Paradis: un systeme de paragraphage dirige par la syntaxe

Ph. Deschamp, Pierre Boullier

INRIA
CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél. : (1) 39 63 55 11

Rapports de Recherche

N° 455

PARADIS: UN SYSTÈME DE PARAGRAPHAGE DIRIGÉ PAR LA SYNTAXE

Philippe DESCHAMP
Pierre BOULLIER

Novembre 1985

Paradis:
Un Système de Paragraphage
Dirigé par la Syntaxe

Philippe Deschamp et Pierre Boullier
INRIA - Rocquencourt

Résumé: Nous présentons dans ce rapport une approche à la spécification de paragrapheurs. Elle permet d'obtenir facilement un paragrapheur pour tout langage décrit par une grammaire hors-contexte. La spécification s'effectue en utilisant la forme textuelle de cette grammaire. Le paragrapheur correspondant décompile un arbre syntaxique élagué, en exploitant du code engendré à partir de la spécification. Un investissement négligeable permet d'obtenir un paragrapheur à partir de la grammaire. Cette approche est à la base de Paradis, un système de paragraphage dirigé par la syntaxe et indépendant du langage. Paradis a permis de construire des paragrapheurs de bonne qualité pour (un sous-ensemble de) PL/1, (un sur-ensemble de) C, Pascal et Ada (1).

Mots-Clés: paragrapheurs, paragraphage, paragraphage dirigé par la syntaxe, paragraphage indépendant du langage, environnements de programmation.

Abstract: We present a very simple, syntax-directed, approach to pretty-printing specification. It allows to easily obtain a pretty-printer for any language which can be described by a context-free grammar. Pretty-printing specifications for the language are derived from the indentation of the grammar. Programs can then be displayed according to these specifications. Each user can tailor the pretty-printer to his taste, by simply indenting differently some of the grammar rules. Pretty-printers are thus produced at a negligible cost. This approach has actually been implemented in a language-independent pretty-printing system, named Paradis. Satisfactory pretty-printers for (a subset of) PL/1, (a superset of) C, Pascal and Ada have been specified.

Keywords: pretty-printers, pretty-printing, syntax-directed pretty-printing, language-independent pretty-printing, programming environments.

---

(1) Ada est une marque déposée du Gouvernement des Etats-Unis d'Amérique, Ada Joint Program Office.

Paradis:
Un Système de Paragraphage
Dirigé par la Syntaxe

Philippe Deschamp et Pierre Boullier
INRIA - Rocquencourt

1 - Introduction

Plusieurs types de programmes permettent d'obtenir des textes agréables à lire, faciles à modifier et dont la présentation est standardisée. On peut grosso-modo en distinguer trois classes principales, qui correspondent à des vues différentes sur le texte à manipuler et donc sur les traitements que l'on peut lui faire subir.

Une première classe, citée ici pour mémoire, est constituée par les programmes de mise en pages de textes. Ces processeurs ne sont en général pas adaptés aux langages de programmation, pour la simple raison qu'ils nécessitent des informations sur le résultat désiré, ces informations étant codées dans le texte accepté en entrée (exemples: directives de saut de lignes ou de page, définitions de blocs, utilisation de macro-procédures pour la création de titres ou l'imbrication de paragraphes).

Les deux autres classes correspondent plus à notre propos: nous les désignerons sous les noms d'**indenteurs** et de **paragrapheurs**. Un indenteur produit, à partir d'un texte en entrée, un texte comportant le même nombre de lignes (la plupart du temps), chaque ligne étant séparée de la marge gauche par un espace vide (**indentation**) dont la taille permet de repérer visuellement les imbrications de notions. Un paragrapheur en revanche effectue une analyse plus poussée du texte, de façon à produire lui-même la séparation en lignes (pour permettre de mieux distinguer les diverses notions entrant dans la composition du texte), avant de produire les indentations de ces lignes.

Indenteurs et paragrapheurs sont bien adaptés à la manipulation de textes représentant des programmes. L'utilisation de l'un ou l'autre de ces types d'outils n'est toutefois pas une simple affaire de goût personnel, car leurs propriétés générales sont très différentes.

Les indenteurs -- tout du moins ceux qui ont été utilisés par les auteurs -- sont des programmes non paramétrés, sur lesquels l'utilisateur n'a que très peu de possibilités d'action; notamment si l'indentation produite n'est pas satisfaisante sur un texte particulier, son auteur n'a pour seule solution que de ne pas utiliser l'indenteur. Un autre inconvénient des indenteurs est que la structure de lignes du texte engendré est imposée par celle du texte entré; l'uniformité des textes manipulés, au niveau d'un projet par exemple, ne peut donc pas être garantie par la simple utilisation d'un indenteur. Celui-ci devra être doublé d'un règlement définissant exactement où dans son programme le programmeur devra insérer une fin de ligne! Cette déficience peut toutefois être considérée comme un avantage lorsqu'une présentation uniforme n'est pas nécessaire, car il est alors possible d'obtenir dans un même texte deux indentations différentes d'une même construction, chaque indentation explicitant un sens différent de cette construction (un exemple typique de ceci est l'utilisation de

tests en cascade dans les langages ne possédant pas d'instruction de choix). Un autre avantage, souvent décisif, est l'efficacité générale de ce type de programmes, "codés à la main" et effectuant très peu de vérifications sur la syntaxe du texte à indenter. En revanche, du point de vue de l'implanteur, un indenteur est très difficile à modifier pour l'adapter à un autre langage de programmation que celui pour lequel il a été conçu. L'écriture d'un indenteur pour un nouveau langage est donc en général une tâche qui est accomplie en repartant à zéro.

Un paragrapheur en revanche utilise une forme interne du texte pré-analysé, de façon à prendre en compte la **structure** de ce texte (définie comme l'imbrication des notions qu'il utilise). Son utilisation est donc particulièrement intéressante dans un environnement de programmation, par exemple un éditeur syntaxique ou un compilateur (pour la sortie de messages d'erreurs). Cependant, lors d'une utilisation hors d'un tel environnement, ceci peut être considéré comme un désavantage, car il est alors nécessaire de construire cette forme intermédiaire; l'existence d'outils très performants d'analyse lexicale et syntaxique de programmes tempère cet inconvénient, qui est en outre compensé par la détection éventuelle d'erreurs dès cette construction. Un autre avantage de ce fonctionnement par **décompilation** de forme interne est que la standardisation des formats de programmes au sein d'un projet est ainsi aisément assurée, puisque le texte produit ne dépend pas de la structure de lignes du texte accepté en entrée. De plus, les paragrapheurs sont en général complètement paramétrés (dirigés par tables), ce qui d'une part donne la possibilité à un utilisateur de spécifier une indentation à son goût -- bien que les tables nécessaires soient la plupart du temps fastidieuses à écrire -- et d'autre part permet d'obtenir un paragrapheur pour un nouveau langage assez aisément.

Ce rapport décrit les concepts que nous avons dégagés pour la spécification de paragrapheurs dirigés par la syntaxe ainsi que l'implantation que nous avons réalisée (système **Paradis**). Nous aborderons brièvement les principaux problèmes rencontrés -- principalement la longueur bornée des lignes et le positionnement des commentaires --. Notre **méta-paragrapheur** est utilisé dans notre projet depuis 1981 et nous verrons quels enseignements on a pu en tirer quant aux améliorations possibles.

2 - Paragraphage

Une remarque préliminaire s'impose: la toute première particularité nécessaire à un langage pour lequel un paragrapheur est désiré est bien évidemment que les espaces et fins de ligne entre deux unités lexicales de tout programme y soient considérés comme des commentaires, n'influant donc pas sur la sémantique du programme. Ceci ne surprendra personne, mais il s'agit toutefois d'une limitation assez importante: des langages comme FORTRAN-IV ou BASIC ne peuvent que très difficilement être paragraphés.

L'idée nous est venue de nous intéresser au paragraphage de programmes à la suite de la remarque suivante: de même qu'il est bien plus agréable et efficace de travailler sur des programmes bien présentés, la compréhension de la grammaire d'un langage de programmation est facilitée si les parties droites de ses règles syntaxiques sont indentées de façon à refléter la présentation désirée pour les programmes écrits dans ce langage. Il est alors envisageable d'obtenir automatiquement l'indentation des programmes à partir de cette "indentation de la grammaire".

Nous avons donc décidé d'étudier la faisabilité d'un système de paragraphage dont l'entrée serait une grammaire du langage à paragrapher; cette grammaire, écrite dans un formalisme proche de la BNF, spécifie à la fois la syntaxe de ce langage et la décompilation désirée des arbres syntaxiques obtenus par analyse de programmes écrits dans ce langage.

Cette spécification devant être aussi aisée que possible pour l'utilisateur et complètement transparente pour toute personne désireuse de ne considérer que la partie syntaxique de la grammaire, nous avons dans un premier temps décidé de ne tenir compte pour le paragraphage que des espacements et retours à la ligne qui apparaissent dans les parties droites de règles (1).

