur ne sera pas aussi facile. En particulier lorsque l'erreur est qu'une tâche a été lancée et que sa fin n'a pas été expressément spécifiée. Parfois, au lieu de chercher l'expression « tex error » dans le fichier « `.log` », vous devez chercher un astérisque.

Ce caractère au début d'une ligne du fichier journal représente, non pas une erreur fatale, mais un avertissement. Et les avertissements peuvent être utiles pour localiser l'erreur.

Et si les informations du fichier « `.log` » ne sont pas suffisantes, il faudra aller, petit à petit, localiser l'endroit de l'erreur. Une bonne stratégie pour cela consiste à changer l'emplacement de la commande `\stoptext` dans le fichier source. Rappelez-vous que ConTEXt arrête de traiter le texte dès qu'il trouve cette commande. Par conséquent, si, dans mon fichier source, j'écris, plus ou moins à la hauteur du milieu, un `\stoptext` et que je compile, seule la première moitié sera traitée ; si l'erreur se répète, je saurai qu'elle se trouve dans la première moitié du fichier source, si elle ne se répète pas, cela signifie que l'erreur se trouve dans la deuxième moitié... et ainsi, petit à petit, en changeant l'emplacement de la commande `\stoptext`, nous pouvons localiser l'emplacement de l'erreur.

Une autre astuce consiste à mettre en commentaires le paquets de lignes douteuses avec le caractère « % » (certain éditeur de texte propose une fonction pour commenter et décommenter tout un paquet de ligne automatiquement).

Une fois que nous l'avons localisée, nous pouvons essayer de la comprendre et de la corriger ou, si nous ne pouvons pas comprendre pourquoi l'erreur se produit, au moins, ayant localisé le point où elle se trouve, nous pouvons essayer d'écrire les choses d'une manière différente pour éviter que l'erreur se reproduise.i

Ce dernier point, bien sûr, uniquement si nous sommes les auteurs ; si nous nous limitons à composer le texte de quelqu'un d'autre, nous ne pourrons pas le modifier et nous devrons continuer à enquêter jusqu'à ce que nous découvrions les raisons de l'erreur et sa possible solution.

Dans la pratique, lorsqu'on crée un document relativement volumineux avec ConTEXt, ce que l'on fait habituellement, c'est de le compiler de temps en temps, au fur et à mesure de la rédaction du document, de sorte que si une erreur se produit, nous savons plus ou moins clairement quelle est la partie du document qui vient d'être introduite depuis la précédente compilation et qui engendre la nouvelle erreur.

# Chapitre 3

# Les commandes et autres concepts fondamentaux de ConTEXt



Nous avons déjà vu que dans le fichier source, avec le contenu réel de notre futur document formaté, nous insérons les instructions nécessaires pour expliquer à ConTEXt comment nous voulons que notre contenu soit mis en forme. Nous pouvons appeler ces instructions « commandes », « macros » ou « séquences de contrôle ».

> Du point de vue du fonctionnement interne de ConTEXt (en fait, du fonctionnement de TEX), il y a une différence entre les *primitives* et les *macros*. Une primitive est une instruction simple qui ne peut pas être décomposée en d'autres instructions plus simples. Une macro est une instruction qui peut être décomposée en d'autres instructions plus simples qui, à leur tour, peuvent peut-être aussi être décomposées en d'autres encore, et ainsi de

suite. La plupart des instructions de ConTEXtsont, en fait, des macros. Du point de vue du programmeur, la différence entre les macros et les primitives est importante. Mais du point de vue de l'utilisateur, la question n'est pas si importante : dans les deux cas, nous avons des instructions qui sont exécutées sans que nous ayons besoin de nous préoccuper de leur fonctionnement à un niveau inférieur. Par conséquent, la documentation ConTEXt parle généralement d'une *commande* lorsqu'elle adopte le point de vue de l'utilisateur, et d'une *macro* lorsqu'elle adopte le point de vue du programmeur. Puisque nous ne prenons que la perspective de l'utilisateur dans cette introduction, j'utiliserai l'un ou l'autre terme, les considérant comme synonymes.

Les *commandes* sont des ordres donnés au programme ConTEXt pour qu'il fasse quelque chose. Nous *contrôlons* les performances du programme par leur intermédiaire. Ainsi KNUTH, le père de TEX, utilise le terme de *séquences de contrôle* pour se référer à la fois aux primitives et aux macros, et je pense que c'est le terme le plus précis de tous. Je l'utiliserai lorsque je penserai qu'il est important de distinguer entre *symboles de contrôle* et *mots de contrôle*.

Les instructions de ConTEXt sont essentiellement de deux sortes : les caractères réservés, et les commandes proprement dites.

## 3.1 Les caractères réservés de ConTEXt

Lorsque ConTEXt lit le fichier source composé uniquement de caractères de texte, puisqu'il s'agit d'un fichier texte, il doit d'une manière ou d'une autre distinguer ce qui est le contenu textuel à mettre en forme, et les instructions qu'il doit exécuter. Les caractères réservés de ConTEXt sont ce qui lui permet de faire cette distinction. En principe, ConTEXt suppose que chaque caractère du fichier source est un texte à traiter, sauf s'il s'agit de l'un des 11 caractères réservés qui doivent être traités comme une *instruction*.

Seulement 11 instructions ? Non. Il n'y a que 11 caractères réservés, mais l'un d'entre eux, le caractère de « \ », a pour fonction de convertir le ou les caractères qui le suivent immédiatement en instruction, rendant ainsi le nombre potentiel de commandes illimité. ConTEXt a environ 3000 commandes (en additionnant les commandes exclusives à Mark II, Mark IV et celles communes aux deux versions).

Les caractères réservés sont les suivants :

\ % { } # ~ | $ _ ^ &

ConTEXt les interprète de la façon suivante :

\ Ce caractère est le plus important pour nous : il indique que ce qui vient immédiatement après ne doit pas être interprété comme du texte mais comme une instruction. Il est appelé « Caractère d'échappement » ou « Séquence d'échappement » (même s'il n'a rien à voir avec la touche « Esc » que l'on trouve sur la plupart des claviers).[1]

% Indique à ConTEXt que ce qui suit jusqu'à la fin de la ligne est un commentaire qui ne doit pas être traité ou inclus dans le fichier formaté final. L'introduction de commentaires dans le fichier source est extrêmement utile. Cela permet par exemple d'expliquer pourquoi quelque chose a été fait d'une certaine manière, comment tel ou tel effet graphique a été obtenu, garder un rapper d'une idée à compléter ou à réviser, d'une illustration à construire.

Il peut également être utilisé pour aider à localiser une erreur dans le fichier source, puisqu'en commentant une ligne, nous l'excluons de la compilation, et pouvons voir si elle est à l'origine de l'erreur de compilation. Le commentaire peut aussi être utilisé pour stocker deux versions différentes d'une même macro, et ainsi obtenir des résultats différents après la compilation ; ou pour empêcher la compilation d'un extrait dont nous ne sommes pas sûrs, mais sans le supprimer du fichier source au cas où nous voudrions y revenir plus tard ; ou pour partager des commentaires lros de l'édition en mode collaboratif d'un document... etc.

Avec la possibilité que notre fichier source contienne du texte que personne d'autre que nous ne puisse voir, nos utilisations de ce caractère ne sont limitées que par notre propre imagination. J'avoue que c'est l'un des utilitaires qui me manque le plus

[1] Dans la terminologie informatique, la touche qui affecte l'interprétation du caractère suivant est appelée le « caractère d'échappement ». En revanche, la touche *escape key* des claviers est appelée ainsi car elle génère le caractère 27 en code ASCII, qui est utilisé comme caractère d'échappement dans cet encodage. Aujourd'hui, l'utilisation de la touche Echap est davantage associée à l'idée d'annuler une action en cours.

lorsque le seul remède pour écrire un texte est un logiciel de traitement de texte.

`{` Ce caractère ouvre un groupe. Les groupes sont des blocs de texte auxquels on souhaite appliquer certaines effet ou affecter certaines caractéristiques. Nous en parlerons dans la section ??.

`}` Ce caractère cloture un groupe préalablement ouvert avec « `{` ».

`#` Ce caractère est utilisé pour définir les macros. Il fait référence aux arguments de la macro. Voir section 3.7.1 dans ce chapitre.

`~` Introduit un espace blanc insécable dans le document pour éviter un saut de ligne entre les mots qu'il sépare, ce qui signifie que deux mots séparés par le caractère `~` resteront toujours sur la même ligne. Nous parlerons de cette instruction et de l'endroit où elle doit être utilisée dans section ??.

