Cahiers GUTenberg

TYPOGRAPHIE DE PROGRAMMES ADA

P. Naudin, C. Quitté

Cahiers GUTenberg, nº 5 (1990), p. 30-40.

<http://cahiers.gutenberg.eu.org/fitem?id=CG_1990___5_30_0>

Typographie de programmes Ada

P. NAUDIN[†] et C. QUITTÉ[‡]

[†]*Laboratoire d'informatique, Université Bordeaux I, 351 cours de la Libération, 33405 TALENCE Cedex*
[‡]*Département de mathématiques, Université de Poitiers 40, avenue du recteur Pineau, 86022 POITIERS Cedex*

Abstract

In this article, we introduce a tool for typesetting programs written in a programming language with keywords. We propose a style derived from the one used by the WEB *system to typeset Pascal programs. The strength of the set of macros that really do the job lies in the use of internal mechanisms of TEX in order to recognize the keywords, and thus no complex automaton needs to be considered. This way, the process is of reasonable efficiency, in contrast with the horrible slowness of a classical method written with TEX.*

Keywords: `\toks`, `\csname...\endcsname`, `\ifx`, `\ifcat`.

Résumé

Le but du petit outil présenté ici est la typographie de programmes écrits dans un langage à mots-clés. Le style de typographie utilisé pour cela est fortement inspiré des résultats fournis par WEB sur des programmes Pascal. La particularité de l'ensemble des macros qui effectuent ce travail est le fait qu'elles utilisent les mécanismes internes de TEX pour effectuer la reconnaissance des mots-clés du langage, et ne nécessitent pas la définition complexe d'un automate. Ceci permet d'assurer une efficacité raisonnable à un procédé qui serait d'une lenteur effroyable avec une algorithmique classique développée en TEX.

Mots-clés : `\toks`, `\csname...\endcsname`, `\ifx`, `\ifcat`.

> ... You may not have to resort to any subterfuge at all, since TEX is able to do lots of things in a straightforward way.
> — Donald E. Knuth, The TEXbook (1983)

Dès qu'une personne croise la route de TEX, elle est prise d'une frénésie d'écriture, qui va croissant au fur et à mesure de la découverte de possibilités nouvelles de typographie. Nous n'avons pas échappé à ce syndrome, et avons décidé un beau jour de rédiger des cours d'introduction à l'algorithmique algébrique, utilisant Ada comme langage support, tant pour la spécification et l'expression des algorithmes que pour la réalisation des programmes exemples. Alors s'est posée la question de choisir un style de typographie pour les programmes Ada.

A l'expérience, on s'aperçoit que l'utilisation extensive de la fonte `typewriter` ou d'une autre police de caractères à espacement fixe détruit une partie de l'harmonie des pages construites par TEX pour un texte mathématique. Après une recherche rapide dans la littérature informatique, et plus précisément dans les œuvres de Knuth lui-même[4], on découvre son système de programmation documentée[1], WEB[3, 6], qui permet de rendre la lecture d'un programme et de sa documentation moins rébarbative qu'à l'habitude. Même si nous ne désirons pas utiliser WEB, la typographie qu'il applique à un programme Pascal correspond assez bien à ce que l'on attend d'un outil de composition de programmes : mots-clés en gras, identificateurs en italiques...

La figure 1 illustre ce que donne l'application de notre macro à une portion d'unité Ada[1], cette unité ayant préalablement été traitée par un formateur de programmes qui a fourni le texte source

[1] Traduction libre du terme "Literate programming" qui signifie textuellement "programmation érudite."

```
generic
  type Item is limited private ;
  type Index is (<>) ;
  type Items is array (Index...
  with function "=" (Left : Ite...
package Pattern_Match_Knuth_Morr...
  function Location_Of (The_Patt...
  Pattern_Not_Found : exception ;
end Pattern_Match_Knuth_Morris_Pratt ;
```

Figure 1: Exemple de programme composé avec la macro

```
GENERIC
  TYPE Item  IS  LIMITED PRIVATE ;
  TYPE Index  IS  (<>) ;
  TYPE Items  IS  ARRAY (Index...
  WITH FUNCTION  "=" (Left : Ite...
PACKAGE Pattern_Match_Knuth_Morr...
  FUNCTION Location_Of (The_Patt...
  Pattern_Not_Found : EXCEPTION ;
END Pattern_Match_Knuth_Morris_Pratt ;
```

Figure 2: Texte source fourni par un formateur

visible sur la figure 2.