Pour illustrer notre propos, nous présentons ci-dessous un extrait d'une grammaire de Pascal. Cette grammaire est utilisée tout au long de ce rapport à titre d'exemple. Elle a exactement la forme utilisée dans notre système: le non-terminal en partie gauche est séparé de la partie droite par le caractère d'égalité (=); les non-terminaux sont encadrés par des chevrons (< et >); les terminaux génériques -- c'est-à-dire les terminaux qui ne sont ni des mots-clés du langage ni des (suites de) caractères spéciaux -- sont précédés du caractère pourcent (%), les autres terminaux sont encadrés par des guillemets (") en cas de possibilité d'ambiguïté; toute règle est terminée par un point-virgule (;); les alternatives sont décrites par plusieurs règles faisant intervenir le même non-terminal en partie gauche.

Exemple: Déclaration simplifiée de procédure en Pascal.

```
<PROC-DECL>      = <PROC-HEADER>

                       <PROC-BODY>  ";" ;
<PROC-HEADER>    = procedure  %IDENT  <PARAM*> ";" ;
<PROC-BODY>      = <COMPOUND-STMT> ;
<COMPOUND-STMT>  = begin
                       <STMT-LIST>
                   end ;
```

Cette spécification permet, à partir du texte:

```
PROCEDURE SCAN ; BEGIN {...} END (* SCAN *) ;
```

d'obtenir le texte ainsi paragraphé (2):

```
procedure SCAN ;

   begin
      {...}
   end (* SCAN *) ;
```

---

(1) On pourra comparer cette approche avec celle décrite en [Rubin 83], où des triplets représentant des commandes de paragraphage sont insérés entre les différentes unités des parties droites des règles de la grammaire.

(2) Le passage en minuscules des mots-clés est obtenu par une option de paragraphage (voir au §4).

La partie de grammaire ci-dessus est pour un non-initié simplement plus lisible qu'une grammaire équivalente présentée sous forme linéaire. Nous verrons plus loin comment elle définit précisément le paragraphage donné en exemple. Il suffit pour l'instant que l'on voit bien que la forme textuelle des parties droites de cette grammaire détermine la forme du texte obtenu.

Un premier principe fondamental de notre approche peut être déduit de cet exemple: si une notion du langage est représentée par un non-terminal de la grammaire, l'imbrication de telles notions est directement décrite par les différentes productions de cette grammaire. Il est donc aisé de refléter ces imbrications en une disposition typographique des terminaux du langage.

Par conséquent, lors de la décompilation de l'arbre syntaxique d'un programme, notre paragrapheur associe à chaque noeud de cet arbre une indentation, qui représente l'endroit dans la ligne où ce noeud va être décompilé. Dans l'exemple précédent, si le noeud correspondant au non-terminal <PROC-DECL> est décompilé en colonne un de la ligne un, il en est de même du noeud <PROC-HEADER> et donc du terminal "PROCEDURE", tandis que le noeud <PROC-BODY> sera décompilé à partir de la quatrième colonne de la troisième ligne.

Ceci correspond bien au premier objectif fixé à un paragrapheur: faire ressortir les imbrications de notions utilisées dans les programmes -- ces notions pouvant être des déclarations de procédures, des listes d'instructions, des expressions et cetera.

Nous pouvons maintenant expliquer un peu plus en détail comment est associée une indentation à un noeud (en fait, de façon plus générique, à un non-terminal de la grammaire). La position située immédiatement à droite du signe = séparant la partie gauche d'une règle de sa partie droite est considérée représenter l'indentation définie pour la partie gauche. Chaque occurrence d'un élément du vocabulaire (terminal ou non-terminal) dans cette partie droite définit par sa position l'indentation associée à cet élément. Nous distinguerons pour l'instant deux cas pour de telles définitions:

- Si l'élément de vocabulaire commence une ligne dans la définition de la partie droite de la règle, son indentation est calculée à partir de son déplacement par rapport à la partie gauche.

- Sinon l'indentation choisie est la colonne courante lors de la décompilation.

Des exemples du premier cas sont les non-terminaux <PROC-HEADER> et <STMT-LIST> de l'exemple précédent; en revanche le non-terminal <PARAM*> correspond à la seconde possibilité.

Ceci permet par exemple d'aligner verticalement des listes. Ainsi, dans l'exemple suivant, nous spécifions que les éléments d'une liste de paramètres de procédure doivent être décompilés les uns au-dessous des autres:

```
<PARAM*>          = ( <PARAMETER-LIST> ) ;
<PARAMETER-LIST>  = <PARAMETER> ;
<PARAMETER-LIST>  = <PARAMETER-LIST>  ";"
                    <PARAMETER> ;
```

Cette grammaire spécifie un paragrapheur qui décompile la forme interne du texte

```
PROCEDURE READACHAR ( VAR F : TEXT ; VAR C : CHAR ) ;
```

en le texte

```
procedure READACHAR (var F : TEXT ;
                     var C : CHAR) ;
```

Il est à remarquer que ceci permet non seulement les indentations (comme dans le cas du non-terminal <COMPOUND-STMT> du précédent exemple) et les alignements (comme pour <PARAMETER-LIST> ci-dessus), mais également ce qu'il est en général convenu d'appeler des **exdentations**, c'est-à-dire les indentations négatives. Ainsi la spécification donnée ci-dessous a pour objet de faire ressortir visuellement la partie variante des enregistrements dans des programmes Pascal, en alignant le mot-clé "case" avec les mots-clés "record" et "end"; cet alignement est en effet possible, bien que la partie variante n'apparaisse pas en partie droite d'une règle définissant le non-terminal <RECORD-TYPE>: il suffit d'exdenter le non-terminal <VARIANT-PART> de la même quantité que son ancêtre <FIELD-LIST> a été indenté.

```
<RECORD-TYPE>  = record
                     <FIELD-LIST>
                 end ;
<FIELD-LIST>   = <FIXED-PART> ";"
               <VARIANT-PART> ;
<VARIANT-PART> = case  <TAG>  of
                     <VARIANT-LIST> ;
```

Le paragrapheur correspondant produira, à partir du texte

```
RECORD {...} ; CASE TAG OF {...} END
```

le texte paragraphé comme suit:

```
record
   {...} ;
case TAG of
   {...}
end
```

Ces principes simples ont été à la base d'une première implantation, décrite en [Lebon 81], appliquée par la suite à la décompilation d'arbres abstraits [Lebon 83]. Mais cette simplicité même s'est révélée à l'usage trop contraignante, bien que les résultats obtenus avec cette méthode soient très satisfaisants en regard du peu d'investissement nécessaire à l'obtention d'un paragrapheur. L'inconvénient de cette méthode est que tout le paragraphage est défini de façon relative: l'indentation choisie pour un non-terminal est déterminée à partir de l'indentation de la partie gauche ou de la colonne atteinte par la décompilation du terminal ou non-terminal situé juste avant dans la règle. Or il existe des cas particuliers où l'on désire un contrôle explicite. Un exemple

courant en est l'étiquette d'instruction; la portion de grammaire ci-dessous définit une indentation possible:

```
<STMT>  = <LABEL>  <UNLABELLED-STMT> ;
<LABEL> = %NATURAL  ":" ;
```

Cette spécification produit un paragraphage de ce type:

```
begin
   ...
   9 : ...
   ...
end
```

Il n'y a aucun moyen de spécifier que les étiquettes doivent être placées en première colonne; or l'utilisation des indenteurs de Pascal et PL/1 qui nous sont accessibles sur notre système d'exploitation nous a convaincus qu'il s'agit de la meilleure indentation possible pour permettre de "retrouver" un point de branchement dans un programme.

Nous avons donc décidé de compléter la syntaxe des spécifications de paragraphages. Il est désormais possible d'indiquer dans la description de la grammaire des "actions" qui devront être effectuées lors du paragraphage. Ces directives, écrites entre tildas (~), permettent de spécifier des "déplacements de la tête d'écriture" et se divisent en trois groupes: positionnement absolu, positionnement relatif et actions spéciales.

Le positionnement absolu est obtenu par la directive COL, qui prend en argument la colonne désirée. Parmi les directives de positionnement relatif, nous avons défini SPACE (qui accepte un argument positif ou négatif) et MARGIN (qui permet de se positionner dans la ligne courante sous le point qui a été défini comme indentation de la partie gauche de la règle courante). Il est à noter qu'aucune de ces actions ne permet d'effectuer de la sur-impression, par définition: le paragrapheur insère des fins de ligne le cas échéant.

En utilisant cette possibilité de spécification d'actions, nous pouvons maintenant modifier l'exemple précédent afin de positionner les étiquettes d'instructions en début de ligne:

```
<LABEL> = ~COL (1)~  %NATURAL  ":"  ~MARGIN~ ;
```

Ceci permet l'obtention du paragraphage ci-dessous:

```
begin
   ...
9 :  ...
   ...
end
```

Un autre problème qui se rencontre très fréquemment lors de la spécification d'un paragrapheur avec notre méthode a pu être résolu grâce à l'introduction de ces directives de paragraphage. Il s'agit du problème que posent les productions vides de la grammaire en relation avec l'espacement vertical. L'exemple ci-dessous permet de le mettre en évidence.