`|` Ce caractère est utilisé pour indiquer que deux mots joints par un élément de séparation constituent un mot composé qui peut être divisé par syllabes en la première composante, mais pas en la seconde. Voir section ??.

`$` Ce caractère est un *interrupteur* pour le mode mathématique. Il active ce mode s'il n'était pas activé, ou le désactive s'il l'était. En mode mathématique, ConTEXt applique des polices et des règles différentes des polices normales, afin d'optimiser l'écriture des formules mathématiques. Bien que l'écriture des mathématiques soit une utilisation très importante de ConTEXt je ne la développerai pas dans cette introduction. Étant un homme de lettres, je ne me sens pas à la hauteur !

`_` Ce caractère est utilisé en mode mathématique pour indiquer que ce qui suit doit être mis en indice. Ainsi, par exemple, pour obtenir $x_1$, il faut écrire `$x_1$`.

`^` Ce caractère est utilisé en mode mathématique pour indiquer que ce qui suit doit être mis en exposant. Ainsi, par exemple, pour obtenir $(x+i)^{n^3}$, il faut écrire `$(x+i)^{n^3}$`.

`&` La documentation de ConTEXt indique qu'il s'agit d'un caractère réservé, mais ne précise pas pourquoi. Ce caractère semble avoir essentiellement deux usages : il est utilisé pour aligner certains éléments verticalements dans les tableaux de base et, dans un contexte mathématique, dans les écritures matricielles . Comme je suis un littéraire, je ne me sens pas capable de faire des tests supplémentaires pour voir à quoi sert précisément ce caractère réservé.

Concernant le choix des caractères réservés, il doit s'agir de caractères disponibles sur la plupart des claviers mais qui ne sont habituellement peu ou pas utilisés dans les écritures. Cependant, bien que peu courants, il est toujours possible que certains d'entre eux apparaissent dans nos documents, comme par exemple lorsque nous voulons écrire que quelque chose coûte 100 dollars ($100), ou qu'en Espagne, le pourcentage de conducteurs de plus de 65 ans était de 16% en 2018. Dans ces cas, nous ne devons pas écrire le caractère réservé directement, mais utiliser une *commande* qui produira le caractère réservé correctement dans le document final. La commande pour chacun des caractères réservés se trouve dans table 3.1.

| Caractère réservé | Commande qui le génère |
|---|---|
| \ | \backslash |
| % | \% |
| { | \{ |
| } | \} |
| # | \# |
| ~ | \lettertilde |
| \| | \| |
| $ | \$ |
| _ | _ |
| ^ | \letterhat |
| & | \& |

**Tableau 3.1** Ecriture des caractères réservés

Une autre façon d'obtenir les caractères réservés est d'utiliser la commande `\type`. Cette commande envoie ce qu'elle prend comme argument au document final sans le traiter d'aucune manière, et donc sans

l'interpréter. Dans le document final, le texte reçu de `\type` sera affiché dans la police monospace typique des terminaux informatiques et des machines à écrire.

> Normalement, nous devrions placer le texte que `\type` doit afficher entre accolades. Cependant, lorsque ce texte comprend lui-même des crochets ouvrants ou fermants, nous pouvons, à la place, enfermer le texte entre deux caractères égaux qui ne font pas partie du texte qui constitue l'argument de `\type`. Par exemple : `\type*…*`, ou `\type+…+`.

Si, par erreur, nous utilisons directement un des caractères réservés autrement que pour l'usage auquel il est destiné, parce que nous avons justement oublié qu'il s'agissait d'un caractère réservé ne pouvant être utilisé comme un caractère normal, trois choses peuvent se produire :

1. Le plus souvent, une erreur est générée lors de la compilation.

2. Nous obtenons un résultat inattendu. Cela se produit surtout avec « ~ » et « % » ; dans le premier cas, au lieu du « ~ » attendu dans le document final, un espace blanc sera inséré ; et dans le second cas, tout ce qui se trouve après « % » sur la même ligne ne sera pas pris en compte par ConTEXt qui le considèrera comme commentaire. L'utilisation incorrecte de la « \ » peut également produire un résultat inattendu si elle ou les caractères qui la suivent immédiatement constituent une commande connue de ConTEXtCependant, le plus souvent, lorsque nous utilisons incorrectement la « \ », nous obtenons une erreur de compilation.

3. Aucun problème ne se produit : Cela se produit avec trois des caractères réservés utilisés principalement en mathématiques (`_ ^ &`) : s'ils sont utilisés en dehors de cet environnement, ils sont traités comme des caractères normaux.

> Le point 3 est ma conclusion. La vérité est que je n'ai trouvé aucun endroit dans la documentation de ConTEXt qui nous indique où ces caractères réservés peuvent être utilisés directement ; dans mes tests, cependant, je n'ai vu aucune erreur lorsque cela est fait ; contrairement, par exemple, à LATEX.

## 3.2 Les commandes à proprement parler

Les commandes proprement dites commencent donc toujours par le caractère « \ ». En fonction de ce qui suit immédiatement ce caractère d'échappement, une distinction est faite entre :

a. **Symboles de contrôle.** Un symbole de contrôle commence par la séquence d'échappement (« \ ») et consiste exclusivement en un caractère autre qu'une lettre, comme par exemple « `\,` », « `\1` », « `\'` » ou « `\%` ». Tout caractère ou symbole qui n'est pas une lettre au sens strict du terme peut être un symbole de contrôle, y compris les chiffres, les signes de ponctuation, les symboles et même un espace vide. Dans ce document, pour représenter un espace vide (espace blanc) lorsque sa présence doit être soulignée, le symbole que j'utilise est ␣. En fait, « `\␣` » (une barre oblique inversée suivie d'un espace blanc) est un symbole de contrôle couramment utilisé, comme nous pourrons bientôt le constater.

   > Un espace vide ou blanc est un caractère « invisible », ce qui pose un problème dans un document comme celui-ci, où il faut parfois préciser clairement ce qui doit être écrit dans un fichier source. Knuth était déjà conscient de ce problème et, dans son « The TEXBook », il a pris l'habitude de représenter les espaces vides importants par le symbole « ␣ ». Ainsi, par exemple, si nous voulions montrer que deux mots du fichier source doivent être séparés par deux espaces vides, nous écririons « word1␣␣word2 ».

b. **Mots de contrôle.** Si le caractère qui suit immédiatement la barre oblique inversée est une lettre à proprement parler, la commande sera un *Mot de contrôle*. Ce groupe de commandes est largement majoritaire. Il a une caractéristique très important : le nom de la commande ne peut être composé que de lettres ; les chiffres, les signes de ponctuation ou tout autre type de symbole ne sont pas autorisés. Seules les lettres minuscules ou majuscules sont autorisées. N'oubliez pas, par ailleurs, que ConTEXt fait une distinction entre les minuscules et les majuscules, ce qui signifie que les commandes `\mycommand` et `\MyCommand` sont différentes. Mais `\MaCommande1` et `\MaCommande2` seraient considérées comme identiques, puisque

n'étant pas des lettres, «1» et «2» ne font pas partie du nom des commandes.

Le manuel de référence de ConTEXt ne contient aucune règle sur les noms de commande, tout comme le reste des « manuels » inclus avec « ConTEXt Standalone ». Ce que j'ai dit dans le paragraphe précédent est ma conclusion basée sur ce qui se passe dans TEX (où, par ailleurs, des caractères comme les voyelles accentuées qui n'apparaissent pas dans l'alphabet anglais ne sont pas considérés comme des « lettres »).

Lorsque ConTEXt lit un fichier source et trouve le caractère d'échappement (« \ »), il sait qu'une commande va suivre. Il lit alors le premier caractère qui suit la séquence d'échappement. Si ce n'est pas une lettre, cela signifie que la commande est un symbole de contrôle et ne consiste qu'en ce premier symbole. Mais d'un autre côté, si le premier caractère après la séquence d'échappement est une lettre, alors ConTEXt continuera à lire chaque caractère jusqu'à ce qu'il trouve le premier caractère qui ne soit pas une lettre, et il saura alors que le nom de la commande est terminé. C'est pourquoi les noms de commande qui sont d␣es mots de contrôle ne peuvent pas contenir de caractères autres que des lettres.