Le style étant choisi, il reste à déterminer comment le respecter. La première solution qui vient à l'esprit repose sur une transformation manuelle du programme : à partir de définitions construites sur le motif général

`\def \<mot-clé>{{\bf <mot-clé>}}`

il est possible d'écrire un pseudo-programme contenant toutes les directives de typographie nécessaires à sa composition correcte.

Avec cette solution apparaît immédiatement le problème de toutes les solutions analogues : il faut répercuter, à la main, toute modification du texte du programme sur la version prête pour la composition. Sans assistance, personne n'est capable de s'imposer cette discipline surtout si les modifications sont aussi nombreuses et fréquentes que les différentes versions d'un programme en cours de mise au point.

On peut se demander pourquoi nous n'avons pas utilisé un système comme WEB, applicable à des programmes Ada (il en existe). La raison essentielle de ce choix tient au fait que WEB a été conçu pour décrire d'une manière structurée un programme et sa documentation ; en conséquence, il impose une architecture de document totalement incompatible avec un texte dans lequel les programmes ne sont que des "accessoires" d'illustration, et non plus la matière essentielle du discours. De plus, cette solution WEB présente également les inconvénients des solutions à base de préprocesseurs : si l'on ne s'interdit pas toute modification de la sortie du préprocesseur, on tend très rapidement à avoir un programme qui ne concorde pas avec sa version typographiée.

Il ne reste évidemment que la solution de la macro TEX qui permettra d'éviter toute duplication de l'information ; cette macro doit lire le texte du programme, reconnaître les mots-clés qui y figurent, et en faire la typographie sélective convenue au départ.

Bien entendu, la commodité d'emploi d'une telle solution ne doit pas se payer

au prix d'une programmation TEX complexe, et donc probablement très lente : le compromis doit pencher très nettement en faveur de la commodité d'emploi. Heureusement le monde, et TEX en particulier, est bien fait : il est possible d'agir directement sur l'analyseur lexical de TEX, et de lui demander de reconnaître des mots de commandes particuliers ; c'est d'ailleurs l'un des mécanismes fondamentaux de TEX. L'analyseur lexical de TEX est la partie de TEX chargée de découper le flot de caractères d'entrée en lexèmes (tokens). Comme dans tout langage textuel, on peut agir sur cet analyseur, notamment grâce aux catégories associées aux caractères, et également par la construction de nouveaux noms de commande. Il ne reste alors plus qu'à encoder les mots-clés dans des mots de commande particuliers ; ainsi, on ne programme pas un automate d'analyse, on se borne à énumérer les mots-clés à reconnaître.

Pour parachever la typographie d'un programme, il faut de plus respecter sa structure en lignes, son indentation, et parfois numéroter les lignes du programme pour pouvoir y faire des références. Ces différents points, archiclassiques pour un utilisateur averti de TEX, s'avèrent plus complexes qu'à l'habitude dans ce contexte très spécial, la non-primalité de `\leavevmode` n'étant pas neutre en cette affaire.

On peut élargir le champ d'application des méthodes utilisées, et appliquer ce principe de reconnaissance de mots pour composer des textes-puzzles. On trouve sur la figure 3 un tel puzzle, dans lequel certains mots ont été effacés.

Le but du jeu est, bien sûr, de deviner quels sont les mots cachés afin de reconstituer le texte original. Le même principe peut être appliqué de manière complémentaire au même texte, en ne faisant apparaître que les mots fantômes, à la place exacte qu'ils auraient occupée si le texte avait été typographié normalement (figure 4). De cette façon les parties positive et négative du texte s'ajustent exactement pour composer le texte original.

.... Puis-je ajouter cependant que j'ai beaucoup d'affection pour une grande partie de la qu'il ecrivit adolescent, de sorte qu'il me paraitrait plus juste de dire que est devenu un " mediocre" et bien sur, si je dis qu'il est mort trop vieux, cela ne doit pas etre compris comme une reaction peu charitable a son deces premature ; cela ne consiste qu'a rejeter toutes les speculations du genre de "Songez a ce qu'il aurait pu faire s'il avait vecu soixante-dix ans".