Un bloc en Pascal est constitué d'une partie déclarative suivie d'une partie impérative. Cette partie déclarative est formée de plusieurs types de déclarations qui ne doivent apparaître que dans un certain ordre mais peuvent ne pas être présents (1):

```
<BLOCK>      = <LABEL_PART>
               <CONST_PART>
               <TYPE_PART>
               <VAR_PART>
               <PROC_PART>
               <STMT_PART> ;
<LABEL_PART> = LABEL  <LABEL_LIST>  ";" ;
<LABEL_PART> = ;
...
```

Cette description, bien qu'elle semble correspondre au paragraphage désiré, présente l'inconvénient de ne pas tenir compte des parties déclaratives vides. Ceci fait que le bloc qui peut s'écrire

```
LABEL 6 ; BEGIN {...} END
```

sera paragraphé sous la forme suivante:

```
label 6 ;



begin
   {...}
end
```

en raison des fins de lignes séparant les parties déclaratives dans la production <BLOCK>; ceci n'est évidemment pas le résultat escompté.

Une solution possible à ce type de problème est de ne pas aller à la ligne entre les non-terminaux de la production <BLOCK>, mais à la fin de chaque production qui en dérive. Cette solution s'écrirait ainsi (2):

```
<BLOCK>      = <LABEL_PART> <CONST_PART> ~
               ~ <TYPE_PART> <VAR_PART> ~
               ~ <PROC_PART> <STMT_PART> ;
<LABEL_PART> = LABEL  <LABEL_LIST>  ";" ~
               ~ ~SKIP (1)~ ~MARGIN~ ;
<LABEL_PART> = ;
...
```

---

(1) Ainsi que le précise la troisième production de cette partie de grammaire, dont la partie droite est vide.

(2) Les directives de paragraphage vides (sauts de ligne et espace blanc entre deux tildas) ne sont là que pour nous permettre de couper une ligne physiquement mais pas logiquement. La directive SKIP permet d'insérer un certain nombre de fins de ligne et de se positionner en colonne un.

Cette solution n'est pas très esthétique et est très lourde à mettre en oeuvre; elle présente en outre l'inconvénient de ne pouvoir être appliquée à des non-terminaux apparaissant dans plusieurs productions, si l'on désire un paragraphage différent suivant la production.

Une directive particulière a donc été introduite pour résoudre ce problème. Elle permet d'inhiber l'instruction de changement de ligne qui serait normalement exécutée immédiatement après lors du paragraphage. Cette directive, qui a pour nom INH, ne peut se trouver qu'en fin de production. Notre exemple peut donc s'écrire de la façon suivante:

```
<BLOCK>       = <LABEL_PART>
                <CONST_PART>
                <TYPE_PART>
                <VAR_PART>
                <PROC_PART>
                <STMT_PART> ;
<LABEL_PART> = LABEL  <LABEL_LIST>  ";" ;
<LABEL_PART> = ~INH~ ;
...
```

On voit que l'introduction de cette directive spéciale nous permet de garder à la grammaire une forme très lisible, tout en résolvant le problème assez fréquent du paragraphage en présence de productions vides. Ce problème se pose en raison du faible nombre de primitives que nous avons décidé de fournir à l'utilisateur; nous sommes cependant persuadés que cette pauvreté du langage de spécification de paragraphage est un des intérêts majeurs de notre approche. En effet les spécifications obtenues sont toujours extrêmement lisibles -- nous espérons que les exemples cités pour le cas de Pascal sont convaincants --, le travail nécessaire à leur obtention est quasiment nul et l'apprentissage des directives de paragraphage est immédiat. Nous n'en voulons pour preuve que le mode d'emploi ci-dessous.

3 - Mode d'emploi du constructeur de paragrapheurs

Une grammaire décrivant les grammaires acceptées par le constructeur de paragrapheurs -- directives exclues -- est donnée en annexe A; l'analyseur lexical associé est décrit en annexe B. La spécification lexico-syntaxique des directives elles-mêmes fait l'objet de l'annexe C.

En l'absence de directive explicite, les conventions suivantes s'appliquent -- on dénommera unité un terminal, générique ou non, un non-terminal, le signe = séparant la partie gauche de la partie droite de toute règle ainsi que le point-virgule dénotant la fin de règle --:

- L'apparition d'un terminal non générique est une demande d'écriture de son texte à l'endroit où se trouvera la tête d'écriture (colonne courante pour le paragrapheur).

- L'apparition d'un terminal générique est de même une demande d'écriture du texte effectif correspondant, provenant du programme à paragrapher.

- L'apparition d'un non-terminal correspond à une demande de décompilation-paragraphage de l'arbre qu'il représente. L'indentation de base pour ce non-terminal sera la colonne courante lors du paragraphage.

- L'axiome est décompilé en colonne un.

- Tout espace blanc suivant une unité autre que la fin de règle sera répercuté dans le texte paragraphé, à un caractère près -- car un caractère blanc est nécessaire dans tous les cas pour séparer deux unités --.

- Toute suite de fins de lignes après une unité autre que la fin de règle sera répercutée dans le texte paragraphé.

- La position située deux colonnes à droite de l'unité = séparant la partie gauche de la partie droite d'une production est représentative de la colonne courante lors du début de la décompilation d'un sous-arbre correspondant à cette production.

- Le décalage entre la position de la première unité qui suit une fin de ligne et la position définie à l'alinéa ci-dessus est utilisé sans modification pour déterminer la colonne où cette unité doit être décompilée (ceci est valable également pour le point-virgule de fin de règle, bien qu'aucun texte ne soit produit).

En revanche, lorsqu'une ou plusieurs directives explicites de paragraphage sont présentes entre deux unités, ces conventions sont modifiées comme suit:

- Tout espace blanc entourant les directives est ignoré.

- Les fins de lignes et l'espace blanc à l'intérieur des directives sont également ignorés.

L'effet des directives elles-mêmes est le suivant:

~ ~	(directive vide) Permet de modifier l'aspect d'une règle sans influer sur le paragraphage.
~ COL (N) ~	Positionnement en colonne N; si cela est nécessaire, une fin de ligne sera insérée auparavant.
~ INH ~	Inhibition de la prochaine directive de fin de ligne si elle est rencontrée immédiatement ensuite (ni directive, ni écriture ne doit intervenir).
~ MARGIN ~	Positionnement en la colonne où la décompilation de la production courante a débuté. Une fin de ligne sera insérée auparavant si nécessaire.
~ MARGIN + N ~	Identique à: ~MARGIN~ ~SPACE (N)~.
~ MARGIN - N ~	Identique à: ~MARGIN~ ~SPACE (-N)~.
~ PAGE (N) ~	Insertion de N sauts de page et positionnement en colonne un.

~ SKIP (N) ~	Insertion de N fins de ligne et positionnement en colonne un.
~ SPACE (Z) ~	Insertion de Z caractères blancs si Z est positif; suppression des -Z caractères blancs précédents si Z est négatif. Dans ce dernier cas, si la colonne courante n'est pas précédée d'un espacement suffisant, la tête d'écriture est préalablement positionnée à la même colonne sur la ligne suivante; si cela ne suffit pas, une erreur est signalée et le positionnement se fait en colonne un.
~ TAB (N) ~	Insertion du nombre de caractères blancs nécessaires pour se trouver à la N-ième position de tabulation qui suit la colonne courante lors du paragraphage (une position de tabulation étant une colonne dont le numéro est congru à un modulo dix).

Dans cette description, N est un entier naturel non nul et Z un entier relatif non nul. Il est possible de ne pas spécifier d'argument aux directives COL, SKIP, PAGE et SPACE; la valeur 1 est alors prise par défaut.

4 - Fonctionnement du paragrapheur

Comme nous l'avons précisé précédemment, le paragrapheur fonctionne par décompilation d'une forme interne du texte à paragrapher. La spécification du paragraphage désiré étant faite sur la grammaire du langage, cette forme interne est naturellement dérivée de l'arbre syntaxique correspondant à l'analyse du texte suivant cette grammaire.

Chaque noeud de cet arbre correspond soit à un non-terminal de la grammaire, soit à un terminal générique; les premiers sont étiquetés par (le numéro de) la production correspondante de la grammaire, les seconds sont constitués par des actualisations (instanciations) des terminaux génériques. Les terminaux non génériques (mots-clés, symboles spéciaux) sont absents de l'arbre, puisque la connaissance de la production de la grammaire suffit à les déterminer: ceci permet d'économiser de la place dans l'arbre.

Une fois cet arbre construit, le paragrapheur en effectue un parcours "haut-bas gauche-droite", exécutant pour chaque noeud les directives -- implicites ou explicites -- spécifiées dans la production correspondante. Ainsi l'apparition dans une production d'un non-terminal ou d'un terminal générique est considérée comme une directive implicite de décompilation du sous-arbre correspondant; de même la présence d'un terminal non générique est une directive implicite d'écriture de ce terminal.