Lorsque la « non-lettre » à la fin du nom de la commande est un espace vide, il est supposé que l'espace vide ne fait pas partie du texte à traiter, mais qu'il a été inséré exclusivement pour indiquer la fin du nom de la commande, donc ConTEXt se débarrasse de cet espace. Cela produit un effet qui surprend les débutants, car lorsque l'effet de la commande en question implique d'écrire quelque chose dans le document final, la sortie écrite de la commande est liée au mot suivant. Par exemple, les deux phrases suivantes dans le fichier source

```
Connaître \TeX aide à l'apprentissage de \ConTeXt.

Connaître \TeX, si non indispensable, aide à l'apprentissage de
\ConTeXt.

Connaître \TeX        aide à l'apprentissage de \ConTeXt.

Connaître \TeX{} aide à l'apprentissage de \ConTeXt.

Connaître \TeX\ aide à l'apprentissage de \ConTeXt.
```

[1] **Note:** par convention, pour illustrer quelque chose dans cette introduction, les exemple de code source utilise une police à espacement fixe. une coloration syntaxique cohérente de ConTEXt dans un cadre de fond gris. Le résultat de la compilation est présenté dans un cadre de fond de couleur jaune foncé.

produisent le résultat suivant [1]

Connaître TEXaide à l'apprentissage de ConTEXt.
Connaître TEX, si non indispensable, aide à l'apprentissage de ConTEXt.
Connaître TEXaide à l'apprentissage de ConTEXt.
Connaître TEX aide à l'apprentissage de ConTEXt.
Connaître TEX aide à l'apprentissage de ConTEXt.

```
Connaître \TeX aide à l'apprentissage de \ConTeXt.

Connaître \TeX, si non indispensable, aide à l'apprentissage de \ConTeXt.

Connaître \TeX        aide à l'apprentissage de \ConTeXt.

Connaître \TeX{} aide à l'apprentissage de \ConTeXt.

Connaître \TeX\ aide à l'apprentissage de \ConTeXt.
```

Connaître TEXaide à l'apprentissage de ConTEXt.
Connaître TEX, si non indispensable, aide à l'apprentissage de ConTEXt.
Connaître TEXaide à l'apprentissage de ConTEXt.
Connaître TEX aide à l'apprentissage de ConTEXt.
Connaître TEX aide à l'apprentissage de ConTEXt.

Notez comment, dans le premier cas, le mot « TEX » est relié au mot qui suit mais pas dans le second cas. Cela est dû au fait que, dans le premier cas du fichier source, la première « non-lettre » après le nom de la commande `\TeX` était un espace vide, supprimé parce que ConTEXt a supposé qu'il n'était là que pour indiquer la fin d'un nom de commande, alors que dans le second cas, il y avait une virgule, et comme ce n'est pas un espace vide, il n'a pas été supprimé. Le troisième exemple montre que l'ajout d'espaces blancs supplémentaires ne change rien, car une règle de ConTEXt (que nous verrons dans section ??) fait qu'un espace blanc « absorbe » tous les blancs et tabulations qui le suivent (1 espace ou 15, c'est pareil).

Par conséquent, lorsque nous rencontrons ce problème (qui heureusement n'arrive pas trop souvent), nous devons nous assurer que la première « non-lettre » après le nom de la commande n'est pas un espace blanc. Il existe deux candidats pour cela :

- Les caractères réservés « {} », utilisé à la quatrième ligne de l'exemple. Le caractère réservé « { », comme je l'ai dit, ouvre un groupe, et « } » ferme un groupe, donc la séquence « {} » introduit un groupe vide. Un groupe vide n'a aucun effet sur le document final, mais il aide ConTEXt à savoir que le nom de la commande qui le précède est terminé. On peut aussi créer un groupe autour de la commande en question, par exemple en écrivant « {\TeX} ». Dans les deux cas, le résultat sera que la première « non-lettre » après \TeX n'est pas un espace vide.

- Le symbole de contrôle « \␣ » (une barre oblique inverse suivie d'un espace vide, voir la note sur page 48) utilisé à la cinquième ligne de l'exemple. L'effet de ce symbole de contrôle est d'insérer un espace blanc dans le document final. Pour bien comprendre la logique de ConTEXt, il peut être utile de prendre le temps de voir ce qui se passe lorsque ConTEXt rencontre un mot de contrôle (par exemple \TeX) suivi d'un symbole de contrôle (par exemple « \␣ ») :

  – ConTEXt rencontre le caractère \ suivi d'un « T » et sachant que cela vient avant un mot de contrôle, il continue à lire les caractères jusqu'à ce qu'il arrive à une « non-lettre », ce qui se produit lorsqu'il arrive au caractère \ introduisant le prochain symbole de contrôle.
  – Une fois qu'il sait que le nom de la commande est \TeX, il exécute la commande et imprime TEX dans le document final. Il retourne ensuite à l'endroit où il a arrêté la lecture pour vérifier le caractère qui suit immédiatement la deuxième barre oblique inversée.
  – Il identifie qu'il s'agit d'un espace vide, c'est-à-dire d'une « non-lettre », ce qui signifie qu'il s'agit d'un symbole de contrôle, qu'il peut donc exécuter. Il le fait et insère un espace vide.
  – Enfin, il revient une fois de plus au point où il a arrêté la lecture (l'espace blanc qui était le symbole de contrôle) et continue à traiter le fichier source à partir de là.

J'ai expliqué ce mécanisme de manière assez détaillée, car l'élimination des espaces vides surprend souvent les nouveaux venus. Il convient toutefois de noter que le problème est relativement mineur, car les mots

de contrôle ne s'impriment généralement pas directement dans le document final, mais en affectent le format et l'apparence. En revanche, il est assez fréquent que les symboles de contrôle s'impriment sur le document final.

Il existe une troisième procédure pour éviter le problème des espaces vides, qui consiste à définir (à la manière de TEX) une commande similaire et à inclure une « non-lettre » à la fin du nom de la commande. Par exemple, la séquence suivante :

```
\def\txt-{\TeX}
```

\def\txt-{\TeX}

créerait une commande appelée `\txt`, qui aurait exactement la même fonction que la commande `\TeX` et ne fonctionnerait correctement que si elle était suivie d'un trait d'union `\txt-`. Ce trait d'union ne fait pas techniquement partie du nom de la commande, mais celle-ci ne fonctionnera que si le nom est suivi d'un trait d'union. La raison de cette situation est liée au mécanisme de définition des macros TEX et est trop complexe pour être expliquée ici. Mais cela fonctionne : une fois cette commande définie, chaque fois que nous utilisons `\txt-`, ConTEXt la remplace par `\TeX` en éliminant le trait d'union, mais en l'utilisant en interne pour savoir que le nom de la commande est déjà terminé, de sorte qu'un espace blanc immédiatement après ne serait pas supprimé.

Cette «astuce» ne fonctionnera pas correctement avec la commande `\define`, qui est une commande spécifiquement ConTEXt pour définir des macros.

## 3.3 Périmètre des commandes

### 3.3.1 Les commandes qui nécessite ou pas une périmètre d'application

De nombreuses commandes ConTEXt en particulier celles qui affectent les fonctions de formatage des polices (gras, italique, petites capitales, etc.), activent une certaine fonction qui reste activée jusqu'à ce qu'une autre commande la désactive ou active une autre fonction incompatible avec elle. Par exemple, la commande `\bf` active le gras, et elle restera

active jusqu'à ce qu'elle trouve une commande *incompatible* comme, par exemple, `\tf`, ou `\it`.

Ces types de commandes n'ont pas besoin de prendre d'argument, car elles ne sont pas conçues pour s'appliquer uniquement à certains textes. C'est comme si elles se limitaient à *activer* une fonction quelconque (gras, italique, sans serif, taille de police donnée, etc.).

Lorsque ces commandes sont exécutées dans un *groupe* (voir section 3.8.1), elles perdent également leur efficacité lorsque le groupe dans lequel elles sont exécutées est fermé. Par conséquent, pour que ces commandes n'affectent qu'une partie du texte, il faut souvent générer un groupe contenant cette commande et le texte que l'on souhaite qu'elle affecte. Un groupe est créé en l'enfermant entre des accolades. Par conséquent, le texte suivant

```
In {\it The \TeX Book},
{\sc Knuth} explained everything
you need to know about \TeX.
```

In *The TEXBook*, KNUTH explained everything you need to know about TEX.

```
In {\it The \TeX Book}, {\sc Knuth} explained \bf{everything} you need to know
about \TeX.
```

In *The TEXBook*, KNUTH explained **everything you need to know about TEX.**

```
In {\it The \TeX Book}, {\sc Knuth}
explained \bf{everything} you need to
know about \TeX.
```

In *The TEXBook*, KNUTH explained **everythir you need to know about TEX.**

```
In {\it The \TeX Book}, {\sc Knuth} explained everything you need to know about \TeX.
```

In *The TEXBook*, KNUTH explained everything you need to know about TEX.

```
In {\it The \TeX Book}, {\sc Knuth}
explained everything you need to know
about \TeX.
```

In *The TEXBook*, KNUTH explained everythin you need to know about TEX.

```
In {\it The \TeX Book}, {\sc Knuth} explained
everything you need to know about \TeX.
```

In *The TEXBook*, KNUTH explained everything you need to know about TEX.

crée deux groupes, l'un pour déterminer la portée de la commande `\it` (italique) et l'autre pour déterminer la portée de la commande `\sc` (petites capitales, small capital en anglais).

