Annexe C

Un *pretty-printer*

Un *pretty-printer* prend en paramètre un programme sous la forme d'un arbre de syntaxe abstraite, et retourne une version bien composée du même programme sous la forme d'un texte mis en page selon des règles préétablies. Le programme résultat ne diffère du programme source que par la mise en page. Pour le travail proposé, le principal élément de mise en page est l'alignement vertical des commandes d'un même corps de programme, de boucle, ou de conditionnelle, et l'indentation des commandes selon leur degré d'imbrication dans un programme (c'est-à-dire le décalage par rapport à la marge de gauche à l'aide de caractères *espace*[1] insérés à dessein).

Ce sujet peut être traité en quatre séances dont une séance « papier ». Il ne comporte aucun enjeu formel lié à la sémantique des programmes, mais permet de se familiariser avec la manipulation d'un programme via son arbre de syntaxe abstraite. Dans la version présentée ici, ce sujet ne tient pas compte de la largeur de la page pour indenter le programme. Une variante qui en tiendrait compte est beaucoup plus difficile, mais très intéressante à concevoir et réaliser.

C.1 SPÉCIFICATION

On fixe le cahier des charges suivant :

- L'arbre de syntaxe abstraite passé en paramètre est supposé correct[2].

1. Plus communément appelés *blancs*, ces caractères sont rendus visibles par le symbole ␣ dans certains exemples.

2. Il peut en être autrement s'il résulte d'un calcul, par exemple d'une transformation de programme, réalisé par un programme incorrect.

- Le programme résultat ne comporte qu'une commande élémentaire (telle que `nop` et v `:=` e) par ligne.
- Le programme résultat ne comporte qu'une tête de commande composée (telle que `while` e `do`, `for` e `do`, et `if` e `then`) par ligne, et dans ce cas ne comporte que celle-ci. Le `od` de fermeture de corps de boucle et le `fi` de fermeture de conditionnelle sont composés seuls sur une ligne. Par conséquent, même la plus simple des boucles est composée sur au moins trois lignes :

```
while E do
    I
od
```

  Le `else` de séparation entre la branche *alors* et la branche *sinon* d'une conditionnelle est lui aussi composé seul sur une ligne.
- Les % qui délimitent les sections entrée, calcul et sortie des programmes sont composés seuls sur leur ligne. De même, les sections d'entrée et de sortie sont composées chacune sur une seule ligne quelle que soit leur longueur.
- Les expressions sont composées sur une seule ligne quelle que soit leur taille[3].
- Il y aura une valeur d'indentation par défaut, mais l'utilisateur devra pouvoir paramétrer le *pretty-printer* de façon à spécifier d'autres valeurs d'indentation.
- Les % et les sections d'entrée et de sortie ne subissent pas d'indentation.
- Le corps d'un programme, d'une boucle ou d'une branche de conditionnelle est indenté à droite par rapport au %, `while`, `do`, `if` ou `else` qui le commande.
- Les `while` et `od`, `for` et `od`, `if`, `else` et `fi` qui commandent les différentes parties d'une même commande subissent la même indentation.
- Le ';' qui construit les séquences de commandes est toujours placé en fin de ligne.
- Il y aura toujours un espace de part et d'autre de `:=` et de `=?`, avant les arguments de `=?`, `hd`, `tl` et `cons`, avant le ';' qui construit les séquences de commandes, avant le `do` et le `then`, et après le `read`, le `write`, le `while`, le `for` et le `if`.
- Les parenthèses n'introduisent pas de nouveaux espaces.

Par exemple, le programme source

```
read X % Y := nil;
while X do Y      := (cons(hd X)Y) ; X:=(tl
          X) od % write Y
```

dont l'arbre de syntaxe abstraite est

3. C'est la principale limitation du *pretty-printer* proposé. Gérer la mise en page des expressions serait donc le premier travail complémentaire à réaliser pour qui voudrait obtenir un *pretty-printer* plus réaliste.

```
(PROGR (list (VAR "X"))
       (list (SET (VAR "Y") NIL)
             (WHILE (VAR "X")
             (list (SET (VAR "Y") (CONS (HD (VAR "X")) (VAR "Y")))
                   (SET (VAR "X") (TL (VAR "X"))))) )
       (list (VAR "Y")))
```

est mis en page de la façon suivante si on fixe l'indentation du corps du programme à 2 espaces et l'indentation du corps de boucle WHILE à 6 espaces :

```
read X
%
  Y := nil ;
  while X do
        Y := (cons (hd X) Y) ;
        X := (tl X)
  od
%
write Y
```