Les terminaux, génériques ou non, sont écrits en tenant compte d'un certain nombre d'options que l'utilisateur peut choisir. Ces options permettent une "mise en forme" des éléments lexicaux du texte. Elles sont actuellement au nombre de huit. Les quatre premières permettent de choisir les types de caractères à utiliser pour l'écriture des terminaux génériques: il est possible de spécifier, pour chaque type de terminal générique, s'il faut l'écrire en majuscules, en minuscules, en minuscules avec l'initiale de chaque mot en majuscule, ou tel qu'il apparaissait dans le texte d'origine. Les quatre autres options permet-

tent d'effectuer le même type de choix pour les terminaux non génériques (c'est-à-dire, en l'occurrence, les "mots-clés").

Pour des langages sans commentaires (autres que les espacements horizontaux et verticaux) et sur des systèmes d'exploitation permettant l'utilisation d'un nombre quelconque de caractères pour une ligne de programme, ceci suffit à déterminer complètement le fonctionnement du paragrapheur.

Une telle situation n'est pas courante. C'est pourquoi se posent deux problèmes majeurs: la coupure de lignes trop longues (l'utilisateur désirant en général se limiter soit à une ligne d'écran soit à une ligne d'imprimante) et le positionnement des commentaires. Ce sont ces problèmes, et les approches de solutions que nous avons adoptées dans notre système et son implantation actuelle, qui font l'objet du paragraphe suivant.

5 - Problèmes liés au paragraphage

5.1 - Lignes trop longues

Le problème qui semble a priori le plus épineux est celui de la coupure et de l'indentation des lignes trop longues. Ce problème est en fait caractéristique de la notion d'indentation, et les indenteurs eux-même n'y échappent pas. Pour un paragrapheur cependant il se pose avec plus d'acuité, puisque la solution adoptée devrait être satisfaisante pour tous les cas possibles dans tous les langages... Nous nous sommes actuellement arrêtés à une stratégie double: d'une part le paragrapheur limite l'indentation maximale d'une ligne, de façon à éviter de commencer une ligne avec si peu d'espace libre qu'elle a "de fortes chances" d'être scindée; d'autre part, si une ligne doit être scindée, la césure est réalisée le plus à droite possible, et la tête d'écriture est positionnée cinq caractères plus à droite que le premier caractère de la ligne coupée, avant de débuter l'écriture du reste de la ligne.

Les inconvénients de cette stratégie sont évidents: un bloc de programme peut ne pas être indenté à sa place normale, en raison de la première stratégie, alors qu'une indentation normale n'introduirait aucune coupure supplémentaire; par ailleurs la scission d'une ligne se fait sans respecter sa structure, contrairement au but premier d'un paragrapheur.

Cette stratégie a en revanche deux avantages primordiaux: elle ne nécessite pratiquement pas de calcul auxiliaire ni d'espace mémoire annexe, et elle réduit le nombre de coupures nécessaires dans le cas général. Elle constitue un compromis qui semble acceptable dans la plupart des cas, et de plus n'impose à l'implanteur d'un paragrapheur aucune spécification supplémentaire et respecte donc notre but initial de simplicité.

Il serait par exemple possible d'ajouter des directives "globales" servant à signaler au paragrapheur que telle liste doit être décompilée horizontalement, sauf en cas de débordement, où il conviendrait de la décompiler verticalement (ou en blocs...). L'introduction de telles directives serait bien entendu possible, mais se ferait au détriment de la localité des spécifications, et donc de la simplicité que nous nous étions fixée initialement. D'autant qu'une telle spécification, pour être complète, devrait prévoir toutes les éventualités de

débordement en précisant une stratégie de choix lorsque plusieurs coupures sont possibles pour une même ligne... Or il est relativement rare qu'une ligne ait à être coupée, et en outre, dans la plupart des cas où cette situation se présente, le programmeur lui-même a des difficultés à trouver un paragraphage satisfaisant; les investissements nécessaires pour, d'une part déterminer quelles spécifications devrait accepter notre système, d'autre part utiliser ces spécifications pour décrire le paragraphage désiré dans tous les cas de césure qui peuvent se présenter dans un langage réaliste, nous semblent démesurés en regard de la rareté des cas qui seraient ainsi résolus.

Nous envisageons cependant d'étudier une stratégie adaptée de celle décrite dans [Oppen 80], en considérant que tout sous-arbre à décompiler forme un bloc et en cherchant à déterminer (de façon simple) quels blocs scinder afin d'obtenir le nombre minimal de coupures.

En tout état de cause nous conserverons la stratégie actuelle de limitation de l'indentation d'une ligne à une valeur maximale, stratégie qui donne d'excellents résultats dans le cas général. Cette limitation de l'indentation est dirigée par une option du paragrapheur qui permet, au choix de l'utilisateur, soit d'arrêter l'indentation en une certaine colonne, soit de décaler la ligne en cas de dépassement de cette colonne. Un exemple va nous permettre d'éclairer notre propos. Soit le fragment de programme suivant, qui apparaît ici dans sa forme paragraphée avant césure:

```
if CARACTERE in LETTRES then
   begin
      INDEX := 0;
      repeat
         if INDEX < TAILLECHAINE then
            begin
               INDEX := SUCC (INDEX);
               CHAINE [INDEX] := CARACTERE
            end;
         LIRE (CARACTERE)
      until not (CARACTERE in [LETTRES + CHIFFRES]);
      TRAITER (IDENTIFICATEUR, CHAINE)
   end
else
   ...
```

Si ce fragment est très imbriqué à l'intérieur d'un programme, certaines lignes vont devoir être coupées. Comme nous l'avons vu plus haut, le paragrapheur, afin de limiter le nombre de coupures, limite l'indentation de chaque ligne à une valeur maximale. Si l'utilisateur choisit d'effectuer cette limita-

tion par arrêt de l'indentation, le résultat obtenu sera le suivant:

```
if CARACTERE in LETTRES then
   begin
      INDEX := 0;
      repeat
      if INDEX < TAILLECHAINE then
      begin
      INDEX := SUCC (INDEX);
      CHAINE [INDEX] := CARACTERE
      end;
      LIRE (CARACTERE)
      until not (CARACTERE in [
           LETTRES + CHIFFRES]);
      TRAITER (IDENTIFICATEUR,
           CHAINE)
   end
else
   ...
```

On voit que, pour une partie du programme obtenu, l'imbrication de notions est perdue. Si en revanche l'utilisateur opte pour un décalage de l'indentation en cas de débordement, le paragrapheur produira

```
                  if CARACTERE in LETTRES then
                     begin
                        INDEX := 0;
                        repeat
      if INDEX < TAILLECHAINE then
         begin
            INDEX := SUCC (INDEX);
            CHAINE [INDEX] := CARACTERE
         end;
      LIRE (CARACTERE)
                           until not (CARACTERE in [
                                LETTRES + CHIFFRES]);
                           TRAITER (IDENTIFICATEUR,
                                CHAINE)
                     end
                  else
                     ...
```

Les coupures de lignes se produisent, sur cet exemple, aux mêmes endroits que dans le cas précédent, mais l'indentation reflète cette fois l'imbrication de toutes les parties du programme, sauf, très localement, aux points où est mise en oeuvre la stratégie de limitation. Il est aisé de remarquer qu'en outre, dans ce deuxième cas, le nombre de coupures est au plus égal à celui obtenu avec la première option. Le choix entre ces deux options est uniquement matière de goût personnel, et c'est pourquoi il est laissé à l'utilisateur du paragrapheur.

5.2 - Commentaires

Le second problème qui se pose lors de la spécification d'un paragrapheur est général au domaine de la décompilation de formes internes de programmes. Il s'agit de déterminer l'emplacement adéquat pour tout commentaire apparaissant dans un programme. Les indenteurs, qui utilisent directement le découpage du texte source en lignes, n'ont pour leur part qu'un problème de positionnement horizontal des commentaires; il leur suffit de déterminer les quantités d'espacement qui doivent précéder et suivre chaque commentaire dans sa ligne. Cela est en général réalisé en plaçant certains commentaires en première colonne, en plaçant un blanc avant et après les commentaires encastrés dans une ligne, en alignant sur une certaine colonne les commentaires qui se trouvent en fin de ligne, et autres heuristiques...

Un paragrapheur en revanche doit déterminer à quelle unité se réfère un commentaire pour pouvoir le placer de façon adéquate. Ceci pose un problème extrêmement délicat (psychologique, ergonomique ...) qui pour l'instant n'a pas été résolu dans le cas général. Certains systèmes de paragraphage utilisent une forme interne arborescente accessible à l'utilisateur (au moyen d'un éditeur [DHKLL 75, Vercoustre 83]); l'utilisateur a alors la possibilité d'"accrocher" ses commentaires à son gré, en position préfixe ou suffixe de tout noeud de cet arbre. Ce n'est pas le cas de notre système, qui utilise directement un arbre produit par un analyseur syntaxique; les commentaires sont en l'occurrence attachés par l'analyseur lexical à l'unité lexicale qui les suit.