Au contraire de ce type de commande, il en existe d'autres qui nécessitent immédiatement une indication du texte auquel elles doivent être appliquées. Dans ce cas, le texte qui doit être affecté par la commande est placé entre des crochets immédiatement après la commande. Par exemple, nous pouvons citer la commande `\framed` : cette commande dessine un cadre autour du texte qu'elle prend comme argument, de telle sorte que

```
\framed{Tweedledum and Tweedledee}
```

produira

Tweedledum and Tweedledee

```
\framed{Tweedledum and Tweedledee}
```

Tweedledum and Tweedledee

Notez que, bien que dans le premier groupe de commandes (celles qui ne requièrent pas d'argument), les accolades sont parfois utilisées pour déterminer le champ d'action, mais cela n'est pas nécessaire pour que la commande fonctionne. La commande est conçue pour être appliquée à partir du point où elle apparaît. Ainsi, lorsque vous déterminez son champ d'application en utilisant des crochets, la commande est placée *entre ces crochets*, contrairement au deuxième groupe de commandes, où les parenthèses encadrant le texte auquel la commande doit être s'appliquent, sont placés après le commandement.

Dans le cas de la commande `\framed`, il est évident que l'effet qu'elle produit nécessite un argument – le texte auquel elle est appliquée. Dans d'autres cas, cela dépend du programmeur si la commande est d'un

type ou d'un autre. Ainsi, par exemple, les commandes `\it` et `\color` sont assez similaires : elles appliquent une caractéristique (format ou couleur) au texte. Mais la décision a été prise de programmer la[1] première sans argument, et la seconde comme une commande avec un argument.

Pas toujours, cela dépend de l'environnement en question et de la situation dans le reste du document. ConTEXt diffère de LATEX à cet égard qui est beaucoup plus stricte.

test

### 3.3.2 Commandes nécessitant d'indiquer leur début et fin d'application (environnements)

Certaines commandes fonctionnent par couple afin de déterminer leur[1] portée, en indiquant précisément le moment où elles commencent à être appliquées et celui où elles cessent de l'être. Ces commandes sont donc présentées par paires : l'une indique le moment où la commande doit être activée, et l'autre celui où cette action doit cesser. La commande « start », suivie du nom de la commande, est utilisée pour indiquer le début de l'action, et la commande « stop », également suivie du nom de la commande, pour indiquer la fin. Ainsi, par exemple, la commande « `itemize` » devient `\startitemize` pour indiquer le début d'une *liste d'items* et `\stopitemize` pour indiquer la fin.

Il n'y a pas de nom spécial pour ces paires de commandes dans la documentation officielle de ConTEXt. Le manuel de référence et l'introduction les appellent simplement « start ... stop ». Parfois elles sont appelés *environnements*, qui est également le nom que LATEX donne à un type de construction similaire, mais cela présente un inconvénient dans ConTEXt car ce terme « environnement » est également utilisé pour autre chose (un type spécial de fichier source que nous verrons lorsque nous parlerons des projets multifichiers dans section ??). Néanmoins, puisque le terme environnement est clair, et que le contexte permettra de distinguer facilement si nous parlons de *commandes d'environnement* ou de *fichiers d'environnement*, j'utiliserai ce terme.

Les environnements consistent donc en une commande qui les ouvre, les commence, et une autre qui les ferme, les termine. Si le fichier source contient une commande d'ouverture d'environnement qui n'est pas fermée par la suite, une erreur est normalement générée.[1] D'autre part, ces types d'erreurs sont plus difficiles à trouver, car l'erreur peut se produire bien au-delà de l'endroit où se trouve la commande d'ouverture. Parfois, le fichier « `.log` » nous montrera la ligne où commence

l'environnement incorrectement fermé ; mais d'autres fois, l'absence d'une fermeture [1] d'environnement va impliquer une mauvaise interprétation par ConTEXt qui soulignera le passage qu'il considère comme éronné et non pas le manque de fermeture d'environnement, ce qui signifie que le fichier « `.log` » ne nous est pas d'une grande aide pour trouver où se situe le problème.

Les environnements peuvent être imbriqués, ce qui signifie qu'un autre environnement peut être ouvert à l'intérieur d'un environnement existant. Dans de tels cas un environnement doit absolument être fermé à l'intérieur de l'environnement dans lequel il a été ouvert. En d'autres termes, l'ordre dans lequel les environnements sont fermés doit être cohérent avec l'ordre dans lequel ils ont été ouverts. Je pense que cela devrait être clair à partir de l'exemple suivant :

```
\startQuelqueChose
  …
  \startQuelqueChoseAutre
    …
    \startEncoreQuelqueChoseAutre
      …
    \stopEncoreQuelqueChoseAutre
  \stopQuelqueChoseAutre
\stopQuelqueChose
```

```
\startQuelqueChose
  …
  \startQuelqueChoseAutre
    …
    \startEncoreQuelqueChoseAutre
      …
    \stopEncoreQuelqueChoseAutre
  \stopQuelqueChoseAutre
\stopQuelqueChose
```

Dans l'exemple, vous pouvez voir comment l'environnement « `QuelqueChoseAutre` » a été ouvert à l'intérieur de l'environnement « `QuelqueChose` » et doit être fermé à l'intérieur de celui-ci également. Dans le cas contraire, une erreur se produirait lors de la compilation du fichier.

En général, les commandes conçues comme *environnements* sont celles qui mettent en œuvre un changement destiné à être appliqué à des

unités de texte au moins aussi grande que le paragraphe. Par exemple, l'environnement « `narrower` » qui modifie les marges, n'a de sens que lorsqu'il est appliqué au niveau du paragraphe, ou l'environnement « `framedtext` » qui encadre un ou plusieurs paragraphes. Ce dernier environnement peut nous aider à comprendre pourquoi certaines commandes sont conçues comme des environnements et d'autres comme des commandes individuelles : si nous souhaitons encadrer un ou plusieurs mots, tous sur la même ligne, nous utiliserons la commande `\framed`, mais si ce que nous voulons encadrer est un paragraphe entier (ou plusieurs paragraphes), nous utiliserons l'environnement « `startframed` » ou « `startframedtext` ».

D'autre part, le texte situé dans un environnement particulier constitue normalement un *groupe* (voir section ??), ce qui signifie que si une commande d'activation est trouvée à l'intérieur d'un environnement, parmi les commandes qui s'appliquent à tout le texte qui suit, cette commande ne s'appliquera que jusqu'à la fin de l'environnement dans lequel elle se trouvei. En fait, ConTEXt a un *environnement* sans nom commençant par la commande `\start` (aucun autre texte ne suit ; juste *start*, c'est pourquoi je l'appelle un *environnement sans nom*) et se termine par la commande `\stop`. Je pense que la seule fonction de cette commande est de créer un groupe.

Je n'ai lu nulle part dans la documentation de ConTEXt que l'un des effets des environnements est de grouper leur contenu, mais c'est le résultat de mes tests avec un certain nombre d'environnements prédéfinis, bien que je doive admettre que mes tests n'ont pas été trop exhaustifs. J'ai simplement vérifié quelques environnements choisis au hasard. Mes tests montrent cependant qu'une telle affirmation, si elle était vraie, ne le serait que pour certains environnements prédéfinis : ceux créés avec la commande `\definestartstop` (expliquée dans la section 3.7.2) ne créent aucun groupe, à moins que lors de la définition du nouvel environnement nous n'incluions les commandes nécessaires à la création du groupe (voir section 3.8.1).

Je suppose également que l'environnement que j'ai appelé le *sans nom* (`\start`) n'est là que pour créer un groupe : il crée effectivement un groupe, mais je ne sais pas s'il a ou non une autre utilité. C'est l'une des commandes non documentées du manuel de référence.

# 3.4 Options de fonctionnement des commandes

## 3.4.1 Commandes qui peuvent fonctionner de différentes façon distrinctes

De nombreuses commandes peuvent fonctionner de plusieurs façons. Dans ce cas, il existe toujours une manière prédéterminée de travailler (une manière par défaut) qui peut être modifiée en indiquant les paramètres correspondant à l'opération souhaitée entre crochets après le nom de la commande.