Avec *espaces* visibles :

```
read␣X
%
␣␣Y␣:=␣nil␣;
␣␣while␣X␣do
␣␣␣␣␣␣␣␣Y␣:=␣(cons␣(hd␣X)␣Y)␣;
␣␣␣␣␣␣␣␣X␣:=␣(tl␣X)
␣␣od
%
write␣Y
```

Attention ! La présence ou l'absence de ' ; ' à la fin de chaque ligne du texte de programme produit ne vient pas du cahier des charges adopté. C'est la syntaxe concrète du langage WHILE qui spécifie où l'on place les ' ; ' (voir la grammaire 5.1 page 65). La règle

commande ⟶ commande ; commande

montre que le caractère ' ; ' ne peut se trouver qu'entre deux commandes. C'est pourquoi dans l'exemple précédent, il n'y a pas de ' ; ' après le do car `while X do` n'est pas une commande, ni avant le od car od n'est pas une commande. Il n'y en a pas non plus après le od car il n'est pas suivi d'une commande mais par une spécification de variables de sortie. On aurait pu en trouver un si une commande avait suivi la commande `while`.

Le principe de fonctionnement du *pretty-printer* est le suivant. Une liste de chaînes de caractères est d'abord produite selon le cahier des charges donné plus haut. Dans cette liste, chaque chaîne correspond exactement à une ligne de programme bien composé, et comporte déjà les espaces correspondant à son indentation. Chaque chaîne est produite indépendamment des autres et sans le saut de ligne final ni l'éventuel caractère ' ; '. C'est au moment de former la liste des chaînes que celles qui constituent une fin de commande et sont suivies par une autre commande seront augmentées d'un ' ; '. Dans une seconde étape, toutes ces chaînes sont concaténées et des sauts de ligne sont insérés de façon à produire une chaîne unique prête à être imprimée ou sauvegardée dans un fichier.

C'est en traitant les listes de commandes, qui forment par exemple le corps des boucles, que l'on sait déterminer si une commande est suivie d'une autre, et que l'on peut alors ajouter un caractère ' ; ' à la dernière des chaînes qui la représentent.

Chaque commande donne lieu à une liste d'une ou plusieurs chaînes. La liste de chaînes produite dans un premier temps est la concaténation de toutes ces listes. On veillera à distinguer les chaînes de caractères et les listes de chaînes de caractères, et à ne pas prendre l'une pour l'autre.

C.2 PLAN DE DÉVELOPPEMENT

On propose le plan de développement suivant. Il commence par quelques fonctions utilitaires, pour passer aux fonctions qui manipulent les expressions d'arbres de syntaxe abstraite en commençant par les expressions, puis les commandes, et enfin les programmes. D'autres fonctions peuvent être utiles selon les algorithmes choisis par le programmeur.

1. Définir une valeur d'indentation par défaut, `indent-default=1`.

```
;; default indentation
;; indent-default : int
```

2. Définir une fonction qui retourne une valeur d'indentation adaptée à un contexte.

   Un contexte est une chaîne parmi `"PROGR"`, `"WHILE"`, `"FOR"` et `"IF"`. Une spécification d'indentation est une liste de paires (`cons` *contexte valeur*). La spécification de la fonction est

```
;; searches a list of indentation specifications
;; indents-search : context-name * (list indent-spec) -> int
```

3. Définir une fonction qui prend en paramètre une valeur d'indentation et retourne une chaîne composée d'autant d'espaces que la valeur.

```
;; makes an indentation
;; make-indent : int -> str
```

4. Définir une fonction qui prend en paramètre une chaîne et une liste de chaînes, et concatène la chaîne *devant* chacune des chaînes de la liste. On pourra utiliser avec profit la fonction `string-append` de Scheme.

```
;; appends a string before every element of a list of strings
;; append-string-before-all : str * (list str) -> (list str)
```