Le problème posé par les commentaires dans notre système de paragraphage est donc double: d'une part l'arbre construit lors de l'analyse syntaxique doit comprendre, bien évidemment, tous les commentaires, sans excepter ceux qui sont attachés à des unités lexicales qui n'apparaissent pas dans l'arbre, et ces commentaires doivent être attachés dans l'arbre de telle manière que le paragraphage soit idempotent (1); d'autre part le calcul des espacements verticaux et horizontaux à placer autour d'un commentaire devrait prendre en compte la nature (**préfixée** ou **suffixée** (2)) de ce commentaire...

Ce problème de positionnement des commentaires nous a semblé relativement mineur lors de la spécification de Paradis. Nous avons donc décidé qu'il était préférable d'implanter la partie majeure de notre système (qui concerne le paragraphage des unités syntaxiques) sans trop le prendre en considération, préférant différer une décision à ce sujet jusqu'à un moment où nous aurions plus d'éléments pour l'aborder. Cette approche nous a permis d'accumuler de l'expérience quant aux heuristiques que nous avons dû développer au fur et à mesure, au vu de leurs résultats; ce sont ces heuristiques que nous allons maintenant exposer.

---

(1) C'est-à-dire que le résultat obtenu par paragraphage d'un texte déjà paragraphé soit ce texte (si P est la fonction de paragraphage et T le texte, P (P (T)) = P (T). En fait la stratégie que nous avons adoptée ne nous permet de garantir que l'égalité P (P (P (T))) = P (P (T)).

(2) Il est souvent d'usage de commenter en préfixé une partie de programme par exemple, et en suffixé une expression.

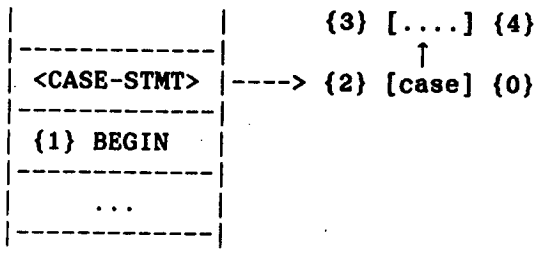
Le premier problème à résoudre était celui de la sauvegarde des commentaires que l'analyseur lexical rattache à une unité non générique. En outre, la résolution de ce problème devait faire en sorte d'éviter, autant que possible, de faire franchir une telle unité à un commentaire. Ceci a été fait au moyen d'un algorithme relativement simple, qui rattache, lors de la construction de l'arbre, tout commentaire posant problème à un non-terminal adéquat. Cet algorithme, lors de toute réduction d'une règle syntaxique, parcourt la pile d'analyse de bas en haut dans les limites de la règle en question et, pour chaque commentaire associé à une unité lexicale non générique, prend une décision en fonction de la position de cette unité dans la règle:

- s'il s'agit de la première unité de la règle, le commentaire est rattaché en préfixé du noeud en cours de construction;
- sinon, si cette unité est immédiatement précédée par un non-terminal ou par un terminal générique, le commentaire est rattaché en suffixé de celui-ci;
- sinon, s'il se trouve un non-terminal ou un terminal générique plus à droite dans la règle, le commentaire est rattaché en préfixé de celui-ci;
- sinon le commentaire est rattaché en suffixé du noeud en cours de construction (1).

Cet algorithme offre l'avantage non négligeable d'être facile à implanter, et de permettre de différencier des commentaires associés à deux terminaux non génériques consécutifs dans le texte du programme, mais provenant de deux règles syntaxiques distinctes (il est rare pour les langages de programmation que deux "mots-clés" soient juxtaposés dans une règle de grammaire). Prenons par exemple le fragment de programme Pascal suivant:

```
PROCEDURE... {1} BEGIN {2} CASE {3} ... {4} END {5} END
```

dans lequel le premier "END" termine une instruction CASE, et le second le corps d'une procédure. A l'issue de la réduction syntaxique qui fait intervenir le premier "END", la pile d'analyse peut avoir la forme suivante (2):

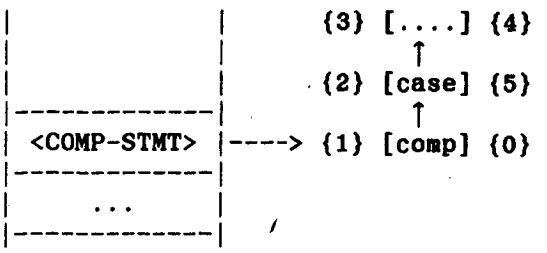


---

(1) Ces trois derniers cas nécessitent la possibilité de concaténer des commentaires; cette possibilité de concaténation est de toute façon utilisée également lors de l'analyse lexicale, puisqu'un nombre quelconque de commentaires peuvent en général précéder toute unité lexicale dans les langages de programmation.

(2) Nous représentons ici chaque noeud de l'arbre construit par son nom encadré par des crochets, précédé du commentaire préfixé et suivi du commentaire suffixé qui lui sont associés. L'absence de commentaire est représentée par un commentaire {0}.

A l'issue de la première réduction syntaxique faisant intervenir le second "END", cette pile et l'arbre associé seront comme suit:



On voit bien sur cet exemple qu'il est aisé, lors de la décompilation de l'arbre en un parcours haut-bas, gauche-droite, de restituer les commentaires juste avant l'unité lexicale à laquelle ils sont associés dans le texte de départ. Il suffit de prendre en compte la présence d'un commentaire préfixé lors de la visite héritée d'un noeud et la présence d'un commentaire suffixé lors de la visite synthétisée de ce même noeud.

Le second volet du problème posé par les commentaires est plus délicat à résoudre, surtout dans le cadre général où nous nous plaçons. Il s'agit en effet de déterminer l'emplacement physique sur la page où doit être positionné un commentaire. Nous avons, dans un premier temps, choisi là aussi la simplicité d'implantation et recouru à des heuristiques relativement simples, et qui semblaient devoir donner de bons résultats dans le cas général. Nous distinguons donc a priori, lors de la décompilation-paragraphage d'un texte, quatre sortes de commentaires:

- Commentaire composé uniquement de sauts de lignes ou de sauts de pages; les sauts de pages éventuels sont effectués, puis le paragrapheur s'assure que la prochaine unité lexicale sera sortie sur une nouvelle ligne, en une colonne déterminée par le mécanisme général de coupure de lignes trop longues.

- Commentaire ne contenant pas de fin de ligne; si la ligne en cours d'écriture n'est pas vide, ce commentaire est écrit, précédé et suivi d'un blanc, dans la ligne courante, et le paragraphage continue sans en tenir plus compte; si en revanche la ligne courante est vide, le commentaire est écrit seul sur cette ligne, en la colonne courante, et la tête d'écriture est repositionnée en la même colonne sur la ligne suivante. Un tel commentaire suit donc en principe les déplacements horizontaux du texte auquel il est associé.

- Commentaire commençant par un déplacement vertical (saut de ligne ou de page); ce commentaire est placé en colonne un, précédé d'une ligne vide ou d'un saut de page. La tête d'écriture est ensuite repositionnée, après une ligne vide, en la colonne originelle. Il s'agit donc là d'un bloc.

- Commentaire contenant des déplacements verticaux. Dans le cas général, la première ligne de ce commentaire est sortie à la marge courante sur une ligne vide; un cas particulier a été prévu pour les commentaires "style Ada", qui ne contiennent un "terminateur de ligne" qu'en fin: un tel commentaire est sorti en la colonne courante, éventuellement précédé d'un blanc. Dans tous les cas la tête d'écriture est repositionnée sur une ligne vide en la colonne originelle.

On voit que ces définitions ne sont finalement pas simples. En outre les traitements associés interagissent: un commentaire du second type ci-dessus peut se trouver, après paragraphage, précédé de lignes vides et donc être "promu" dans la troisième catégorie lors d'un paragraphage ultérieur (c'est la raison pour laquelle le paragraphage n'est pas "immédiatement" idempotent). Enfin, malgré l'implantation assez complexe que nous avons réalisée, elles ne permettent pas toujours au programmeur qui utilise le paragrapheur de placer ses commentaires à l'endroit qu'il désire (1). Ce comportement criticable provient de deux suppositions que nous avions faites afin de simplifier notre implantation d'une part et le travail du spécificateur d'un paragrapheur d'autre part. Tout d'abord, nous n'utilisons pas toute l'information qui nous est accessible quant au positionnement originel d'un commentaire: lors de la construction de l'arbre, seules les unités lexicales et syntaxiques avoisinantes nous servent à placer un commentaire en position préfixée ou suffixée d'une unité; nous pourrions placer l'accent sur une notion de "proximité" dans le texte plutôt que sur la disponibilité dans l'arbre construit. Par ailleurs, nous utilisons pour forme interne des commentaires leur texte tel qu'il apparaît dans le programme, sous forme d'une chaîne; l'intervention de l'utilisateur à ce niveau nous permettrait plus de richesse et de précision dans la classification des différents types de commentaires.