Un bon exemple est la commande `\framed` mentionnée dans la section précédente. Cette commande dessine un cadre autour du texte qu'elle prend comme argument. Par défaut, le cadre a la hauteur et la largeur du texte auquel il est appliqué, mais nous pouvons indiquer une hauteur et une largeur différentes. Ainsi, nous pouvons voir la différence entre le fonctionnement de la commande `\framed` par défaut :

```
\framed{Tweedledum}
```

Tweedledum

```
\framed{Tweedledum}
```

Tweedledum

et celui d'une version personnalisée :

```
\framed
  [width=3cm, height=1cm]
  {Tweedledum}
```

Tweedledum

```
\framed
  [width=3cm, height=1cm]
  {Tweedledum}
```

Tweedledum

Dans le deuxième exemple, nous avons indiqué entre les crochets une largeur et une hauteur spécifiques pour le cadre qui entoure le texte qu'il prend comme argument. À l'intérieur des crochets, les différentes options de configuration sont séparées par une virgule ; les espaces vides et même les sauts de ligne (à condition qu'il ne s'agisse pas d'un double saut de ligne) entre deux ou plusieurs options, ne sont pas pris en considération afin que, par exemple, les quatre versions suivantes de la même commande produisent exactement le même résultat :

```
\framed[width=3cm,height=1cm]{Tweedledum}

\framed[width=3cm,  height=1cm]{Tweedledum}

\framed
  [width=3cm, height=1cm]
  {Tweedledum}

\framed
  [width=3cm,
   height=1cm]
  {Tweedledum}
```

```
\framed[width=3cm,height=1cm]{Tweedledum}

\framed[width=3cm,   height=1cm]{Tweedledum}

\framed
  [width=3cm, height=1cm]
  {Tweedledum}

\framed
  [width=3cm,
    height=1cm]
  {Tweedledum}

\framed
  [
    width=3cm,
    height=1cm,
  ]
  {Tweedledum}
\
```

Tweedledum

Tweedledum

Tweedledum

Tweedledum

Tweedledum

Il est évident que la version finale est la plus facile à lire : nous pouvons voir du premier coup d'oeil combien d'options utilisées et à quelle contenu s'applique la commande. Dans un exemple comme celui-ci,

avec seulement deux options, cela ne semble peut-être pas si important ; mais dans les cas où il y a une longue liste d'options, si chacune d'entre elles a sa propre ligne dans le fichier source, il est plus facile de *comprendre* ce que le fichier source demande à ConTEXt de faire, et aussi, si nécessaire, de découvrir une erreur potentielle (car il est possible de commenter successivement chaque ligne et donc chaque option). Par conséquent, ce dernier format (ou un format similaire) pour l'écriture des commandes est celui qui est «préféré et conseillé» par les utilisateurs.

Quant à la syntaxe des options de configuration, voir plus loin dans (section 3.5).

### 3.4.2 Les commands qui configurent comment d'autres commandes fonctionnent ( `\setupQuelqueChose`)

Nous avons déjà vu que les commandes qui offrent des options de fonctionnement ont toujours un jeu d'options par défaut. Si l'une de ces commandes est appelée plusieurs fois dans notre fichier source, et que nous souhaitons modifier la valeur par défaut pour toutes ces commandes, plutôt que de modifier ces options à chaque fois que la commande est appelée, il est beaucoup plus pratique et efficace de modifier la valeur par défaut. Pour ce faire, il existe presque toujours une commande dont le nom commence par `\setup`, suivi du nom de la commande dont nous souhaitons modifier les options par défaut.

La commande `\framed` que nous avons utilisée comme exemple dans cette section reste un bon exemple. Ainsi, si nous utilisons beaucoup de cadres dans notre document, mais qu'ils nécessitent tous des mesures précises, il serait préférable de reconfigurer le fonctionnement de `\framed`, en le faisant avec `\setupframed`. Ainsi,

```
\setupframed
  [
  width=3cm,
  height=1cm,
  ]
```

```
\setupframed
  [
  width=3cm,
  height=1cm,
  ]
```

fera en sorte qu'à partir de cette déclaration dans le code source, chaque fois que nous appellerons `\framed`, il générera par défaut un cadre de 3 centimètres de large sur 1 centimètre de haut, sans qu'il soit nécessaire de l'indiquer expressément à chaque fois. Dans le vocabulaire des logiciels de traitement de texte, cela peut être rapproché de la définition d'un élément de style.

Il existe environ 300 commandes dans ConTEXt qui nous permettent de configurer le fonctionnement d'autres commandes. Ainsi, nous pouvons configurer le fonctionnement par défaut de (`\framed`), des listes (« `itemize` »), des titres de chapitre (`\chapter`), ou des titres de section (`\section`), etc.

### 3.4.3 Définir des versions personnalisée de commande configurables ( `\defineQuelqueChose`)

En continuant avec l'exemple du `\framed`, il est évident que si notre document utilise plusieurs types de cadres, chacun avec des mesures différentes, l'idéal serait de pouvoir *prédéfinir* différentes configurations de `\framed`, et de les associer à un nom particulier afin de pouvoir utiliser l'un ou l'autre selon les besoins. Nous pouvons le faire dans ConTEXt avec la commande `\defineframed`, dont la syntaxe est :

`\defineframed[MonCadre][MaConfigurationPourCadre]`

```
\defineframed
  [MonCadre]
  [MaConfigurationPourCadre]
```

où *MonCadre* est le nom attribué au type particulier de cadre à configurer ; et *MaConfigurationPourCadre* est la configuration particulière associée à ce nom.

L'association entre la configuration et le nom se traduit par l'existence d'une nouvelle fonction « `MonCadre` » que nous pourrons l'utiliser dans n'importe quel contexte où nous aurions pu utiliser la commande originale (`\framed`).

Cette possibilité n'existe pas seulement pour le cas concret de la commande `\framed`, mais pour de nombreuses autres commandes. La combinaison de `\defineQuelqueChose` + `\setupQuelqueChose` est un mécanisme qui donne à ConTEXt son extrême puissance et flexibilité. Si nous examinons en détail ce que fait la commande `\defineSomething`, nous constatons que :

- Tout d'abord, elle clone une commande particulière qui supporte toute une série d'option et de configurations. Par cette opération, le clone *hérite* de la commande initiale et de sa configuration par défaut.
- Il associe ce clone au nom d'une nouvelle commande.
- Enfin, il définit une configuration prédéterminée pour le clone, différente de celle de la commande originale.

Dans l'exemple que nous avons donné, nous avons configuré notre cadre spécial « `MonCadre` » en même temps que nous le créions. Mais nous pouvons aussi le créer d'abord et le configurer ensuite, car, comme je l'ai dit, une fois le clone créé, il peut être utilisé là où l'original aurait pu l'être. Ainsi, dans notre exemple, nous pouvons le configurer avec `\setupframed` en indiquant le nom du cadre (framed) que nous voulons configurer. Dans ce cas, la commande `\setup` prendra un nouvel argument avec le nom du cadre à configurer :

```
\defineframed[MonCadre]

\setupframed
  [MonCadre]
  [ ... ]
```

```
\defineframed
  [MonCadre]

\setupframed
  [MonCadre]
  [MaConfigurationPourCadre]
```

## 3.5 Résumé sur la syntaxe des commandes et des options, et sur l'utilisation des crochets et des accolades lors de leur appel.

this section is especially dedicated to LaTeX users, so they can understand the different use of such brackets.

En résumant ce que nous avons vu jusqu'à présent, nous voyons que dans ConTEXt

- Les commandes commencent toujours par le caractère « \ ».
- Certaines commandes peuvent prendre un ou plusieurs arguments.
- Les arguments qui indiquent à la commande *comment* elle doit fonctionner ou qui affectent son fonctionnement d'une manière ou d'une autre, sont introduits entre crochets.
- Les arguments qui indiquent à la commande sur quelle partie du texte elle doit agir sont présentés entre accolades.

  > Lorsque la commande ne doit agir que sur une seule lettre, comme c'est le cas, par exemple, de la commande `\buildtextcedilla` (pour donner un exemple – le « ç » si souvent utilisée en catalan), les accolades autour de l'argument peuvent être omis : la commande s'appliquera au premier caractère qui n'est pas un espace blanc.

- Certains arguments sont facultatifs, auquel cas nous pouvons les omettre. Mais ce que nous ne pouvons jamais faire, c'est changer l'ordre des arguments que la commande attend.

Les arguments introduits entre crochets peuvent être de différent type : un nom symbolique (dont ConTEXt connaît la signification), une mesure ou une dimension, un nombre, le nom d'une autre commande.

Ils peuvent prendre trois forme différentes :.