Figure 3: Le texte-puzzle

Ces deux parties de texte ont été obtenues grâce à l'utilisation de deux macros dont les appels suivaient le modèle :

```
\phantoms {Mozart} {musique}
          {compositeur}

....~Puis-je ajouter cependant que j'ai
beaucoup d'affection pour une grande
........
avait vecu soixante-dix ans''.
\endoftext
```

La présence d'accents dans cette traduction d'un commentaire de Glenn GOULD[2] n'est possible que si l'on utilise une version de TEX supportant un jeu de caractères étendu, et les fontes correspondantes. La version 3 de

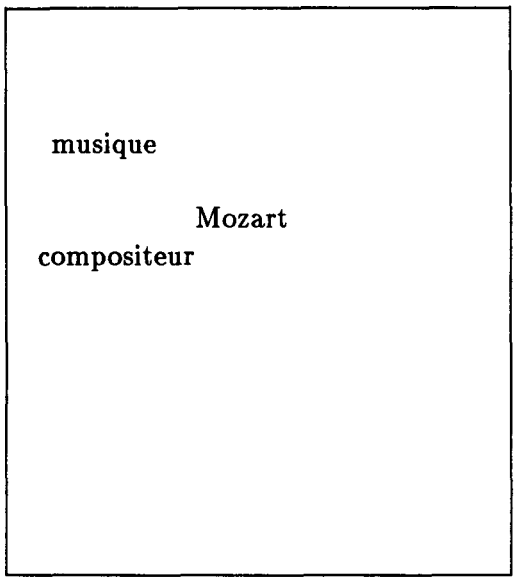

Figure 4: Les mots cachés

TEX devrait permettre cela, et certaines versions 2 évoluées autorisent l'utilisation de caractères à 8 bits, à qui on peut assigner la catégorie 13, ce qui résoud le problème des fontes. Dans ce dernier cas, il faut se restreindre à des mots fantômes ne comportant aucun accent.

1. Reconnaissance des mots

Les deux problèmes évoqués dans l'introduction peuvent être abordés de la même façon et leur solution se décompose en deux parties distinctes : l'extraction des mots d'un texte, et la reconnaissance des mots-clés. Nous allons commencer par exposer la méthode utilisée pour reconnaître les mots-clés, car c'est certainement la partie la plus "belle" de la macro, la plus simple aussi.

Dans le discours qui suit, `TheWord` désignera l'un des mots-clés que l'on veut identifier. La façon la plus simple de coder ce mot dans un mot de commande TEX, est d'utiliser la construction primitive `\csname...\endcsname`. Sans paraphraser le TEXbook outre mesure, on peut quand même rappeler que toute occurrence de `\csname TheWord\endcsname` est équivalente à une apparition du mot de commande `\TheWord`. TEX sait manipuler efficacement ces mots de commandes, par conséquent, ce sont ces mots de commandes qu'il faut faire reconnaître à TEX. Ces mots de commandes sont transformés par TEX en lexèmes, et l'une des méthodes utilisables pour reconnaître des lexèmes consiste à utiliser judicieusement le test `\ifx`. Plus précisément, si `\TheWord` est une macro, alors `\ifx \TheWord \cs` est équivalent à `\iftrue` si et seulement si `\TheWord` et `\cs` ont même expansion au premier niveau (TEXbook p. 210). On peut donc résoudre le problème de la reconnaissance du mot de commande `\TheWord` en définissant tout d'abord :

```
\def \keyword {\keyword }
\def \TheWord {\keyword }
```

Dans ce cas, le test `\ifx \TheWord \keyword` est équivalent à `\iftrue`. Puis, on peut reconnaître le *mot* `TheWord` en faisant le test

```
\expandafter \ifx
    \csname TheWord\endcsname
    \keyword
```

Si l'on imagine que le mot à tester est l'expansion de la macro `\WordToCheck`, le test

```
\expandafter \ifx
    \csname \WordToCheck \endcsname
    \keyword
```

donnera `\iftrue` si cette expansion est `TheWord` (ou `keyword`), et `\iffalse` sinon.