Nous prévoyons donc de modifier notre stratégie actuelle de traitement des commentaires, d'une part en définissant une notion de distance entre un commentaire et les unités lexicales qui l'entourent, afin de l'associer à l'unité qui est pour le programmeur la plus proche visuellement, d'autre part en permettant -- sans la requérir -- l'intervention de la personne qui décrit le paragrapheur, pour qu'elle puisse éventuellement influer sur la forme de sortie du commentaire et son positionnement (certains indenteurs et paragrapheurs de Lisp, par exemple, placent en une colonne précise tout commentaire commençant par ; et en première colonne tout commentaire débutant par ;;;).

6 - Implantation

Nous décrivons ici brièvement l'implantation actuelle du système Paradis. L'utilisation de ce système se décompose en deux parties: un **constructeur** engendre, à partir des spécifications fournies par l'utilisateur, des tables d'analyse et de paragraphage; ces tables peuvent ensuite être exploitées par un **paragrapheur** général, pour analyser puis décompiler tout texte source, comme il est dit au §4 ci-dessus.

6.1 - Le constructeur

Nous devons donc, à partir d'une grammaire enrichie de directives de paragraphage -- implicites ou explicites --, produire:

- des tables d'analyse (lexicale et syntaxique);
- le code correspondant aux directives de paragraphage.

---

(1) Il n'est notamment pas possible, en Pascal, de commenter une instruction après le point virgule qui la termine, sur la même ligne: un tel commentaire sera considéré comme étant préfixé de la première unité lexicale de l'instruction qui suit, et comme tel en général écrit seul sur la ligne suivante.

Les tables d'analyse, représentant un automate à pile de la classe LR, sont produites en utilisant le système SYNTAX ([Boullier 84]). La spécification d'entrée est une BNF enrichie, dans laquelle les directives explicites de paragraphage (commandes entre tildas) sont considérées comme de simples commentaires et donc ignorées par le constructeur d'analyseur. Les spécifications de paragraphage sont traitées par un module séparé qui joue, vis-à-vis de SYNTAX, le rôle d'"extracteur de sémantique". Ce module n'a pratiquement aucune influence sur le traitement de la partie syntaxique.

On peut remarquer tout de suite que la sortie d'un texte paragraphé ne peut que difficilement s'effectuer en parallèle avec l'analyse syntaxique; cela est dû principalement à deux causes:

- l'analyse LR est une méthode ascendante, et par conséquent la reconnaissance des manches dans les formes sentencielles successives ne suit pas strictement un parcours gauche-droit du texte source,
- et surtout le positionnement du texte associé à une phrase d'un symbole de la grammaire utilise en général des informations contextuelles, non encore connues en cours d'analyse.

Il est donc nécessaire de prévoir une forme intermédiaire contenant le résultat de l'analyse. Nous avons opté pour l'arbre concret d'analyse élagué des terminaux non génériques (voir au §4). Cet arbre est construit en parallèle avec l'analyse syntaxique du texte à paragrapher. La sortie du texte s'effectue en parcourant l'arbre d'analyse de haut en bas et de gauche à droite, et, simultanément, le code résultant de la traduction des directives de paragraphage.

On peut considérer en première analyse qu'à chaque règle syntaxique est associée une séquence de code analogue au corps d'un sous-programme et que l'arbre syntaxique associé à un texte donné est la structure qui contrôle les appels à ces sous-programmes. Cet arbre sert également à récupérer les textes des commentaires et des terminaux génériques alors que le code spécifie essentiellement les écritures de terminaux non génériques et les déplacements de la tête d'écriture.

Considérons de nouveau l'exemple du §2 spécifiant le paragraphage d'un <RECORD-TYPE>. Le code engendré est le suivant:

Règle	Code
<RECORD-TYPE> = record <FIELD-LIST> end ;	print ("record") skip (1) margin+3 call (<FIELD-LIST>) skip (1) margin print ("end") return

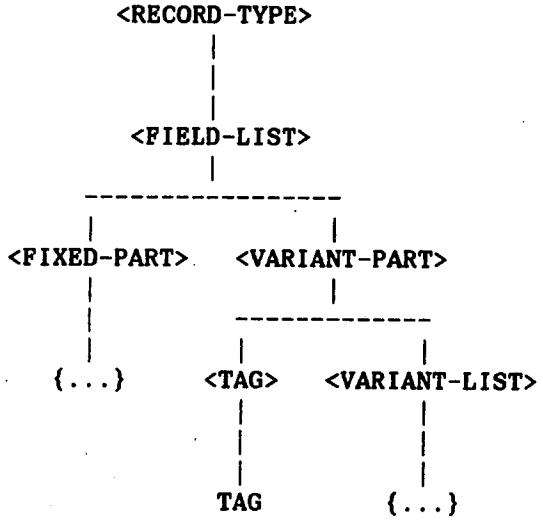
```
<FIELD-LIST>    = <FIXED-PART> ";"          call (<FIXED-PART>)
                  <VARIANT-PART> ;          print (";")
                                            skip (1)
                                            margin-3
                                            call (<VARIANT-PART>)
                                            return

<VARIANT-PART> = case  <TAG>  of            print ("case")
                       <VARIANT-LIST> ;     space (1)
                                            call (<TAG>)
                                            space (1)
                                            print ("of")
                                            skip (1)
                                            margin+3
                                            call (<VARIANT-LIST>)
                                            return
```

L'analyse du texte

```
RECORD {...} ; CASE TAG OF {...} END
```

produit l'arbre concret élagué



Un parcours haut-bas gauche-droite de cet arbre, associé au code engendré pour chaque règle, produit le texte paragraphé

```
record
   {...} ;
case TAG of
   {...}
end
```

Remarquons que, pour une règle donnée, le nombre d'instructions "call" est égal au nombre total de terminaux génériques et de non-terminaux de sa partie droite. L'impression d'un terminal non générique t est précisée par l'instruction print ("t"). Les instructions spécifiant les déplacements élémentaires de la tête d'écriture correspondent exactement aux spécifications explicites de la

fin du §3. La production d'un tel code est extrêmement simple:

- Pour les spécifications implicites, il suffit de repérer la position textuelle du début du symbole en cours d'examen par rapport à la fin de l'unité qui le précède et de produire le code assurant un déplacement équivalent de la tête d'écriture, puis de produire le code ("call" ou "print") assurant la sortie du symbole en main.

- Les spécifications explicites sont traduites directement en leur code associé.

- La fin de la séquence de code associée à une règle est marquée par une instruction "return".

Le traitement des spécifications explicites de paragraphage est illustré par

```
<LABEL> = ~COL (1)~  %NATURAL  ":"  ~MARGIN~ ;
                                        col (1)
                                        call (%NATURAL)
                                        space (1)
                                        print (":")
                                        margin
                                        return
```

Afin de faciliter la lecture du code, nous associons, dans la liste sortie par le constructeur, à chaque instruction "call" un paramètre donnant le nom du symbole auquel elle s'applique; en fait le nom de ce symbole (non-terminal ou terminal générique) figurera également dans les arbres et il est donc inutile de le préciser dans le code. Le code associé à une règle est par conséquent essentiellement caractéristique des déplacements de la tête d'écriture et ne dépend pas des noms des symboles de sa partie droite. Dans ces conditions, deux règles différentes peuvent avoir le même code de paragraphage; ce code peut donc être réutilisé. L'optimisation résultant de cette remarque assure un gain en place de l'ordre de deux ou trois sur le code engendré.

6.2 - Le paragrapheur

6.2.1 - Analyse du texte source

L'analyse (lexicographique et syntaxique) d'un texte à paragrapher est réalisée par les analyseurs de SYNTAX conformément aux grammaires spécifiant le langage d'écriture de ce texte.

La partie lexicale du langage à paragrapher doit être décrite en utilisant le constructeur lexicographique de SYNTAX. Cette description diffère généralement de celle utilisée pour une traduction "normale", par le fait que la plupart des caractères rencontrés doivent être gardés; par exemple, pour un compilateur Ada, la spécification des nombres entiers peut être:

```
ENTIER = CHIFFRE  { [-"_"] ~CHIFFRE }
```

Cette spécification nous indique qu'un nombre entier en Ada commence par un chiffre et est suivi d'un nombre quelconque (éventuellement nul) de chiffres; chacun de ces chiffres peut être précédé d'un blanc souligné qui, s'il existe,

est supprimé de la forme interne.

Exemple: forme externe "10_000" — forme interne "10000"

De cette façon, la procédure de conversion de chaîne en nombre ne manipulera que des chiffres. En revanche, pour spécifier un paragrapheur de Ada, les entiers doivent être décrits par

```
ENTIER = CHIFFRE  { ["_"]  CHIFFRE }
```

ce qui permet à l'utilisateur du paragrapheur de retrouver les nombres tels qu'il les avait écrits.