- une information unique
- une série d'informations uniques, séparées par des virgules
- une série d'informations sous la forme de couple « `clé=valeur` », utilisant pour clé des noms de variables auxquelles il faut donner une valeur. Dans ce cas, la définition officielle de la commande (voir section `??`) fournit un guide utile pour connaître les clés disponibles et le type de valeur attendu pour chacune.

Enfin, il n'arrive jamais avec ConTEXt qu'au sein d'un même argument on mélange le format de déclaration. Nous pouvons donc avoir les cas de figures suivants

```
\commande[Option1, Option2, …]
\commande[Variable1=valeur, Variable2=valeur, …]
\commande[Option1][Variable1=valeur, Variable2=valeur, …]
```

Mais nous n'aurons jamais un mélange du genre :

```
\commande[Option1, Variable1=valeur, …]
```

Certaines règles syntaxiques sont à bien prendre en compte :

- Les espaces et les sauts de ligne entre les différents arguments d'une commande sont ignorés.
- une information utilisée dans l'argument peut contenir des espaces vides ou des commandes. Dans ce cas, il est fortement conseillée de la placer entre accolades.
- Les espaces et les sauts de ligne (autres que les doubles) entre les différentes informations sont ignorés.
- Par contre, et ceci est une erreur très commune, entre la première lettre de la clé et la virgule indiquant la fin du couple « `clé=valeur` », les espaces ne sont pas ignorés. Les règles syntaxiques consistent donc à juxtaposer sans aucun espace le mot clé, le signe égal, la valeur et la virgule. Pour prendre en compte des espaces

dans la valeur, la pratique est encore une fois de la mettre entre accolades.

- Nous devons également inclure le contenu de la valeur entre accolades si elle intègre elle-même des crochets. Sinon le premier crochet fermant sera considéré comme fermant non seulement la valeur mais aussi l'argument que nous sommes en train de définir. Voyez :

```
\startsection[title=mon titre[5] avec crochets]
  Du texte pour cette section
FONCTIONNERA PAS
\stopsection
\startsection[title={mon titre[5] avec crochets}]
  Du texe pour cette section
\stopsection
```

1 mon titre[5

avec crochets] Du texte pour cette section NE FONCTIONNERA PAS

2 mon titre[5] avec crochets

Du texe pour cette section FONCTIONNERA

## 3.6 La liste officielle des commandes ConTEXt

Parmi la documentation de ConTEXt il existe un document particulièrement important contenant la liste de toutes les commandes, et indiquant pour chacune d'entre elles combien d'arguments elles attendent et de quel type, ainsi que les différentes options possibles et leurs valeurs autorisées. Ce document s'appelle « `setup-en.pdf` », et est généré automatiquement pour chaque nouvelle version de ConTEXt. Il se trouve dans le répertoire appelé « `tex/texmf-context/doc/context/documents/general/qrcs` ».

> En fait, la « `qrc` » possède sept versions de ce document, une pour chacune des langues disposant d'une interface ConTEXt : allemand, tchèque, français, néerlandais, anglais, italien et roumain. Pour chacune de ces langues, il existe deux documents dans le répertoire : un appelé « `setup-LangCode` » (où LangCode est le code en deux lettres d'identification des langues internationales) et un second document appelé « `setup-mapping-LangCode` ». Ce second document contient une liste de commandes par ordre alphabétique et indique la commande *prototype*, mais sans les informations des valeurs possibles pour chaque argument.

Ce document est fondamental pour apprendre à utiliser ConTEXt, car c'est là que nous pouvons savoir si une certaine commande existe ou non ; ceci est particulièrement utile, compte tenu de la combinaison COMMANDE (ou ENVIRONNEMENT) + setupCOMMANDE + defineCOMMANDE. Par exemple, si je sais qu'une ligne vierge est introduite avec la commande `\blank`, je peux savoir s'il existe une commande appelée `\setupblank` qui me permet de la configurer, et une autre qui me permet d'établir une configuration personnalisée pour les lignes vierges, (`\defineblank`).

> « `setup-fr.pdf` » est donc fondamental pour l'apprentissage de ConTEXt. Mais je préférerais vraiment, tout d'abord, qu'il nous dise si une commande ne fonctionne que dans Mark II ou Mark IV, et surtout, qu'au lieu de nous indiquer seulement la liste et le type d'arguments que chaque commande autorise, il nous dise à quoi servent ces arguments. Cela réduirait considérablement les lacunes de la documentation de la ConTEXt. Certaines commandes autorisent des arguments facultatifs que je ne mentionne même pas dans cette introduction parce que je ne sais pas à quoi ils servent et, puisqu'ils sont facultatifs, il n'est pas nécessaire de les mentionner. C'est extrêmement frustrant.
>
> La méthode mise en oeuvre par la communauté ConTEXt est dorénavant de documenter tout cela dans le Wiki avec une adresse web spécifique pour chaque commande, par exemple pour `\setupframed` : https://wiki.contextgarden.net/index.php?title=Command/setupframed
>
> Ainsi, chaque utilisateur est invité à compléter progressivement la documentation au fil de ses découvertes, souvent issues des échanges sur la liste de diffusions NTG-context où les développeurs demanderont à Wikifier les réponses apportées.

## 3.7 Définir de nouvelles commandes

### 3.7.1 Mécanisme général pour définir de nouvelles commandes

Nous venons de voir comment, avec `\defineQuelqueChose`, nous pouvons cloner une commande préexistante et développer une nouvelle version de celle-ci qui à partir de là, fonctionnera comme une nouvelle commande.

En plus de cette possibilité, qui n'est disponible que pour certaines commandes spécifiques (quelques-unes, certes, mais pas toutes), ConTEXt a un mécanisme général pour définir de nouvelles commandes qui est extrêmement puissant mais aussi, dans certaines de ses utilisations, assez complexe. Dans un texte comme celui-ci, destiné aux débutants, je pense qu'il est préférable de le présenter en commençant par certaines de ses utilisations les plus simples. La plus simple de toutes est d'associer des bouts de texte à un mot, de sorte que chaque fois que ce mot apparaît dans le fichier source, il est remplacé par le texte qui lui est lié. Cela nous permettra, d'une part, d'économiser beaucoup de temps de frappe et, d'autre part, comme avantage supplémentaire, de réduire les possibilités de faire des erreurs de frappe, tout en s'assurant que le texte en question est toujours écrit de la même façon.

Imaginons, par exemple, que nous sommes en train d'écrire un traité sur l'allitération dans les textes latins, où nous citons souvent la phrase latine « *O Tite tute Tati, tibi tanta, tyranne, tulisti !* ». (C'est toi-même, Titus Tatius, qui t'es fait, à toi, tyran, tant de torts !). Il s'agit d'une phrase assez longue dont deux des mots sont des noms propres et commencent par une majuscule, et où, avouons-le, même si nous aimons la poésie latine, il nous est facile de « trébucher » en l'écrivant. Dans ce cas, nous pourrions simplement mettre dans le préambule de notre fichier source :

```
\define\Tite{\quotation{O Tite tute Tati, tibi tanta, tyranne, tulisti}}
```

Sur la base d'une telle définition, chaque fois que la commande `\Tite` apparaîtra dans notre fichier source, elle sera remplacée par la séquence indiquée : la phrase elle même, prise comme argument de la commande `\quotation` qui met son argument entre guillemets en respectant les règles typographiques de la langue du document. Cela nous permet de garantir que la façon dont cette phrase apparaîtra sera toujours la même. Nous aurions également pu l'écrire en italique, avec une taille de police plus grande... comme bon nous semble. L'important, c'est que nous ne devons l'écrire qu'une seule fois et qu'elle sera reproduite dans tout le texte exactement comme elle a été écrite, aussi souvent que nous le voulons. Nous pourrions également créer deux versions de la commande, appelées `\Tite` et `\tite`, selon que la

phrase doit être écrite en majuscules ou non. De plus, il suffira de modifier la définition et elle sera répercutée automatiquement dans l'ensemble du document.

Le texte de remplacement peut être du texte pur, ou inclure des commandes, ou encore former des expressions mathématiques dans lesquelles il y a plus de chances de faire des fautes de frappe (du moins pour moi). Par exemple, si l'expression $(x_1, ..., x_n)$ doit apparaître régulièrement dans notre texte, nous pouvons créer une commande pour la représenter. Par exemple

```
\define\xvec{$(x_1,\ldots,x_n)$}
```

de sorte que chaque fois que `\xvec` apparaît dans le code source, il sera remplacé par l'expression qui lui est associée durant la compilation par ConTEXt.