On voit immédiatement que la définition de l'ensemble des mots à reconnaître,

extrêmement facile, n'est, pour les mots-clés d'Ada, que la suite de définitions :

```
\def \ABORT      {\keyword }
\def \ABS        {\keyword }
    ... cinquante-neuf
    definitions analogues...
\def \WITH       {\keyword }
\def \XOR        {\keyword }
```

Telle qu'elle est définie, cette table ne permettra, en utilisant les principes développés dans la suite, que la reconnaissance des mots-clés écrits en majuscules. Si l'on désire, sans ajouter d'autres entrées à cette table, reconnaître les mots-clés d'Ada quelle que soit leur écriture, il faut, avant de faire le test, transformer la macro **\WordToCheck**, de sorte que son expansion représente la même suite de lettres, mais majuscules. Cette transformation n'est pas si simple qu'il y paraît de prime abord, et le lecteur intéressé se reportera à l'exercice 20.19[5].

De plus, un concours de circonstances malheureuses fait que **else** est un mot-clé Ada, et si l'on désire reconnaître des mots-clés donnés en minuscules sans utiliser la méthode donnée dans l'exercice, la redéfinition de la primitive **\else** risque de provoquer les pires catastrophes. Cependant, on peut facilement s'affranchir de ce problème en modifiant légèrement les définitions.

Il existe encore une dernière raison justifiant le fait que notre macro ne reconnaît les mots-clés que s'il figurent en majuscules : en Ada, il peut arriver que les mots réservés soient utilisés comme identificateurs d'attributs prédéfinis, et dans ce cas leur typographie doit être celle des identificateurs standards. Pour toutes ces raisons, les mots-clés du programme devront être en majuscules, ce qui n'est pas une contrainte bien lourde, tout formateur de programmes Ada digne de ce nom étant capable de faire le travail de majusculisation.

2. Extraction des mots d'un texte

La seconde étape importante dans le traitement que l'on veut faire subir aux textes, est le découpage de ces textes en mots, qui pourront être reconnus grâce aux principes précédents. Pour effectuer ce découpage, la technique utilisée est la même que dans les langages de programmation classiques :

- au niveau le plus bas, on lit un flot de caractères
- on sélectionne, parmi ces suites de caractères les lettres consécutives, que l'on regroupe pour former des mots.

Voici le prototype de macro TEX qui permet de traiter, caractère par caractère, un paragraphe :

```
\def \paragraphhandler #1%
  {\ifx #1\par
    \let \next = \relax
  \else
    #1
    \let \next = \paragraphhandler
  \fi \next }
```

Cette macro appliquée à un paragraphe effectue la typographie de tous les tokens non blancs rencontrés lors de la lecture du paragraphe, chaque token étant suivi d'un espace.

Comme on l'a vu à la section précédente, il faut stocker, dans le corps d'une macro, le mot dont on veut savoir s'il figure dans la table des mots-clés. Au lieu de placer les mots dans le texte de substitution d'une macro, on peut les loger dans un registre **\toks**. Deux questions subsistent encore : comment distinguer

les lettres des autres caractères, et comment accumuler, dans un `\toks` les différentes lettres d'un mot ? La réponse à la première question est contenue dans la question ; TEX possède en effet une primitive permettant de déterminer la catégorie d'un token : les lettres sont de catégorie 11. La réponse à la seconde question n'est pas beaucoup plus difficile : il suffit d'appliquer à un `\toks` une opération analogue à celle que l'on applique aux variables entières d'un programme classique lorsqu'on les incrémente. Si `\word` est un registre `\toks` valant `TheWor`, après exécution de la directive `\word=\expandafter{\the\word d}`, ce même registre aura pour valeur `TheWord`, et pourra donc être reconnu grâce à la méthode exposée dans la section précédente.