Ce point est particulièrement crucial pour la description des commentaires:

- Les espacements, tabulations, retours à la ligne et sauts de page doivent en général être supprimés à l'extérieur du texte des commentaires proprement dits; dans le cas contraire, en effet, ils pourraient modifier les directives de paragraphage et produire un éloignement cumulatif des unités lexicales au cours des paragraphages successifs.

- Les commentaires proprement dits doivent être conservés afin de figurer dans le texte paragraphé.

Les tables d'analyse utilisées par l'analyseur syntaxique sont celles produites par SYNTAX à partir de la grammaire dans laquelle les directives (implicites ou explicites) de paragraphage sont ignorées -- elles sont considérées comme de simples commentaires --.

Chaque fois que la partie droite d'une règle de la grammaire est reconnue, l'analyseur syntaxique appelle une action qui construit la portion de l'arbre d'analyse associée à cette règle.

6.2.2 - Construction de l'arbre

L'arbre syntaxique est construit de bas en haut au fur et à mesure de l'analyse. Cette construction s'effectue très simplement en faisant évoluer une pile, parallèlement à la pile d'analyse, de telle manière qu'à chaque non-terminal corresponde le sous-arbre qui lui est associé. Chaque fois que l'analyseur reconnaît la partie droite d'une production, il y a création d'un noeud dont le nom est le numéro de cette production et dont les fils sont les noeuds correspondant aux non-terminaux et aux terminaux génériques de sa partie droite. Ce nom permet donc de repérer la portion de code associée au paragraphage de cette règle.

Les feuilles de cet arbre correspondent

- soit à des terminaux génériques

- soit à des productions dont la partie droite est vide ou formée uniquement de terminaux non génériques.

Outre son nom, chaque noeud comporte

- des informations sur les commentaires qui lui sont éventuellement attachés (voir le §5.2);

- des informations structurelles permettant de connaître sa position dans l'arbre. Ces informations structurelles sont bien entendu étroitement liées à l'utilisation (décompilation) de cet arbre.

Le paragraphage résultant d'un parcours haut-bas gauche-droit de cet arbre, il suffit que chaque noeud soit relié à son père, à son frère de droite et à son premier fils. On s'aperçoit en fait qu'un lien statique vers le père est inutile, car il suffit de le conserver dynamiquement dans une pile reflétant l'imbrication des niveaux de l'arbre: chaque code "call" augmentant ce niveau de un, chaque code "return" le diminuant d'une unité. En poursuivant dans cette voie, on constate que les liens "premier fils" et "frère de droite" peuvent eux-mêmes être fusionnés en un lien vers le noeud suivant, pour former un arbre complètement linéarisé. Chaque instruction "call" est alors interprétée

- en empilant l'adresse de l'instruction suivante,

- puis en allant exécuter le code correspondant au nom du noeud référencé par le lien suivant.

L'instruction "return" se contente dans ce schéma de dépiler et d'aller exécuter le code se trouvant à l'adresse ainsi récupérée.

Pour linéariser l'arbre au fur et à mesure de la construction il suffit, pour chaque sous-arbre associé à un non-terminal donné, de mémoriser le début et la fin de la liste linéaire le représentant. Chaque réduction effectue alors la concaténation des listes partielles associées aux non-terminaux et des listes triviales de longueur un associées aux terminaux génériques; le résultat de cette concaténation est une nouvelle liste linéaire qui est ensuite associée au non-terminal en partie gauche.

6.2.3 - La décompilation

Le fonctionnement global du décompilateur a été esquissé au §4, et certains détails d'implantation ont été abordés au §5.

La première version du décompilateur-paragrapheur était une implantation directe de ce qui a été exposé précédemment. Nous avons ensuite consacré beaucoup d'attention à son optimisation. Une des améliorations les plus notables que nous y avons apporté est l'utilisation d'un mécanisme de sauvegarde des textes des mots-clés et des unités génériques après modifications lexicographiques -- passage en minuscules, en gras... --. Mais le gain le plus important que nous ayons réalisé mérite d'être souligné, car il a été obtenu par une technique d'implantation plus que par une optimisation d'un point particulier de l'algorithme.

Le parcours haut-bas gauche-droit de l'arbre d'analyse était initialement effectué par une procédure prenant en paramètres un noeud de l'arbre, la marge à prendre en compte pour le paragraphage de ce noeud, ainsi que la marge d'erreur

courante (voir le §5.1). Chaque instruction "call" donnait lieu à un appel récursif de cette procédure, avec pour arguments le prochain fils à décompiler, la colonne courante et la marge d'erreur à utiliser.

Le remplacement de cette récursion par une sauvegarde des paramètres avant leur modification a permis d'obtenir un gain extraordinairement important sur la vitesse d'exécution de l'ensemble du processus de décompilation-paragraphage: plus de 40%!

7 - Expérience

La version initiale de Paradis date de 1981. Depuis cette époque, le système, en constante évolution, a été et est utilisé sur de nombreuses applications.

On peut tout d'abord citer les différents langages définis au sein de l'équipe Langages et Traducteurs:

- Langages de description du niveau lexicographique pour SYNTAX (10CL, new_10cl, lecl).
- Divers langages d'entrée de Perluette [DMR 82].
- Langages de description du traitement des erreurs pour SYNTAX.

L'expérience d'utilisation de Paradis a bien entendu été menée sur des langages plus classiques parmi lesquels on peut citer Pascal, PL/1, C et Ada. Paradis fait même partie intégrante de deux systèmes de traduction du type "source à source", l'un de Pascal vers Ada [BDJ 84], l'autre de PL/1 vers C [Mazaud 85]; Paradis assure dans ces deux systèmes le paragraphage des textes objets produits, à savoir Ada pour le premier et C pour le second.

Comme beaucoup de progiciels réalisant des transformations sur des textes, Paradis a même été "détourné" pour réaliser ce que l'on pourrait appeler un extracteur-paragrapheur. Celui-ci permet, à partir d'une spécification syntaxique et sémantique d'un langage, d'en extraire la grammaire et de la paragrapher. Cette extraction se réalise simplement en décrivant, au niveau lexical, la partie sémantique de la définition comme étant des commentaires de la grammaire et en supprimant ces commentaires.

L'expérience a montré que le système est souple d'emploi, efficace en temps et que le paragrapheur d'un langage dont on possède la grammaire s'obtient à un coût pratiquement nul.

On doit cependant signaler qu'une grammaire qui est adaptée pour une traduction peut ne pas convenir parfaitement à la description d'un paragrapheur et doit en conséquence être localement modifiée.

Supposons que l'on veuille, en Pascal, obtenir le paragraphage suivant: dans une liste d'instructions, une instruction composée doit être séparée de l'ins-

truction qui la précède par une et une seule ligne vide; s'il s'agit d'une instruction simple elle est séparée de celle qui la précède par une ligne vide si et seulement si cette dernière est une instruction composée; on ne doit pas sauter de ligne devant la première instruction d'une liste. Pour réaliser une telle spécification, il suffit de différencier

- les listes d'instructions dont la dernière est une instruction simple: les "A-STATEMENT-LIST";

- les listes d'instructions dont la dernière est une instruction composée: les "B-STATEMENT-LIST";

puis de réaliser la description suivante:

```
<STATEMENT-LIST>   = <A-STATEMENT-LIST> ;
<STATEMENT-LIST>   = <B-STATEMENT-LIST> ;
<A-STATEMENT-LIST> = <A-STATEMENT-LIST>
                     <SIMPLE-STATEMENT> ;
<A-STATEMENT-LIST> = <B-STATEMENT-LIST>

                     <SIMPLE-STATEMENT> ;
<A-STATEMENT-LIST> = <SIMPLE-STATEMENT> ;
<B-STATEMENT-LIST> = <A-STATEMENT-LIST>

                     <COMPOUND-STATEMENT> ;
<B-STATEMENT-LIST> = <B-STATEMENT-LIST>

                     <COMPOUND-STATEMENT> ;
<B-STATEMENT-LIST> = <COMPOUND-STATEMENT> ;
```

L'exemple du §5.1, traité avec cette spécification, se paragraphe en:

```
if CARACTERE in LETTRES then
   begin
      INDEX := 0;

      repeat
         if INDEX < TAILLECHAINE then
            begin
               INDEX := SUCC (INDEX);
               CHAINE [INDEX] := CARACTERE
            end;

         LIRE (CARACTERE)
      until not (CARACTERE in [LETTRES + CHIFFRES]);

      TRAITER (IDENTIFICATEUR, CHAINE)
   end
else
   ...
```

Les modifications qu'il faut parfois apporter à la description syntaxique d'un langage pour obtenir le paragraphage désiré sont en général minimes. Il faut cependant noter que, par essence même, les paragrapheurs construits par Paradis sont basés sur une grammaire non contextuelle du langage, ce qui signifie

que, plus la quantité d'information contextuelle nécessaire au paragraphage d'une unité est grande, plus la grammaire risque de s'éloigner de la forme "naturelle".