La syntaxe générale de la commande `\define` est la suivante :

```
\define[NbrArguments]\NomCommande{TexteOuCodeDeSubstitution}
```

où

- **`NbreArguments`** désigne le nombre d'arguments que la nouvelle commande prendra. Si elle n'a pas besoin d'en prendre, comme dans les exemples donnés jusqu'à présent, elle est omise.
- **`NomCommande`** désigne le nom que portera la nouvelle commande. Les règles générales relatives aux noms de commande s'appliquent ici. Le nom peut être un caractère unique qui n'est pas une lettre, ou une ou plusieurs lettres sans inclure de caractère « non-lettre ».
- **`TexteOuCodeDeSubstitution`** contient le texte ou le code source qui sera substitier à la commande à chacune des ses occurences dans le fichier source.

La possibilité de fournir aux nouvelles commandes des arguments dans leur définition confère à ce mécanisme une grande souplesse, car elle permet de définir un texte de remplacement variable en fonction des arguments pris.

Par exemple : imaginons que nous voulions écrire une commande qui produise l'ouverture d'une lettre commerciale. Une version très simple de cette commande serait la suivante

```
\define\EnTetedeLettre{
  \rightaligned{Anne Smith}\par
  \rightaligned{Consultant}\par
  Marseille, \date\par
  Chère Madame,\par}
\EnTetedeLettre
```

Anne Smith
Consultant

Marseille, 16 mai 2021
Chère Madame,

mais il serait préférable d'avoir une version de la commande qui écrirait le nom du destinataire dans l'en-tête. Cela nécessiterait l'utilisation d'un paramètre qui communiquerait le nom du destinataire à la nouvelle commande. Il faudrait donc redéfinir la commande comme suit :

```
\define[1]\EnTetedeLettre{
  \rightaligned{Anne Smith}\par
  \rightaligned{Consultant}\par
  Marseille, \date\par
  Chère Madame #1,\par}
\EnTetedeLettre{Dupond}
```

Anne Smith
Consultant

Marseille, 16 mai 2021
Chère Madame Dupond,

Notez que nous avons introduit deux changements dans la définition. Tout d'abord, entre le mot clé `\define` et le nouveau nom de la commande, nous avons inclus un 1 entre crochets ([1]). Cela indique à ConTEXt que la commande que nous définissons prendra un argument.

Plus loin, à la dernière ligne de la définition de la commande, nous avons écrit « `Chère Madame #1` », en utilisant le caractère réservé « `#` ». Cela indique qu'à l'endroit du texte de remplacement où apparaît « `#1` », le contenu du premier argument sera inséré.

Si elle avait deux paramètres, « `#1` » ferait référence au premier paramètre et « `#2` » au second. Afin d'appeler la commande (dans le fichier source) après le nom de la commande, les arguments doivent être inclus entre accolades, chaque argument ayant son propre ensemble. Ainsi, la commande que nous venons de définir doit être appelée de la manière suivante : « `\EnTetedeLettre{Nom du destinataire}` », tel que cela est fait dans l'exemple.

Nous pourrions encore améliorer la fonction précédente, car elle suppose que la lettre sera envoyée à une femme (elle met « chère Madame »), alors que nous pourrions peut-être inclure un autre paramètre pour distinguer les destinataires masculins et féminins. par exemple :

```
\define[2]\EnTetedeLettre{
  \rightaligned{Anne Smith}\par
  \rightaligned{Consultant}\par
  Marseille, \date\par
  #1\ #2,\par}
\EnTetedeLettre{Cher Monsieur}{Antoine Dupond}
```

Anne Smith
Consultant

Marseille, 16 mai 2021
Cher Monsieur Antoine Dupond,

bien que cela ne soit pas très élégant (du point de vue de la programmation). Il serait préférable que des valeurs symboliques soient définies pour le premier argument (homme/femme ; 0/1 ; m/f) afin que la macro elle-même choisisse le texte approprié en fonction de cette valeur. Mais pour expliquer comment y parvenir, il faut aller plus en profondeur que ce que je pense que le lecteur novice peut comprendre à ce stade.

## 3.7.2 Création de nouveaux environments

Pour créer un nouvel environnement, ConTEXt fournit la commande `\definestartstop` dont la syntaxe est la suivante :

```
\definestartstop[Nom][Configuration]
```

Dans la définition *officielle* de `\definestartstop` (voir section ??) il y a un argument supplémentaire que je n'ai pas mis ci-dessus parce qu'il est optionnel, et je n'ai pas été capable de trouver à quoi il sert . Ni le manuel d'introduction « ConTEXt Mark IV, an Excursion », ni le manuel de référence ne l'expliquent. J'avais supposé que cet argument (qui doit être saisi entre le nom et la configuration) pouvait être le nom d'un environnement existant qui servirait de modèle initial pour le nouvel environnement, mais mes tests montrent que cette hypothèse était fausse. J'ai consulté la liste de diffusion ConTEXt et je n'ai vu aucune utilisation de cet argument possible.

où

- **Nom** est le nom que portera le nouvel environnement.

- **Configuration** nous permet de configurer le comportement du nouvel environnement. Nous disposons des valeurs suivantes avec lesquelles nous pouvons le configurer :
  - `before` : Commandes à exécuter avant d'entrer dans l'environnement.
  - `after` : Commandes à exécuter après avoir quitté l'environnement.
  - `style` : Style que doit avoir le texte du nouvel environnement.
  - `setups` : Ensemble de commandes créées avec `\startsetups ... \stopsetups`. Cette commande et son utilisation ne sont pas expliquées dans cette introduction.
  - `color` : Couleur à appliquer au texte
  - `inbetween, left, right` : Options non documentées que je n'ai pas réussi à faire fonctionner . D'après les tests que j'ai effectués, indiquant une certaine valeur pour ces options, je ne vois aucun changement dans l'environnement. Il est possible que l'impact ne concerne pas l'environnement mais la commande qui semble créer avec `\Nom`. Une piste sur la liste de diffusions NTG-context.

Un exemple de la définition d'un environnement pourrait être le suivant :

Si nous voulons que notre nouvel environnement soit un groupe (section 3.8.1), de sorte que toute altération du fonctionnement normal de ConTEXt qui se produit en son sein disparaisse en quittant l'environnement, nous devons inclure la commande `\bgroup` dans l'option « `before` », et la commande `\egroup` dans l'option « `after` ».

## 3.8 Other fundamental concepts

There are other notions, other than commands, that are fundamental to understanding the logic behind how ConTEXt works. Some of them,

```
\definestartstop
  [TextWithBar]
  [before=\bgroup\startmarginrule\noindentation,
    after=\stopmarginrule\egroup,
    style=\ss,
    color=darkyellow]

\starttext
The first two fundamental laws of human stupidity state unambiguously
that:
\startTextWithBar
  \startitemize[n,broad]
    \item Always and inevitably we underestimate the number of stupid
    individuals in the world.
    \item The probability that a given person is stupid is independent
    of any other characteristic of the same person.
  \stopitemize
\stopTextWithBar
\stoptext
```

The first two fundamental laws of human stupidity state unambiguously that:

1. Always and inevitably we underestimate the number of stupid individuals in the world.
2. The probability that a given person is stupid is independent of any other characteristic of the same person.

because of their complexity, are not appropriate for an introduction and therefore will not be dealt with in this document; but there are two notions that should be examined now: groups and dimensions.

### 3.8.1 Groups

A group is a well-defined fragment of the source file that ConTEXt uses as a *working unit* (what this means is explained shortly). Every group has a beginning and end that needs to be expressly indicated. A group begins:

- With the reserved character « { » or with the command `\bgroup`.
- With the command `\begingroup`
- With the command `\start`
- With the opening of certain environments (`\startSomething` command).
- By beginning a maths environment (with the reserved character «$»).

and is closed

- With the reserved character « } » or with the command \egroup. [1]
- With the command \endgroup
- With the command \stop
- With the closing of the environment (\stopSomething command).
- On leaving the maths environment (with the reserved character «$»).

[1] The *box* notion is also a central ConTEXt one, but its explanation is not included in this introduction.

Certain commands also automatically generate a group, for example, \hbox, \vbox and, in general, commands linked to the creation of *boxes*[1]. Outside of these latter cases (groups automatically generated by certain commands), the way of closing a group has to be consistent with the way it is opened. This means that a group that is begun with « { » must close with « } », and a group begun with \begingroup must be closed with \endgroup. This rule has only one exception, that a group begun with « { » can be closed with \egroup, and the group begun with \bgroup can be closed with « } »; in reality, this means that « { » and \bgroup are completely synonymous and interchangeable, and similarly for « } » and \egroup.