5. Réaliser la même opération pour la concaténation *derrière*.

```
;; appends a string after every element of a list of strings
;; append-string-after-all : str * (list str) -> (list str)
```

6. Définir une fonction qui prend en paramètre l'arbre de syntaxe abstraite d'une expression et une spécification d'indentation (inutile dans un premier temps), et retourne une chaîne de caractères représentant la syntaxe concrète de l'expression. On pourra utiliser avec profit la fonction `string-append` de Scheme.

```
;; pretty-prints an expression
;; pretty-print-expr : expr * (list indent-spec) -> str
```

7. Définir une fonction qui prend en paramètre l'arbre de syntaxe abstraite d'une commande et une spécification d'indentation et retourne une liste de chaînes (même s'il n'y en a qu'une) représentant toutes les lignes de la syntaxe concrète de la commande.
   Dans la ou les chaînes produites, les espaces correspondant à l'indentation relative à la commande courante sont présents, mais pas ceux qui correspondent à des commandes emboîtantes. Par exemple, si une commande apparaît dans un programme emboîtée dans d'autres commandes, les contributions de ces commandes à l'indentation ne sont pas présentes. En revanche, si une commande est composée, elle comprend des commandes emboîtées dans son corps, et la fonction aura ajouté devant les lignes correspondantes la contribution de la commande courante à l'indentation des sous-commandes.
   Lorsqu'une commande est simple, `NOP` et `SET`, la chaîne qui la représente ne comporte pas de ' ; ' en fin de ligne. En effet, dans la syntaxe concrète, le ' ; ' est un séparateur qui est placé entre deux commandes, et pas un terminateur de commande[4]. Aussi, lorsqu'une commande simple est prise isolément, on ne peut décider de lui adjoindre un ' ; '. C'est le traitement des commandes composées, et particulièrement des listes de commandes, qui permet de décider si on place un caractère ' ; '.

4. Le caractère ' ; ' du langage WHILE ressemble donc davantage à la virgule du français écrit qu'à son point.

```
;; pretty-prints a command
;; pretty-print-command : comm * (list indent-spec) -> (list str)
```

8. Définir une fonction qui prend en paramètre une liste de commandes et une spécification d'indentation, et retourne une chaîne de caractères représentant toutes les lignes de la syntaxe concrète de la liste de commandes.
   C'est cette fonction qui place les '; ' en fin de ligne, seulement si une commande suit à la ligne d'après.

   ```
   ;; pretty-prints a list of commands
   ;; pretty-print-commands :
   ;; (list command) * (list indent-spec) -> (list str)
   ```

   Certains des tests de `pretty-print-command` sont placés ici, car ils font appel indirectement à `pretty-print-commands`.

9. Définir une fonction qui prend en paramètre une liste de variables et une spécification d'indentation et retourne une chaîne représentant la syntaxe concrète d'une liste de variables d'entrée d'un programme WHILE.

   ```
   ;; pretty prints an input list
   ;; pretty-print-in : (list var) * (list indent-spec) -> str
   ```

10. Définir la fonction duale pour les variables de sortie.

    ```
    ;; pretty-prints an output list
    ;; pretty-print-out : (list var) * (list indent-spec) -> str
    ```

11. Définir une fonction qui prend en paramètre un arbre de syntaxe abstraite de programme WHILE et une spécification d'indentation, et retourne une liste de chaînes représentant toutes les lignes de la syntaxe concrète du programme.

    ```
    ;; pretty-prints a while program
    ;; pretty-print-progr : progr * (list indent-spec) -> (list str)
    ```

12. Définir une fonction qui prend en paramètre un programme source du langage WHILE sous la forme de son arbre de syntaxe abstraite et une spécification d'indentation optionnelle, et retourne une chaîne unique représentant la syntaxe concrète de ce programme.
    Alors que dans les fonctions précédentes la notion de séquence de lignes était représentée par celle de liste de chaînes, cette fonction la représente par l'insertion de caractères de saut de ligne (`#\newline` en Scheme) dans une unique chaîne de caractères.

    ```
    ;; pretty-prints a while program
    ;; pretty-print : progr [* indent-spec ...] -> str
    ```