Il est maintenant possible d'écrire le squelette d'une macro permettant de reconnaître, dans un texte, le mot `TheWord` :

```
\def \keyword {\keyword }
\def \TheWord {\keyword }
\def \wordmanager #1%
  {\ifcat \noexpand #1a%
    \word=\expandafter{\the\word#1}%
  \else
    \expandafter \ifx
        \csname\the\word\endcsname
        \keyword
      % 1. Le mot lu est TheWord
    \else
      % 2. Le mot lu n'est pas
      %    TheWord
    \fi
  \fi \wordmanager }
```

Dans chacun des cas, 1 ou 2, il faut éventuellement traiter le mot contenu dans le registre `\word`, le token `#1` qui a délimité ce mot, et impérativement réinitialiser le registre `\word`. Incidemment, dans le cas 1, on reconnaît aussi toute occurrence de `keyword`. Il faut noter également que rien n'est prévu dans ce squelette pour arrêter l'exécution de la macro.

3. Les textes-puzzles

Il est maintenant possible de revenir à l'implémentation de la seconde application discutée au début de cet article : le texte puzzle. Cette macro ne va pas être d'une utilisation totalement transparente à l'utilisateur afin de ne pas compliquer outre mesure sa réalisation. Plus précisément, dans le texte fourni par l'utilisateur, les caractères de fin de ligne seront strictement équivalents à des blancs (ce qui n'est pas le cas des caractères de catégorie 5), de plus, chaque blanc sera significatif (ce qui n'est pas le cas des caractères de catégorie 10). Toutes ces contraintes sont imposées afin de pallier au fait qu'une macro ne peut pas avoir comme argument un espace, dans le cas où le paramètre correspondant n'est pas délimité (cf. TEXbook p.204), sauf à mettre cet espace entre accolades, ce qui nécessite une intervention sur le texte original.

Une des méthodes utilisables pour rendre tout caractère blanc significatif est de lui assigner la catégorie 13, ce qui en fait un caractère actif, et de lui donner la signification de `\space` ; dans ce cas, la rencontre d'un espace peut se détecter, si besoin est, grâce au test `\ifx`<token>`\space`, qui ne provoque pas l'expansion du lexème <token>.

Il y a plusieurs solutions au problème des fins de lignes. L'une d'elles consiste à rendre actif le caractère de fin de ligne, et à lui donner la signification `\space`. Une autre solution consiste à définir `\endlinechar=32`, TEX remplace alors chaque fin de ligne physique par le caractère de code 32, qui est l'espace,

dont la catégorie a été auparavant fixée à 13.

On peut maintenant écrire les principales macro-définitions nécessaires à la confection des textes-puzzles donnés en exemple.

```
\newtoks \word
{\obeyspaces\global\let =\space}
{\catcode `\^^M = \active%
 \global \let ^^M = \space }

\def \keyword {\keyword }
\let \Mozart = \keyword
\let \musique = \keyword
\let \compositeur = \keyword

\def \textwithoutphantoms
  {\begingroup
  \let \specialword = \hideword
  \let \normalword = \showword
  \catcode `\^^M = \active
  \obeyspaces
  \hbadness = 10000
  \tolerance = 10000
  \everypar={\wordmanager}}

\def \endoftext
  {\par \let \next = \endgroup }

\def \wordmanager #1%
  {\let \next = \wordmanager
  \ifcat \noexpand #1a%
    \word=\expandafter{\the\word#1}%
  \else
    \expandafter \ifx
        \csname\the\word\endcsname
        \keyword
      \specialword {\the \word }%
    \else
      \normalword {\the \word }%
    \fi
    \word = {}%
    \ifx #1\space
      \space
    \else
      \ifx #1\endoftext
        \endoftext
      \else
        \normalword #1%
      \fi
    \fi
  \fi
  \next }
```

La macro `\textwithoutphantoms` définit tout d'abord les commandes `\specialword` et `\normalword` spécifiant le traitement à infliger aux mots selon qu'ils sont fantômes ou pas ; en changeant seulement ces deux commandes, on peut obtenir le puzzle ou les mots cachés. Après avoir rendu le blanc et le caractère de fin de ligne actifs, elle définit les paramètres à utiliser pour la construction des paragraphes du texte (ici des paramètres très laxistes afin d'éviter toute coupure de mot, et toute plainte de TEX). Enfin, elle ne doit pas appliquer directement la macro `\wordmanager` — ce qui donnerait des résultats bizarres si TEX n'était pas en mode horizontal à ce moment — mais définir simplement `\everypar`, qui provoquera cette application dès l'entrée en mode horizontal. Il faut noter que ce problème d'entrée en mode horizontal est loin d'être réglé ; il réapparaîtra de manière plus explicite lors de la présentation de la macro de typographie des programmes Ada, et sera résolu à ce moment.

Dans la macro `\wordmanager`, il est possible de supprimer le dernier test (celui qui permet de détecter la fin du texte), mais cela donne une solution encore plus embrouillée. Ici, lorsqu'on détecte cette fin de texte, par la présence du nom de macro `\endoftext`, on applique cette macro, qui termine le paragraphe avant de faire en sorte que le groupe ouvert par `\textwithoutphantoms` se termine ; ainsi le paragraphe est élaboré en tenant compte des paramètres définis par cette même macro à l'intérieur du groupe.

Les deux macros `\showword` et `\hideword`, qui décrivent le traitement que l'on fait subir aux mots selon qu'ils sont fantômes ou pas, peuvent être écrites très simplement si l'on dis-

pose de fontes visibles et invisibles se correspondant, comme dans SLiTEX. Nous ne disposions pas de telles fontes dans une taille raisonnable, aussi nous avons procédé différemment, en nous inspirant de la macro **\phantom** de plain TEX.

Le lecteur attentif aura remarqué que ces macros ne s'utilisent pas exactement comme nous l'avons décrit dans l'introduction, mais le travail restant à faire pour assurer cette interface est simple, et laissé en exercice.

4. La typographie des programmes

Nous voici enfin arrivés à l'application initiale de cette macro : la typographie des programmes Ada. Nous avons tous les éléments en main pour définir cette macro : lecture mot par mot d'un texte, reconnaissance des mots-clés, il ne reste plus qu'à ajouter les commandes habituelles qui imposent le respect de l'espacement et des lignes du programme. La partie lecture des mots s'écrit, comme précédemment

```
\long \def \wordmanager #1%
  {\ifcat \noexpand #1a%
    \word=\expandafter{\the\word#1}%
    \let \next = \wordmanager
  \else
    \expandafter \ifx
        \csname\the\word\endcsname
        \keyword
      \specialword {\the \word }%
    \else
      \normalword {\the \word }%
    \fi  \word = {}%
    \handleother #1%
  \fi \next }
```

où **\handleother** définit le traitement que l'on applique au lexème qui a provoqué l'arrêt de la lecture d'un mot. La macro utilisateur **\adaprogram** est alors définie de la manière suivante :

```
\outer \def \adaprogram #1\par
  {\par \begingroup
  \parindent = 0pt \parskip = 0pt
  \def \par {\leavevmode \endgraf }
  \everypar = {\wordmanager }
  \def \do##1{\catcode`##1=12 }
  \dospecials \obeylines \obeyspaces
  \it \input #1 \endgroup }
```

étant entendu que l'on a rendu le caractère blanc actif équivalent à \␣, afin que les blancs de début de ligne ne soient pas éliminés. Pour des raisons analogues à celles données dans la section précédente, la macro **\wordmanager** est appliquée uniquement lorsque TEX entre en mode horizontal ; en conséquence, toute fin de ligne doit provoquer un retour au mode vertical, et la macro **\handleother** doit être définie par :

```
\def \handleother #1%
  {\ifx #1\par
         \let\next=\relax \par
  \else  \let\next=\wordmanager
         {\it #1}\fi }
```

Les définitions semblent complètes, et si l'on applique la macro **\adaprogram** au programme exemple donné dans l'introduction, tout se passera effectivement bien. Mais le texte de ce programme est un peu particulier : il ne contient pas de ligne vide. Que se passerait-il sinon ? Lorsque l'on applique la macro à un texte identique au précédent, mais comportant une seconde ligne vide, on obtient un message d'erreur semblable au suivant :

```
! Missing number, treated as zero.
<to be read again>
                   }
\wordmanager ...\handleother #1
                         \fi \next
\leavevmode ->\unhbox
                       \voidb@x
^^M->\leavevmode
                  \endgraf
```

Que s'est-il passé ? L'explication n'est pas simple ! Le traitement de la première

ligne du programme s'est passé normalement, et en fin de ligne, TEX est repassé dans le mode vertical. Puis a commencé la lecture de la seconde ligne, qui est vide ; TEX rencontre un caractère de fin de ligne en mode vertical, et l'interprète comme le lexème **\par** dont l'expansion est **\leavevmode \endgraf**. Mais **\leavevmode** n'est pas une primitive de TEX, c'est une macro dont on voit l'expansion dans le message d'erreur ci-dessus, et c'est la primitive **\unhbox** qui provoque l'entrée en mode horizontal. Et c'est là que la catastrophe se produit : TEX vient de lire **\unhbox** et attend donc le numéro d'un registre **\box**, et que lui fournit-on ? La liste de lexèmes **\everypar**, qui demande l'application de la macro **\wordmanager**... Et cette macro tarde à fournir un numéro de registre à **\unhbox** ! Nous nous tournons immédiatement vers la bible et l'exercice 13.1 nous fournit la réponse. La solution choisie dans plain TEX pour **\leavevmode** l'a été pour des raisons d'efficacité, mais il existe au moins une autre solution qui n'a pas l'inconvénient précédent. On modifie donc, dans la macro **\adaprogram**, la redéfinition de **\par** :

```
\def \par {\ \unskip \endgraf }
```

Avec cette modification, on élimine le problème précédent, mais il en apparaît un autre. En effet, dans les mêmes conditions, après être entré en mode horizontal sur cette seconde ligne, TEX applique **\wordmanager** qui va rencontrer **\endgraf**, mais ne va pas traiter cette primitive comme elle traite le lexème **\par** : **\endgraf** subit donc le sort des lexèmes qui ne sont pas des lettres, et est placé tel quel dans la liste que construit TEX. Par conséquent, TEX repasse en mode vertical et la macro **\wordmanager** va continuer à opérer, en mode vertical maintenant, ce qui va provoquer des phénomènes bizarres dès qu'elle va placer du matériel horizontal dans la page. Il faut donc modifier le traitement effectué par **\handleother**, de sorte que **\endgraf** ait le même traitement que **\par** :

```
\def \handleother #1%
  {\ifx #1\par
    \let\next=\relax \par
  \else \ifx #1\endgraf
    \let\next=\relax \endgraf
  \else   \let\next=\wordmanager
    {\it #1}\fi \fi }
```

Tous les problèmes liés au fonctionnement de TEX sont maintenant réglés, il reste à s'affranchir des problèmes liés à la fonte italique utilisée. En effet, cette police de caractères contient des caractères pour le moins bizarres en lieu et place de caractères tout à fait ordinaires dans un texte de programme (par exemple — à la place de |). De plus, certains caractères italiques ont une esthétique incompatible avec le listing d'un programme (comme par exemple les guillemets ″). L'idéal serait de sélectionner des caractères dans différentes polices de façon à se confectionner une police "listing italique", mais dans un premier temps, on peut se satisfaire d'une solution à base de caractères actifs, ce qui n'accélère évidemment pas le traitement effectué par TEX.

On peut aussi apporter des perfectionnements à cette macro :

- La possibilité de numéroter les lignes, ce qui est très utile lorsqu'on doit faire des références au programme
- Un contrôle fin des coupures de page : en effet, il est bon que ces coupures se produisent entre des

```
function Location_Of(The_Pattern  : Items;
                     In_The_Items : Items)
    return Index;
```

Figure 5: Exemple d'indentation de programme

structures logiques du programme, et non pas n'importe où. Généralement, on distingue ces structures, dans le texte source, en plaçant judicieusement des lignes vides dans le programme ; il est facile de faire en sorte que TEX ne coupe des pages que lorsqu'il rencontre un certain nombre de lignes vides consécutives.

Enfin, cette macro, qui respecte une certaine indentation du programme, ne permet pas de maintenir des alignements autres que ceux concernant les débuts de lignes, comme dans l'exemple de la figure 5.

Or, bien souvent, de tels alignements sur des repères situés au milieu des lignes rendent la lecture des programmes plus facile. Seul un véritable préprocesseur, comportant un analyseur syntaxique complet, pourra faire un tel traitement.

Malgré tous ces efforts, on doit encore se demander si cette solution est sufisamment simple.

> ... Always remember, however, that there's usually a simpler and better way to do something than the first way that pops into your head...
>
> — Donald E. Knuth, The TEXbook (1983)

Références bibliographiques

[1] Grady Booch. *Software components with Ada.* Benjamin/Cummings, 1987.

[2] Glenn Gould. *Le dernier puritain, écrits I réunis, traduits et présentés par Bruno Monsaingeon.* Librairie Arthème Fayard, 1983.

[3] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, October 1984.

[4] Donald E. Knuth. *TEX: The Program.* Addison-Wesley, 1986.

[5] Donald E. Knuth. *The TEXbook.* Addison-Wesley, 1986.

[6] Donald E. Knuth. *The WEB system of structured documentation.* Stanford Computer Science Report 980, Stanford University, California, September 1983.

A. Textes-puzzles

Les macros développées dans la section 3 ne correspondent pas vraiment au modèle d'appel donné dans l'introduction. Voici une solution à l'exercice donné à la fin de la section 3 :

```
\newtoks\word \def\keyword{\keyword}
{\obeyspaces\global\let =\space}
{\catcode `\^^M = \active%
  \global \let ^^M = \space }
\long \def \phantoms #1%
  {\ifx #1\par  \let\next=\text
  \else  \let\next=\phantoms
    \expandafter
    \let\csname#1\endcsname = \keyword
  \fi \next }
\def \text
  {\begingroup
  \let \specialword = \hideword
  \let \normalword = \showword
  \catcode `\^^M = \active
  \obeyspaces
  \hbadness = 10000 \tolerance = 10000
  \everypar={\hfill\wordmanager}}
\def \endoftext
  {\par \let \next = \endgroup }
\def \wordmanager #1%
```

```
  {\let \next = \wordmanager
  \ifcat \noexpand #1a%
    \word=\expandafter{\the\word#1}%
  \else
    \expandafter \ifx
        \csname\the\word\endcsname
        \keyword
      \specialword {\the \word }%
    \else \normalword {\the \word }%
    \fi  \word = {}%
    \ifx #1\space  \space
    \else  \ifx #1\endoftext
        \endoftext
    \else  \normalword #1%
    \fi \fi \fi \next }
\def \showword #1{\hbox {#1}}
\def \hideword #1%
  {\setbox0=\hbox{#1}\setbox1=\null
  \wd1=\wd0\ht1=\ht0\dp1=\dp0 \box1 }
```

B. Composition des programmes Ada

Voici le texte complet des macros de composition de programmes utilisées pour l'exemple donné en introduction ; pour l'essentiel, il rassemble les macros développées dans la dernière section, en y ajoutant des définitions de caractères actifs.

```
\newtoks\word \def\keyword{\keyword}
{\catcode`\ =\active\global\let =\ }
{\catcode`\_=\active\global\let_=\_}
\def\doublequote{{\tt"}}
{\catcode `\" = \active
  \global \let " = \doublequote }

\long \def \wordmanager #1%
  {\ifcat \noexpand #1a%
    \word=\expandafter{\the\word#1}%
    \let \next = \wordmanager
  \else \expandafter\ifx
        \csname\the\word\endcsname
        \keyword
      \specialword {\the \word }%
  \else \normalword {\the\word }\fi
  \word = {}\handleother #1\fi \next }
\def \specialword #1%
  {{\bf\lowercase\expandafter{#1}}}
\def \normalword #1{{\it #1}}
\def \handleother #1%
  {\ifx #1\par  \let\next=\relax
  \else \ifx #1\endgraf
    \let\next=\relax
  \else   \let\next=\wordmanager
  \fi \fi {\it #1}}
\outer \def \adaprogram #1\par
  {\par \begingroup
  \parindent = 0pt \parskip = 0pt
  \def \par {\ \unskip \endgraf }
  \everypar = {\wordmanager }
  \def \do##1{\catcode`##1=12}
  \dospecials \obeylines \obeyspaces
  \catcode`\_ = \active
  \catcode`\" = \active
  \catcode `\> = \active
  \catcode `\< = \active
  \it \input #1 \endgroup }

\def \ABORT     {\keyword }
\def \ABS       {\keyword }
\def \ACCEPT ...
```

Ce fichier de macros se termine par l'énumération des 63 mots-clés du langage Ada.