> The commands \bgroup and \egroup were designed to be able to define commands to open a group and others to close a group. Therefore, for reasons internal to TEX syntax, those groups could not be opened and closed with curly brackets, since this would generate unbalanced curly brackets in the source file, and this would always throw an error when compiling.
>
> The commands \begingroup and \endgroup, by contrast, are not interchangeable with curly brackets or the \bgroup ... \egroup commands, since a group begun with \begingroup has to be closed with \endgroup. These latter commands were designed to allow for much more in-depth error checking. In general, normal users do not have to use them.

We can have nested groups (a group within another group), and in this case the order in which groups are closed must be consistent with the order in which they were opened: any subgroup has to be closed within the group in which it began. There can also be empty groups generated with « {} ». An empty group, in principle, has no effect on the final document, but it can be useful, for example, for indicating the end of a command name.

The main effect of the groups is to encapsulate their content: as a rule, the definitions, formats and value assignments that are made within a group are « forgotten » once we leave the group. This way, if we want ConTEXt to temporarily alter its normal way of functioning, the most efficient way is to create a group and, within it, alter that functioning. Thus, when we leave the group, all the values and formats previous to it will be restored. We have already seen some examples of this when mentioning commands like `\it`, `\bf`, `\sc`, etc. But this doesn't only happen with format commands: the group in a way *isolates* its contents, so that any change in any of the many internal variables that ConTEXt is constantly managing, will remain effective only as long as we are within the group in which that change took place. Likewise, a command defined within a group will not be known outside it.

So, if we process the following example

```
\define\A{B}
\A
{
  \define\A{C}
  \A
}
\A
```

we will see that the first time we run the command `\A`, the result corresponds to that of its initial definition («B»). Then we created a group and redefined the command `\A` within it. If we run it now within the group, the command will give us the new definition («C» in our example), but when we leave the group in which the command `\A` was redefined, if we run it again it will type «B» once more. The definition made within the group is « forgotten » once we have left it.

Another possible use of the groups concerns those commands or instructions designed to apply exclusively to the character that is written after them. In this case, if we want the command to be applied to more than one character, we must enclose the characters we want the command or instruction to be applied to, in a group. So for example, the reserved « `^` » character which, we already know, converts the following character into a superscript when used inside the maths environment; so if we write, for example, « `$4^2x$` » we will get « $4^2x$ ». But if we write « `$4^{2x}$` » we will get « $4^{2x}$ ».

Finally: a third use of grouping is to tell ConTEXt that what is enclosed within the group must be treated as one. This is the reason why before (section 3.5) it was said that on certain occasions it is better to enclose the contents of some command option within curly brackets.

### 3.8.2 Dimensions

Although we could use ConTEXt perfectly without worrying about dimensions, we would not be able to make use of all the configuration possibilities without giving them some consideration. Because to a large extent the typographical perfection achieved by TEX and its derivatives lies in the great attention that the system pays internally to dimensions. Characters have dimensions; the space between words, or between lines, or between paragraphs have dimensions; lines have dimensions; margins, headers and footers. For almost every element on the page we can think of there will be some dimension.

In ConTEXt dimensions are indicated by a decimal number followed by the unit of measurement. The units that can be used are found in table 3.2.

| **Name** | **Name in ConTEXt** | **Equivalent** |
|---|---|---|
| Inch | in | 1 in = 2.54 cm |
| Centimetre | cm | 2.54 cm = 1 inch |
| Millimetre | mm | 100 mm = 1 cm |
| Point | pt | 72.27 pt = 1 inch |
| Big point | bp | 72 bp = 1 inch |
| Scaled point | sp | 65536 sp = 1 point |
| Pica | pc | 1 pc = 12 points |
| Didot | dd | 1157 dd = 1238 points |
| Cicero | cc | 1 cc = 12 didots |
| | ex | |
| | em | |

**Tableau 3.2** Units of measurement in ConTEXt

The first three units in the table 3.2 are standard measures of length; the first is used in some parts of the English-speaking world and the others outside it or in some parts of it. The remaining units come from the world of typography. The last two, for which I have put no equivalent,

are relative units of measurement based on the current font. 1 « em » is equal to the width of an « M » and an « ex » is equal to the height of an « x ». The use of measures related to font size allows the creation of macros that look just as good whatever the source used at any given moment. That is why, in general, it is recommended.

With very few exceptions, we can use any unit of measurement we prefer, as ConTEXt will convert it internally. But whenever a dimension is indicated, it is compulsory to indicate the unit of measurement, and even if we want to indicate a measurement of « 0 », we have to say «0pt» or «0cm». Between the number and the name of the unit, we may or may not leave a blank space. If the unit has a decimal part, we can use a decimal separator, either the (.) or the comma (,).

The measurements are usually used as an option for some command. But we can also directly assign a value to some internal measure of ConTEXt as long as we know the name of it. For example, indentation of the first line of an ordinary paragraph is internally controlled by ConTEXt with a variable called `\parindent`. By expressly assigning a value to this variable we will have altered the measurement that ConTEXt uses from that point on. And so, for example, if we want to eliminate the indentation of the first line, we only need to write in our source file:

```
\parindent=0pt
```

We could also have written `\parindent 0pt` (without the equal sign) or `\parindent0pt` with no space between the name of the measure and its value.

However, assigning a value directly to an internal measure is considered « inelegant ». In general, it is recommended to use the commands that control that variable, and to do so in the preamble of the source file. The opposite results in source files that are very difficult to debug because not all the configuration commands are in the same place, and it is really difficult to obtain a certain consistency in typographical characteristics.

Some of the dimensions used by ConTEXt are « elastic », that is, depending on the context, they can take one or other measure. These measures are assigned with the following syntax:

```
\MeasureName plus MaxIncrement minus MaxDecrease
```

```
\MeasureName plus MaxIncrement minus MinDecrease
```

For example

```
\parskip 3pt plus 2pt minus 1pt
```

With this instruction we are telling ConTEXt to assign to `\parskip` (indicating the vertical distance between paragraphs) a *normal* measurement of 3 points, but that if the composition of the page requires it, the measurement can be up to 5 points (3 plus 2) or only 2 points (3 minus 1). In these cases it will be left to ConTEXt to choose the distance for each page between a minimum of 2 points and a maximum of 5 points

## 3.9 Self-learning method for ConTEXt

Section added at the last moment, when I realised that I myself had become so imbued with the spirit of ConTeXt that I was able to guess the existence of certain commands.

The huge quantity of ConTEXt commands and options turns out to be truly overwhelming and can give us the feeling that we will never end up learning to work well with it. This impression would be a mistake, because one of the advantages of ConTEXt is the uniform way it handles all its structures: learning well a few structures, and knowing, more or less, what the remaining ones are for, when we need some extra utility it will be relatively easy learn to use it. Therefore, I think of this introduction as a kind of *training* that will prepare us to make our own investigations.

To create a document with ConTEXt it is probably only essential to know the following five things (we could call them the ConTEXt *Top Five*):

1. Know how to create the source file or project of any; this is explained in Chapter ?? of this introduction.

2. Set the main font for the document, and know the basic commands to change font and colour (Chapter ??).

3. Know the basic commands for structuring the content of our document, such as chapters, sections, subsections, etc. This is all explained in Chapter ??.

4. Perhaps know how to handle the *itemize* environment explained in some detail in section ??.

5. ... and little else.

For the rest, all we need to know is that it is possible. Certainly no one will use a utility if they do not know that it exists. Many of them are explained in this introduction; but, above all, we can repeatedly watch how ConTEXt always acts when faced with a certain type of construction:

- First there will be a command that allows it to do so.

- Second, there is almost always a command that allows us to configure and predetermine how the task will be carried out; a command whose name starts with `\setup` and usually coincides with the basic command.

- Finally, it is often possible to create a new command to perform similar tasks, but with a different configuration.

To see whether these commands exist or not, look up the official list of commands (see section 3.6), which will also inform us of the configuration options that these commands support. And although at first glance the names of these options may seem *cryptic*, We will soon see that there are options that are repeated in many commands and that work the same or very similarly in all of them. If we have doubts about what an option does, or how it works, it will be enough to generate a document and test it. We can also look at the abundant ConTEXt documentation. As is common in the world of free software, « ConTEXt Standalone » includes the sources of almost all its documentation in the distribution. A utility like « `grep` » (for GNU Linux systems) can help us search whether the command or option that we have doubts about is used in any of these source files so that we can have an example on hand.

This is how learning ConTEXt has been conceived: the introduction explains in detail the five (actually four) aspects that I have highlighted, and many more: as we read, a clear picture of the sequence will form in our minds: *a command to carry out the task – a command that configures the previous one – a command that allows us to create a similar command*. We will also learn some of the main structures of ConTEXt, and we will know what they are for.

# II
# Global aspects of the document

# III
# Particular issues

# Appendices