8 - Conclusion

Nous avons présenté un système de paragraphage dirigé par la syntaxe: la spécification d'un paragrapheur se fait, de façon très simple et lisible, sur la grammaire du langage; le paragrapheur correspondant construit un arbre proche de l'arbre d'analyse syntaxique, puis le décompile en exploitant un code engendré à partir de la spécification. Les exemples donnés sont tirés d'une spécification d'un paragrapheur pour Pascal et démontrent à notre avis l'intérêt de cette approche.

Les limitations de **Paradis** sont inhérentes aux principes sur lesquels il est fondé: il est impossible de prendre en compte des informations non purement syntaxiques. Il est par exemple exclu d'écrire en majuscules les identificateurs d'un type donné ou d'aligner les opérateurs ":=" dans une suite d'affectations.

Par ailleurs, certaines améliorations au fonctionnement du paragrapheur peuvent être envisagées: le rattachement des commentaires et leur placement n'est pas toujours satisfaisant; le paragraphage horizontal (sur une ligne) d'une liste trop longue conduit à une césure "aveugle" alors qu'une décompilation verticale (en colonne sur plusieurs lignes) serait quelquefois préférable.

En dépit de ces restrictions, quelques années d'utilisation ont montré que ce système est parfaitement viable. Ses atouts principaux, du point de vue de l'implanteur d'un paragrapheur, sont la facilité de conception et de réalisation ainsi qu'une maintenabilité et une évolutivité aisées et rapides. Enfin les paragrapheurs engendrés sont très sûrs et raisonnablement rapides, et pourraient même être intégrés sans beaucoup de modifications dans des outils (compilateurs, éditeurs syntaxiques) travaillant sur un arbre proche d'un arbre concret d'analyse syntaxique.

BIBLIOGRAPHIE

[BDJ 84] P. Boullier, Ph. Deschamp, M. Jourdan: **Application d'Outils de Haut Niveau à la Réalisation d'un Traducteur Automatique Pascal-Ada**; 2ème Colloque AFCET de Génie Logiciel, Nice, France, Juin 1984, pp. 375-385.

[Boullier 84] P. Boullier: **Contribution à la Construction Automatique d'Analyseurs Lexicographiques et Syntaxiques**; Thèse d'Etat, Université d'Orléans, Janvier 1984.

[DHKLL 75] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang et J.J. Lévy: **A Structure Oriented Program Editor: A first Step Towards Computer Assisted Programming**; International Computing Symposium 1975, Antibes, France. Egalement Rapport Laboria 114, IRIA, Avril 1975.

[DMR 82] Ph. Deschamp, M. Mazaud et R. Rakotozafy: **Perluette - Un Système de Métacompilation utilisant les Types Abstraits Algébriques**; Journées GROPLAN 1982, Bergerac, France. Egalement Rapport Gréco de Programmation 24, 1982.

[Lebon 81] E.R. Lebon: **Réalisation d'un Paragrapheur pour SYNTAX**; Rapport de DEA, Université Paris VI, Septembre 1981.

[Lebon 83] E.R. Lebon: **Réalisation d'un Décompilateur d'Arbre Abstrait Spécifié à l'aide de la Grammaire Paragraphée**; Thèse de 3ème cycle, Université Paris VI, Décembre 1983.

[Mazaud 85] M. Mazaud: **Un Traducteur de PL/1 vers C**; Rapport Technique INRIA, à paraître.

[Oppen 80] D.C. Oppen: **Prettyprinting**; ACM Transactions on Programming Languages and Systems, Vol. 2, No. 4, Octobre 1980, pp. 465-483.

[Rubin 83] L.F. Rubin: **Syntax-Directed Pretty Printing - A First Step towards a Syntax-Directed Editor**; IEEE Transactions on Software Engineering, Vol. SE-9, No. 2, Mars 1983, pp. 418-427.

[Vercoustre 83] A.M. Vercoustre: **Minerve: un méta-éditeur syntaxique**; Rapport INRIA 229, Juillet 1983.

<u>**Annexe A**</u>

Grammaire décrivant les grammaires acceptées par le constructeur de paragrapheurs, directives exclues

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<INTRO>                        = <RULE_LIST>                                ~~
                                 %DOL                                       ~~
                                 <ERROR_RECOVERY_T_LIST>                    ~~
                                 %DOL                                       ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ ;
<RULE_LIST>                    = <RULE_LIST>                                ~~
                                 <RULE>                                     ~~ ;
<RULE_LIST>                    = <RULE>                                     ~~ ;
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<RULE>                         = <VOCABULARY_LIST> %PV                      ~~ ;
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<VOCABULARY_LIST>              = <VOCABULARY_LIST>  <VOCABULARY>            ~~ ;
<VOCABULARY_LIST>              = %LHS_NON_TERMINAL "="                      ~~ ;
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<VOCABULARY>                   = %NON_TERMINAL                              ~~ ;
<VOCABULARY>                   = %TERMINAL                                  ~~ ;
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<ERROR_RECOVERY_T_LIST>        = <ERROR_RECOVERY_T_LIST>                    ~~
                                 <ERROR_RECOVERY_T>                         ~~ ;
<ERROR_RECOVERY_T_LIST>        =                                            ~~ ;
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<ERROR_RECOVERY_T>             = %TERMINAL                                  ~~ ;
<ERROR_RECOVERY_T>             = ";"                                        ~~ ;
<ERROR_RECOVERY_T>             = "<"                                        ~~ ;
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
*
*
* Error recovery token:
$
%PV
$
```

Annexe B

Spécification de l'analyseur lexical associé à la grammaire de l'annexe A

Classes

```
LAYOUT         = SP + HT + EOL ;
NOT_PRINTABLE  = #000..#010 + #013..#037 + #177 ;
PRINTABLE      = ANY - LAYOUT - NOT_PRINTABLE ;
QUOTE          = """" ;
T_HEADER       = PRINTABLE - "<" - "#" - ";" - QUOTE - "~" ;
```

Abbreviations

```
DIRECTIVE      = "~"&Is_Reset (1) {ANY} "~". ;
LINE           = {ANY} EOL. ;
COMMENT        = "*"&Is_First_Col LINE ;
T_BEGIN        = -"#" PRINTABLE | T_HEADER ;
STRING         = -QUOTE <PRINTABLE | -QUOTE. QUOTE> -QUOTE. ;
NT_BODY        = {PRINTABLE | SP} ;
```

Tokens

```
Comments       = -(<LAYOUT> | COMMENT | DIRECTIVE) ;
%LHS_NON_TERMINAL
               = "<"&1 NT_BODY ">". ;
%NON_TERMINAL  = "<" NT_BODY ">". ;
%TERMINAL      = T_BEGIN {PRINTABLE} | STRING ;
"="            = -"="&Is_Set (1) @Reset (1) ;
%PV            = -";" @1 @Set (1) ;
%DOL           = -("$"&Is_First_Col LINE) @Set (2) ;
";"            = ";"&Is_Set (2) {PRINTABLE} ;
"<"            = "<"&Is_Set (2) {PRINTABLE} ;
```

```
-- Variables:
--  1    : is set when waiting for a Left-Hand-Side non-terminal
--  2    : is set when analysing the error-recovery terminals list
--
-- Predicates:
--  &1   : => (&is_first_col and &is_set (1))
--
-- Actions:
--  @initialize:
--        erases the source text from beginning until the first "<"
--        laying at column 1, @set (1), @reset (2)
--  @1   : Skip the source text until the first "<" or "$"
--        laying at column 1
```

<u>**Annexe C**</u>

Grammaire décrivant les directives
acceptées par le constructeur de paragrapheurs

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<PP_SPEC>              =                                     ~~ ;
<PP_SPEC>              = INH                                 ~~ ;
<PP_SPEC>              = MARGIN <DELTA>                      ~~ ;
<PP_SPEC>              = SPACE  <FORWARD_MOVE_SPEC>          ~~ ;
<PP_SPEC>              = SPACE  <BACKWARD_MOVE_SPEC>         ~~ ;
<PP_SPEC>              = COL    <FORWARD_MOVE_SPEC>          ~~ ;
<PP_SPEC>              = SKIP   <FORWARD_MOVE_SPEC>          ~~ ;
<PP_SPEC>              = PAGE   <FORWARD_MOVE_SPEC>          ~~ ;
<PP_SPEC>              = TAB    <FORWARD_MOVE_SPEC>          ~~ ;
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<FORWARD_MOVE_SPEC>    =                                     ~~ ;
<FORWARD_MOVE_SPEC>    = ( %NUMBER )                         ~~ ;
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<BACKWARD_MOVE_SPEC>   = ( - %NUMBER )                       ~~ ;
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<DELTA>                = + %NUMBER                           ~~ ;
<DELTA>                = - %NUMBER                           ~~ ;
<DELTA>                =                                     ~~ ;
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

$
$
```

Analyseur lexical associé

Classes

```
    MOVE          = SP + HT + EOL + FF ;
```

Tokens

```
    Comments      = -<MOVE> ;
    %NUMBER       = <DIGIT> ;
    KEY_WORD      = <LOWER @Lower_To_Upper | UPPER> ;
```

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique