Cahiers GUTenberg

ESQUISSE D'UNE TYPOGRAPHIE APPLICATIVE

Emmanuel Saint-James

Cahiers GUTenberg, nº 17 (1994), p. 3-19.

<http://cahiers.gutenberg.eu.org/fitem?id=CG_1994___17_3_0>

Esquisse d'une typographie applicative

Emmanuel Saint-James

Bull – Département ISM
BP 68
78340 Les Clayes sous Bois
E-mail : Emmanuel.Saint-James@frcl.bull.fr

1. Introduction

Lorsque le spécialiste des langages de programmation observe les outils de traitement de textes informatisés, il ne peut qu'être frappé par la ressemblance d'évolution de ces deux problématiques. Par delà la différence des résultats escomptés, là une *valeur*, ici un *document*, apparaît en effet la même difficulté à concevoir des systèmes où les besoins de l'utilisateur priment sur les contraintes du matériel. Mais la programmation étant antérieure au traitement de texte, les deux évolutions n'en sont pas au même stade : le traitement du multilinguisme, en particulier [1], révèle des contorsions déjà évacuées des langages de programmation depuis une trentaine d'années.

Le présent article se propose donc de tirer parti de ce décalage pour esquisser le traitement de texte du futur. Cette esquisse n'est pour l'instant qu'une prospective, non une expérimentation. Mais c'est justement l'une des premières leçons que les outils de traitement de textes informatisés doivent tirer des langages de programmation : concevoir une architecture rationnelle englobant toutes les fonctionnalités souhaitées, plutôt que d'en accumuler des implémentations aux interactions malcommodes voire imprévisibles.

Les critiques contenues dans cet article ne se veulent évidemment pas un réquisitoire contre les traitements de texte actuels : leur viabilité était suffisamment peu sûre au départ pour que leur réalisation ne s'encombre pas d'idées encore mal maîtrisées en programmation. Si l'on veut absolument

1. Voir à ce sujet les *Commentaires sur la portabilité de TEX* de Daniel Taupin dans le numéro 15 des *Cahiers GUTenberg* d'avril 1993.

y voir un procès, ce serait plutôt contre les chercheurs en programmation (et en tant que membre de cette communauté, l'auteur de ces lignes plaide coupable) qui, au lieu d'*exposer des concepts scientifiques universels* applicables à des pratiques en perpétuelle évolution, ont *imposé des produits technologiques datés*, exigeant une totale soumission aux limites de la dernière théorie à la mode. Mais il est plus sage de renvoyer tout le monde dos à dos et de reconnaître que chacun contribue à l'édifice en apportant une pierre que le temps polira. On exposera donc successivement :

1. une perspective historique de quelques acquis en programmation ;
2. une perspective historique des outils de traitements de texte informatisés ;
3. une analyse des manuels de typographie classique à travers ces acquis ;
4. une architecture *fonctionnelle* du traitement de texte.

2. Petite histoire de la programmation

2.1. Langage

Programmer, c'est répondre successivement à trois questions : *quoi ? comment ?* et *où ?* autrement dit, quel but a le programme, quels algorithmes utilise-t-il, quel espace, en place et en temps, nécessite-t-il ? Il est aujourd'hui possible d'écrire des programmes répondant seulement à la première ou à la seconde de ces questions, le système prenant la suite à sa charge. On distingue ainsi trois sortes de programmes :

- un programme est *déclaratif* lorsque le calcul est exprimé par ses propriétés ;
- un programme est *applicatif* lorsque le calcul est exprimé par des fonctions ;
- un programme est *impératif* lorsque le calcul est exprimé par des états.

Mais cette succession logique, qui dépasse d'ailleurs le cadre des langages de programmation, inverse leur succession historique. Après l'époque des

assembleurs puis des *macro-assembleurs*, langages impératifs par essence, sont apparus trois familles de *langages évolués*, calquées sur les trois sortes de programmes.

Les langages impératifs, aussi nommés *procéduraux*, sont fondés sur l'affectation du matériel utilisé. Exclusivement mécanographique pendant trois siècles, essentiellement électronique depuis trois décennies, ce matériel a pour modèle mathématique la *théorie des automates*. Ces langages, apparus au début des années 50, ont pour plus vieux représentant encore utilisé FORTRAN.

Les langages applicatifs sont aussi nommés *fonctionnels*, par une traduction littérale qui ne conserve malheureusement pas le sens de *langage de fonctions*, laissant naïvement entendre que les autres langages ne remplissent pas leur fonction. Le calcul s'y effectue par appel d'une fonction avec ses arguments qui peuvent eux-mêmes être des appels de fonctions et ainsi de suite, jusqu'à trouver des *données*. Cette formulation induit une découpe du problème posé en sous-problèmes indépendants. Dès lors, les mises au point et les mises à jour sont elles aussi indépendantes les unes des autres, permettant une évolution d'autant plus rapide des programmes, que la gestion des résultats intermédiaires est prise en charge par le système. Le modèle mathématique original de cette famille est le *λ-calcul*, son plus vieux représentant, diffusé vers 1960, LISP.

Les langages déclaratifs sont aussi nommés *logiques*, par une traduction littérale encore plus malheureuse que la précédente. Ici, le système prend à sa charge non seulement les résultats intermédiaires mais l'algorithme les calculant, en parcourant l'ensemble des solutions potentielles, à la recherche de celles recevables. Son modèle mathématique est la *théorie de l'unification*, son plus vieux représentant, apparu dans les années 70, PROLOG.

2.2. Utilisation

Au départ, les langages applicatifs et déclaratifs furent *interprétés*. Ce mode de fonctionnement suscita les premiers outils d'*interaction*, notamment les *environnements de programmation* tels que *correcteurs*, *éditeurs dirigés par la syntaxe* et *traces*. Mais la volonté de rivaliser avec l'efficacité des langages impératifs privilégia par la suite leur *compilation*.

Malgré les trésors d'astuce inventés, notamment dans la *récupération*, la *dérécursion* et la *déforestation*, il faut reconnaître aujourd'hui qu'hormis pour les jeux de tests canoniques bénéficiant d'une attention démesurée, le code produit par ces compilateurs reste plus lent que celui de ses concurrents, le facteur multiplicatif s'exprimant en puissance de 10. Échec prévisible : l'absence de description de la mémoire, plus encore de l'algorithme, est un handicap rédhibitoire, une perte d'informations irréparable dans le cas général. Quant à la plus grande facilité de preuve des programmes, elle s'est révélée négligeable face aux milliers de lignes des programmes actuels.

Néanmoins, le privilège accordé par ces langages au *temps de programmation*, plutôt qu'au *temps d'exécution*, est l'attitude adéquate pour la construction de *maquettes* ou de *prototypes*, d'autant que l'ajout de constructions impératives a tempéré leur principal défaut. L'apparition tardive d'environnements de programmation adaptés aux langages impératifs a moins entamé ce statut inattendu de *langages de prototypage* qu'il n'a brouillé les cartes, les influences réciproques, en particulier avec la notion de *procédure*, étant à présent maximales.

D'autre part et surtout, leur rapidité d'interprétation reste leur apanage, les désignant comme meilleurs *langages de personnalisation*. L'exemple le plus célèbre est l'éditeur EMACS, extensible grâce à un interprète LISP. Ce statut, favorisé de surcroît par la gratuité d'EMACS, fait de LISP le langage applicatif le plus utilisé, et certainement pour longtemps encore.

2.3. Influence

La sous-utilisation des langages applicatifs et déclaratifs a aussi pour cause un apprentissage assez long, particulièrement inacceptable en milieu industriel. Et le retour sur cet investissement en temps est d'autant moindre qu'une fâcheuse habitude a été prise, celle de créer un nouveau langage à chaque nouvelle idée, empêchant ainsi l'accumulation des acquis. Cette attitude n'était pas neuve : alors que l'*allocation dynamique* ou l'*appel système* auraient pu être intégrés dans FORTRAN sans trop de difficultés, d'autres langages impératifs ont été créés spécialement pour eux. Avec un tel précédent, il n'est pas surprenant que la *semi-unification* ou le *typage polymorphe* n'aient pas été intégrés à LISP. En conséquence, les apports en programmation se sont moins traduits par une diffusion massive de nou-

veaux langages que par leur influence : il est devenu plus coûteux de les apprendre que de réimplémenter leurs innovations.

La première d'entre elles est la structure de données *arborescente*, qui s'est banalisée jusqu'aux répertoires de fichiers des micro-ordinateurs les plus petits. La récursivité, naguère considérée comme une aberration perverse, est aujourd'hui communément admise.

La deuxième est l'assimilation d'une commande à une fonction : une commande reçoit un *flux d'entrée* et retourne un *flux de sortie*, d'où une écriture des programmes par composition de programmes existants, non par description exhaustive de l'évolution d'une mémoire. Ces *processus en pipe-lines* se sont d'autant plus facilement imposés que le système de *temps partagé* qui les implémente partage les ressources non en fonction du nombre d'utilisateurs, mais en fonction du nombre de processus, ce qui constitue une prime au savoir très motivante.

Enfin, l'adaptation de ces pipe-lines à des *socquettes* connectant *clients* et *serveurs* sur un *réseau* a définitivement consacré la conception de programmes par fonctions indépendantes, aux lieux d'exécution indéterminés. Aux gros systèmes imposant rapidement à l'utilisateur de tronçonner ses données, se substitue une collection de petits processus morcelant automatiquement le flux des données. Il devient alors possible de changer l'une des fonctionnalités de ce *système distribué* sans que les autres en soient perturbées.

Cette nouvelle place centrale accordée à la *fonctionnalité* est particulièrement féconde dans l'exemple du *fenêtrage à distance*[2]. Alors que les systèmes de fenêtres monolithiques imposent ce que l'on peut dénommer une unique *police de fenêtres*, ici l'on dispose d'un *gestionnaire de fenêtres* offrant tous les comportements et toutes les décorations. Et ce n'est pas un hasard si l'un d'entre eux se programme en Lisp[3].

2. Voir de Robert Sheifler & James Gettys, *X window System*, Digital Press, 1990.

3. Voir de Colas Nahaboo, *L'utilisation de* Lisp *dans la personnalisation des gestionnaires de fenêtres*, actes des JFLA, numéro 72 de *Bigre* 1991.

3. L'informatisation du traitement de texte

3.1. De l'instruction à l'abstraction

La ressemblance entre l'évolution esquissée ci-dessus, et celle des outils de traitements de texte est particulièrement frappante.

Les premiers d'entre eux[4] s'utilisaient en écrivant un document unique, où se mêlent texte à imprimer et instructions pour son impression. On a clairement affaire ici à un *langage d'assemblage* : absence de structures de données et de contrôle, amalgame des programmes et des données, effet global des instructions. Même la contrainte syntaxique d'une commande par ligne y figurait, avant l'apparition de la *macro*. Et ce timide début de structuration, en ne cherchant pas à abstraire la représentation des données, conduit à des comportements erronés dès qu'une macro en appelle une autre ou qu'un préprocesseur est utilisé. Ces *macro-assembleurs* servent donc seulement à écrire plus vite les mêmes incommodités.

Avec la génération suivante[5], les traitements de textes ont atteint un niveau à peu près comparable à ceux des premiers langages évolués impératifs : séparation presque maximale des instructions et des données, assemblage séparé de parties du résultat final, typologie, contrôle d'erreurs, mise au point interactive. On y trouve même de la *semi-unification* dans le passage d'arguments. Mais elle rend malheureusement encore plus problématique l'émergence d'un découpage fonctionnel du traitement de texte selon la séquence désormais classique des compilateurs de langages évolués : analyse lexicale, analyse syntaxique, analyse typologique, production de code.

Plus récemment, apparaît enfin l'*abstraction des données*[6] : la structure des textes est décrite indépendamment de leur contenu, ce qui prélude à des contrôles statiques proches du *typage* des langages de programmation.

Parallèlement, l'environnement de travail se développe comme dans les systèmes de programmation classiques. On y retrouve même la tendance à surinvestir l'amélioration de l'interaction au détriment de la recherche de solutions définitives aux problèmes de fond, attitude encouragée par le fait

4. Le représentant le plus utilisé de cette famille est aujourd'hui NROFF, alias TROFF.

5. Point n'est besoin de donner les références bibliographiques qui s'imposent aux lecteurs des *Cahiers GUTenberg*.

6. On peut citer SGML, que les lecteurs des *Cahiers GUTenberg* connaissent depuis le numéro 12 de ces cahiers, et LOUT de J. KINGSTON de l'université de SIDNEY.

qu'une bonne interaction dissimule mieux une mauvaise conception, qu'une bonne conception ne dissimule une mauvaise interaction.

3.2. Bilan

Au terme de ce parcours, c'est l'*abstraction des fonctions* qui reste la grande absente de cette répétition de l'histoire des langages de programmation. La problématique du trait d'union en TEX en fournit l'illustration : toutes les variables intervenant dans son apparition sont accessibles, mais il n'existe pas de *fonction* en interdisant tout simplement l'usage. Cette déconnexion impossible d'un utilitaire qui n'est adapté qu'aux cas prévus par son auteur entraîne parfois des gaspillages plus énormes que ceux qu'il est censé éviter, problématique classique des langages de programmation évolués. Concevoir une informatisation du traitement de texte fondée sur une approche fonctionnelle est le but de cet article.

Cependant, la remise à plat qui va suivre ne devrait pas être perçue comme un complet ostracisme envers les outils qui viennent d'être décrits. Car c'est une autre leçon de l'histoire de l'informatique qu'il faut prendre en considération ici. À leur début, les nouveaux langages de programmation évolués ont voulu produire du code natif pour les différentes machines visées. En ne voulant pas s'appuyer sur le savoir accumulé par les langages impératifs classiques, ils se sont condamnés à une opérationnalité tardive et médiocre[7]. Aujourd'hui, les langages de cette génération se sont peu à peu résignés à s'appuyer sur les compilateurs des langages impératifs les plus répandus.

De même ici, il faut admettre que les traitements de textes classiques ont acquis une grande maîtrise de la gestion des polices de caractères, parce qu'ils ont dû affronter des matériels différents, essentiellement l'imprimante laser et l'écran graphique, chacun avec leur langage typographique propre. En revanche, c'est parce qu'ils ont voulu imposer leur algorithme de justification, et plus généralement de mise en page, que ces outils révèlent un manque de maniabilité. Sur ce point, et sur ce point seulement, leur exemple ne peut être suivi, et il faut chercher des points d'appuis dans l'imprimerie traditionnelle.

7. L'exemple le plus patent est certainement le langage ADA, dont on a finalement renoncé à écrire des compilateurs autonomes, malgré des subventions sans commune mesure avec les autres langages de programmation.

4. Les manuels de mise en page

4.1. Les différents concepts

La reproduction de l'écriture a subi plusieurs révolutions[8] : plume, plomb composé à la main, linotype puis photocomposeuse, pour ne citer que les étapes les plus marquantes. Ces remises en question lui permettent aujourd'hui d'énoncer certains invariants. Ainsi, le découpage de la typographie en *composition* et *mise en pages* relève d'une abstraction dont on regrette qu'elle n'ait pas été mise plus à profit par les informaticiens[9]. Toutefois, du point de vue de l'informaticien, les manuels de mise en pages mêlent encore différents niveaux de problématique[10].

D'une part, y sont analysés les matériaux disponibles : les différentes polices de caractères, les formats des papiers, les méthodes de reproduction et bien d'autres, avec toutes les indications nécessaires au calcul des devis. C'est encore une pensée *impérative* qui est à l'œuvre ici, les outils d'imprimerie, voire les hommes qui les maîtrisent, s'assimilant aux instructions de langages de programmation, tandis que le prix de revient s'identifie au temps d'exécution d'une instruction.

D'autre part, interviennent les concepts les plus abstraits dans l'opposition entre *lisibilité* et *visibilité*, ou entre *jaillissement*, *objectivité* et *intégration*. On est alors pleinement dans une pensée *déclarative*.

Mais il est rare de trouver une pensée *applicative*, ou *fonctionnelle* si l'on préfère, c'est-à-dire la description de *fonctions* indépendantes de leur support, dont l'enchaînement donne le document final. En effet, la *chaîne graphique* est plus souvent décrite par le nom du document à chaque étape que par l'opération qui le produit, rarement nommée.

Le flou et l'incommodité du vocabulaire sont toujours significatifs d'un concept mal défini. L'accord en nombre dans la locution *mise en page* est

8. Cette histoire est merveilleusement résumée par Ladislas Mandel, dans *L'écriture typographique : vers une prise de conscience*, publié par *Communications et Langages*, numéro 77, 3^{e} trimestre 1988.

9. Elle est pourtant nécessaire pour résoudre rigoureusement le problème classique du `pretty-print` ; voir Emmanuel Saint-James, *La programmation applicative*, chapitre XX, éditions Hermès, 1993.

10. Afin de retirer aux critiques qui vont suivre leur aspect de distribution de mauvais points à des ouvrages que tout utilisateur de traitements de textes devrait longuement méditer, voici en vrac la liste des textes consultés : Pierre Duplan & Roger Jauneau, *Maquette et mise en page*, éditions du moniteur, 1992 ; Roland Fiszel (sous la direction de) *Lexique des règles typographiques*, Imprimerie nationale, 1990 ; Alan Marshall, *La composition typographique*, numéro 8 des *Cahiers GUTenberg*, Mars 1991 ; Massin, *La mise en pages*, éditions Hoëbocke 1991 ; François Richaudeau, *Manuel de typographie et de mise en page*, éditions Retz, 1989.

exemplaire à cet égard : un maquettiste ouvre son ouvrage en expliquant combien déterminante a été sa prise de conscience qu'il s'occupait de *mise en pages* et non de *mise en page* ; un autre auteur affirme que la *mise en page* est le travail de conception du maquettiste, tandis que la *mise en pages* est sa matérialisation à l'imprimerie ; chez d'autres enfin, les deux graphies coexistent indifféremment.

Par ailleurs, on relève un mélange de niveaux conceptuels dans les tentatives de théorisation. Une liste intitulée *16 concepts à la base d'une théorie typographique*, qui ne sont en fait qu'au nombre de 15 et même 13 puisque 2 sont des synonymes, mêle le *bloc typographique*, matériau impératif, avec la *microlisibilité*, concept déclaratif défini comme *lisibilité du bloc typographique*.

4.2. Coupure et césure

À cette confusion dans les niveaux conceptuels, vient s'ajouter l'indiscernement des aspects statique et dynamique d'un phénomène. La problématique de la *césure* ou *coupure* de mots n'est jamais exposée en distinguant deux problématiques :

- la place d'un trait d'union dans un mot, avec éventuellement un ordre de préférence quand plusieurs endroits sont licites ; cette donnée indépendante du contexte est *statique* ;
- la réduction d'espacement entre les mots, opération fortement dépendante du contexte, puisqu'elle dépend non seulement de la ligne courante mais aussi de la page, de par l'interdiction de couper en page impaire, et aussi du paragraphe, en vertu des règles interdisant 3 lignes consécutives terminées par un trait d'union, ainsi qu'une dernière ligne de paragraphe de longueur inférieure au renfoncement ; cette fonction est donc *dynamique*.

Dans cet article, on réservera le mot *coupure* pour l'aspect statique, et le mot *césure* pour l'aspect dynamique.

De même, on réservera la locution « *mot composé* » à un mot ayant déjà un trait d'union, comme « *tire-bouchon* ». Bien qu'un manuel n'hésite pas, sur une même page, à réutiliser cette locution pour désigner un mot dont l'étymologie comporte un préfixe imposant un trait d'union à l'intérieur

d'une syllabe, comme « *transaction* », on préférera ici introduire la locution « *mot agglutinant* » pour désigner ce deuxième cas, afin de lever toute ambiguïté.

4.3. Bilan

Ce relevé d'incohérences des manuels n'est évidemment pas un jugement de valeur, mais la liste des obstacles à une informatisation satisfaisante. Le processus d'informatisation commence toujours par dénoncer un manque de rationalité dans l'organisation du travail, manque qui s'explique historiquement, socialement voire esthétiquement. Le rôle de l'informaticien est de construire des outils observant les usages plus rapidement et plus sûrement, puis de proposer de nouveaux usages sans les imposer. C'est ce que se propose d'esquisser cette dernière section.

5. Architecture

Les concepts déclaratifs définis par les manuels de mise en pages ne pourraient déboucher sur des *traitements de texte déclaratifs* qu'aux conditions suivantes :

- formaliser des concepts comme *visibilité* ou *lisibilité* ;
- essayer tous les placements possibles de chaque lettre sur le support, jusqu'à satisfaire le critère adopté.

En ce qui concerne le premier point, il est permis de penser que tout ce qui relève de la sensation est, par définition, non formalisable. En ce qui concerne le second, la *complexité* sous-jacente rend peu attractif le temps de calcul prévisible, surtout si l'on ne trouve pas mieux comme critère de lisibilité qu'une distance minimale entre chaque couple de lettres. Peut-être l'avenir contredira-t-il ce scepticisme, mais de toute façon, l'histoire suggère de commencer par élaborer une vision applicative de la typographie.

Dans un premier temps, on énoncera les contraintes imposées par cette approche, et dans un deuxième, on donnera un cadre général de résolution.

5.1. Contraintes

5.1.1. La présentation de texte

Le vocabulaire même de la typographie indique à quel point la pensée applicative y est encore peu présente : la plupart des opérations sont désignées par des locutions, non par un substantif, d'où l'on pourrait dériver de surcroît verbe et adjectif. En particulier, *mise en pages* étant syntaxiquement lourd, sémantiquement controversé et techniquement mal adapté à l'évolution du matériel de reproduction, *disposition* semble mieux s'opposer à *composition*, plutôt que de remplacer les deux termes par *macrotypographie* et *microtypographie*, assez abscons. Mais le désir d'alléger un propos très technique ne s'en tirera pas toujours sans néologisme. Au moins, cette technique d'extension du vocabulaire obéit plus au souci d'univocité de l'informaticien, que le glissement de sens par métaphore, métonymie ou emprunt à une autre langue.

Pour commencer, la périphrase *outil de traitement de texte informatisé* est d'autant plus insupportable qu'elle est trop vague : non seulement les analyseurs lexicaux et syntaxiques répondent à cette description, mais c'est aussi le cas de la lexicométrie, de la traduction automatique, et même de certains procédés de la *littérature potentielle*[11]. On proposera ici « typographieur », outil de *présentation de texte*.

5.1.2. Indépendance envers le périphérique de sortie

La pensée fonctionnelle se veut indépendante de son support. En particulier, rien ne doit être supposé quant au périphérique de sortie, qui peut utiliser deux ou trois dimensions de l'espace, ainsi que la dimension du temps. On peut concevoir en effet :

- deux dimensions spatiales : c'est la feuille des imprimantes classiques ;
- trois dimensions spatiales : c'est le livre-objet pour bibliophile ou le livre animé pour enfant ou encore la lettre sculptée pour la signalisation, tous trois à la portée d'une machine-outil ;
- le temps : c'est la déclamation du texte par un synthétiseur vocal ;

11. Voir par exemple *La cimaise et la fraction*, résultat de la méthode S + 7 de l'Oulipo, *La littérature potentielle*, éditions Gallimard, 1973.

– le temps et deux dimensions spatiales : c'est l'écran de l'hypertexte ;

– le temps et trois dimensions spatiales : c'est le multimédia, que préfiguraient déjà les livres-disques ou les livres-cassettes.

À la limite, on peut considérer comme l'exploitation d'une dimension spatiale unique le *volumen* du Moyen Âge, réactualisé par les imprimantes à rouleau de papier. Plus généralement, c'est l'un des traits marquants de l'analyse fonctionnelle que de rapprocher, par delà l'espace et le temps, plusieurs problématiques dissociées par le matériel.

5.1.3. Indépendance envers le périphérique d'entrée

Symétriquement, le texte à typographier peut être :

– une matrice de points synthétisée par un scanner ;

– un flux de caractères émis dynamiquement à partir d'un clavier ;

– un flux de phonèmes émis dynamiquement à partir d'un microphone relié à un programme de reconnaissance de la parole ;

– une structure textuelle présente statiquement dans une mémoire.

Si la reconnaissance du braille, par son statisme, se ramène au cas du scanner, on espère que cette liste inclura un jour des périphériques reconnaissant le langage des signes utilisés par les sourds, voire des programmes effectuant la lecture labiale. Mais parce qu'elle s'abstrait des considérations matérielles, la pensée applicative vise à résoudre les problèmes en profondeur, autrement dit à trouver des solutions qui resteront valables quelles que soient les innovations technologiques.

5.1.4. Indépendance envers la langue

Les langues employées dans les textes constituent également un support qui, pour être immatériel, doit néanmoins être paramétrable. Leur diversité couvre les langages formels, comme la notation mathématique ou les langages de programmation, et les langues dites naturelles. Leur typographie doit respecter :

– le *sens* : vertical, horizontal, vers la gauche, vers la droite, boustrophédon ;

- le *format* : en drapeau, en pavé, en tableau, en calligramme[12] ;
- la *coupure* : syllabique, étymologique, algébrique ;
- la *césure* : trait d'union de longueur et de position fixe des langues justifiant dans le blanc, allongement des lettres contiguës des langues justifiant dans le noir ;
- la *ponctuation* : emploi, espacement autour du signe, suivi de majuscule.

5.2. Bilan

5.2.1. La mise en bit

La solution de facilité consisterait à entasser les outils de mise en page, de mise en écran, de mise en onde voire de mise en scène. Mais la synthèse serait plutôt une *mise en bit*, facilitant tout traitement informatisé d'un texte : *présentation*, mais aussi *indexation*, *traduction* et bien d'autres. Il vaut mieux chercher l'*intersection* des possibilités énoncées ci-dessus que leur *réunion*, mais cette intersection doit être la plus grande possible, afin de ne perdre aucune information provenant du périphérique d'entrée, ni aucune possibilité du périphérique de sortie. En particulier, que le texte puisse être écrit ou parlé conduit aux identifications suivantes :

- *composition* et *diction*, autrement dit choix des caractères et ton de la voix ;
- *disposition* et *élocution*, autrement dit mise en page et rythme du discours.

À l'articulation de la composition et de la disposition, se situe le trait d'union : en tant que caractère il relève de la composition, mais il n'apparaît que si la disposition l'exige. Par ailleurs, il faut remarquer qu'une coupure correcte présuppose une orthographe correcte. Puisqu'aujourd'hui la correction orthographique fondée sur des dictionnaires électroniques s'est banalisée, il est réaliste d'intégrer cette opération dans un typographieur,

12. Voir des exemples de calligrammes dans MASSIN, *La lettre et l'image*, éditions GALLIMARD, 1993 ; voir aussi le *Discours sur rien*, le *Discours sur quelque chose*, et les *Quarante-cinq minutes pour parleur* de JOHN CAGE dans son recueil *Silence* paru chez DENOËL en 1970, où l'auteur utilise les structures rythmiques aléatoires de ses compositions musicales pour énoncer ses conférences à leur sujet.

quitte à employer plusieurs dictionnaires pour un texte utilisant plusieurs langues, ainsi qu'une commande dérogatoire pour la citation textuelle de fautes. Enfin, comme plusieurs langues ont des mots en commun, mais des règles de coupure et de typographie propres, ces dictionnaires devraient étendre l'épellation des mots à l'indication de leurs coupures.

Au total, l'enchaînement *impression de l'évaluation de la lecture*, classique des langages applicatifs, suggère l'enchaînement *disposition de l'épellation de la composition* propre au typographieur. Autrement dit, le fameux :

```
(de toplevel ()
    (print (eval (read))) )
```

qui lit l'expression présente sur le flux d'entrée, l'évalue puis imprime son résultat sur le flux de sortie, devient :

```
(de typographieur ()
    (dispose (epelle (compose))) )
```

ou encore, dans un contexte de processus en pipe-line :

```
compose | epelle | dispose
```

Reste à préciser ces trois opérations. Mais auparavant, émettons deux remarques.

1. La séparation totale entre composition et disposition semble exclure la *lettrine*, qui appartient aux deux problématiques. On peut s'en défendre en se plaçant sous l'autorité de maquettistes la considérant comme un anachronisme, forme abâtardie de la lettre ornée des manuscrits. Plus démocratiquement, on peut faire confiance d'abord à la maîtrise du `patch` chez les `hackers`, ensuite à l'avenir, les langages applicatifs ayant fini par offrir des *processus* purement fonctionnels, avec la découverte tardive de la *réification de continuation.*

2. L'on dispose à présent d'un critère mathématique de qualité du traitement de texte : si les périphériques d'entrée et de sortie appartiennent à la même famille, par exemple scanner et imprimante, ou microphone & synthétiseur vocal, le résultat doit être égal à la donnée, autrement dit le typographieur doit être la fonction identité.

5.2.2. Composition

La composition part d'un flux d'entrée linéaire et retourne une structure composée de mots, chacun avec une police de caractères propre, et de signes de ponctuation ou de repérage. Les opérations successives sont les suivantes :

1. la *ponctuation* qui balise le texte avec les signes de ponctuation usuels ; cette opération inclut la gestion des *notes en bas de page ou en fin de texte*, beaucoup d'auteurs y plaçant du texte de même teneur que du texte entre parenthèses ou entre tirets, imprécision que l'on peut d'ailleurs déplorer ;
2. le « *poliçage* », néologisme indiquant le choix d'une police de caractères ;
3. la *référenciation*, néologisme indiquant la construction de la bibliographie ;
4. l'*indexation*, qui construit l'index ; les références sont bien sûr relatives à la structure construite, la notion de page n'intervenant pas en phase de composition ;
5. la *sommation*, ou construction du sommaire, avec des références semblables à celles de l'indexation.

Au total :

```
(de compose ()
    (sommation
        (referenciation (indexation (policage (ponctuation))))) )
```

5.2.3. Épellation

Cette opération consiste à remplacer chaque mot par sa suite de lettres séparées par des indications de coupures. Comme il existe des coupures prioritaires, les dictionnaires utilisés pourront repérer une coupure par son numéro de priorité, et réserver l'usage du trait d'union aux mots composés. Ainsi, il n'y aura pas confusion avec les mots agglutinants. Bien entendu, ces dictionnaires n'ont pas nécessairement besoin de contenir explicitement

l'intégralité des désinences possibles des mots d'une langue. Une compression des données, ou un dictionnaire réduit aux étymons mais couplé avec un algorithme de construction de mots peut s'insérer dans cette architecture.

5.2.4. Disposition

Cette opération se découpe comme suit.

1. Le *formatage*, qui provoque éventuellement des *césures* ; la *justification* est son mode le plus courant, mais on doit pouvoir lui substituer une répartition du texte en suivant le contour ou la surface d'un dessin, pour obtenir un *calligramme* ;
2. Le *titrage*, qui s'occupe du *péri-texte*, prévoyant en particulier une place pour le numéro de page ;
3. L'*empagement*, néologisme déjà employé par les professionnels, qui répartit sur le support les différents blocs de textes et le péri-texte ;
4. La *pagination*, qui résoud numériquement les références installées.

Au total :

```
(de dispose (t)
    (pagination (empagement (formatage t) (titrage t))) )
```

La fonction binaire que constitue l'empagement atteint les limitations des processus en pipe-lines qui, faute de primitives de *synchronisation*, n'acceptent que des fonctions unaires. Mais on pourra se plier aux pipe-lines en créant la fonction *empagement-titrage* qui prend en argument un texte après avoir mémorisé son formatage[13]. L'expérience dira si cette mémorisation est plus coûteuse que l'adoption d'une architecture avec client et serveur, qui semble superflue à première vue.

13. Autrement dit, on considère une fonction binaire comme une fonction unaire retournant en résultat une fonction unaire ayant mémorisé l'argument initial. Cette *unarisation* est parfois nommée *curryfication*, du nom du logicien CURRY.

6. Conclusion

Le premier but d'une architecture fonctionnelle comme celle-ci est de permettre le remplacement d'un de ses composants sans perturber les autres. En particulier, elle devrait autoriser l'intégration immédiate de langues inusitées, de mises en pages innovantes, d'indexations sophistiquées, de référenciations hypertextuelles et bien d'autres.

Bien entendu, elle nécessite la définition du format des données à chaque étape. De ce format dépendent les performances finales. Il est probable qu'elles seront insatisfaisantes aux premiers essais, comme le furent les performances du premier système LISP. Mais il faut compter et sur les idées d'optimisation qui ne manqueront pas d'apparaître, et sur le succès de ces fonctionnalités nouvelles.