Traducteur “C-like” vers PostScript

Laurent Najman

8th March 2005

1 Le langage PostScript

Le langage PostScript est un langage de *description de page* développé par Adobe. Il est principalement destiné à permettre d'imprimer des documents sur des imprimantes lasers, mais on on peut aussi produire des dessins sur d'autres types de devices. Postscript est un langage interprété, basé sur une pile, de la même manière que les calculatrices en “notation polonaise inverse” : le programme pousse les arguments sur une pile et ensuite appelle l'opérateur.

De nombreux documents décrivant PostScript se trouvent sur le net. On retiendra plus particulièrement les trois suivants :
A first guide to PostScript
http://www.cs.indiana.edu/docproject/programming/postscript/postscript.html
PostScript as a Programming Language
http://adela.karlin.mff.cuni.cz/netkarl/prirucky/Flat/lang.html
Postscript Operators Alphabetic listing
http://atrey.karlin.mff.cuni.cz/ milanek/PostScript/Reference/REF.html

Sous Linux, un document PostScript se visualise par la commande `gv`.

1.1 La base

Un document PostScript commence toujours par

```
%!
```

et si vous voulez faire du PostScript encapsulé par

```
%!PS-Adobe-2.0 EPSF-2.0
%%Creator:
%%BoundingBox: 0 0 200 200
```

où la taille de la boite englobante est précisé. Pour ce qui nous intéresse dans ce projet, ça ne changera pas grand chose.

Pour tracer une ligne, il faut déclarer un nouveau chemin, se placer au point de départ, et tracer une ligne jusqu'au point d'arrivé. La ligne n'est pas tracé tout de suite, elle l'est quand on le demande.

```
newpath      %un nouveau chemin
1 1 moveto   %se déplacer
10 10 lineto %tirer une ligne
stroke       %tracer le dessin
```

On notera que les commentaires se font en plaçant un % à l'endroit où le commentaire commence, ce commentaire finissant en fin de ligne.

Pour écrire du texte, ça se passe comme ci-dessous

```
/Times-Roman findfont   % Obtenir les fontes de base
20 scalefont            % Mettre les fontes à 20 points
setfont                 % En faire la fonte courrante
newpath                 % Commencer un nouveau chemin
12 12 moveto            % Se déplacer
(Hello, world!) show    % Afficher "Hello, world!"
```

L'épaisseur du trait se change par la commande `setlinewidth`

```
1 setlinewidth
```

et la couleur du trait par

```
.7 setgray
```

ou encore (en couleur)

```
.7 .7 .7 setrgbcolor
```

Les 4 opérations sont `add`, `sub`, `mul` et `div`. Par exemple:

```
3 4 add
```

va placer 7 sur le dessus de la pile.

Les comparaisons sont `lt`, `le`, `eq`, `ne`, `ge`, `gt`.

1.2 La programmation PostScript

On déclare une variable comme suit

```
/x 100 def
```

Les variables ont une portée dans tout le document. Pour en limiter la portée, voir plus loin.

Un test se fait par la commande `if` comme en C. Si on veut faire un if/else, il faut utiliser `ifelse`

```
3 4 lt {(3 est plus petit que 4)} if % (3 <  4)
4 3 gt {(Vrai)} {(Faux)} ifelse % (Vrai) car 4 > 3
```

La boucle `for` n'a pas la même signification qu'en C, mais il existe la boucle `loop`. Par exemple

```
/x 0 def
{
    x 50 gt {exit} if % fin si x>50
    newpath
    0 0 moveto
    x 100 lineto stroke
    /x x 10 add def   % On ajoute 10 à x
} loop
```

Les sous programmes se définissent comme les variables. En général, on veut plus d'une commande dans le corps de la procédure, on les groupe par des accolades.

```
/proc {...} def
```

Par exemple, la procédure suivante dessine un cercle plein :

```
/filledcircle {
  gsave
  newpath
  0 0 1 0 360 arc
  fill
  grestore
}
def
```

Ce n'est pas très interessant tel quel, parce que la procédure ne prend pas d'argument. Pour qu'elle prenne des arguments, il faut aller les chercher sur la pile.

```
/proc {
/t exch def %Récupère la valeur au dessus de la pile
... %t est utilisable partout dans le document
} def
```

Par exemple, pour définir le rayon du cercle, on peut faire comme suit

```
/filledcircle {
  /radius exch def
  gsave
  newpath
  0 0 radius 0 360 arc
  fill
  grestore
}
def
```

La procédure s'appelle comme suit : `1 filledcircle`

Un problème que l'on a si on fait de la sorte est que la variable `radius` est globale à tout le document. Pour faire une variable locale, on se sert d'un dictionnaire.

```
/proc {
  1 dict begin %Crée un dictionnaire avec un élément
  /t exch def  %Récupère la valeur au dessus de la pile
  ... %t est utilisable localement ici
  end
} def
```

La procédure de cercle devient alors

```
/filledcircle {
  3 dict begin
  /radius exch def % rayon
  /y exch def      % centre x
  /x exch def      % centre y
```

```
  gsave
  newpath
  x y radius 0 360 arc
  fill
  grestore
  end
}
def
```

Remarquez que nous utilisons ici `gsave` et `grestore`. Cela permet de dessiner le cercle sans déranger les chemins qui peuvent être en train d'être créer.

2 Le projet: un traducteur de "C-like" vers PostScript

Comme vous pouvez vous en rendre compte, programmer en PostScript n'est pas immédiat pour quelqu'un habitué à programmer dans un langage comme C ou Java. Le projet consiste donc à faire un traducteur d'un mini-langage de type C vers PostScript.

Plus précisément, nous allons faire un langage capable de traduire en PostScript des programmes comme celui-là :

```
def dragon(n,x,y,z,t) {
  if (n == 1) {
    draw(x,y,z,t);
  } else {
    local u,v;
    u = (x+z+t-y)/2;
    v = (y+t-z+x)/2;
    dragon(n-1,x,y,u,v);
    dragon(n-1,z,t,u,v);
  }
}
dragon(10,20,20,220,220);
```

2.1 La commande `draw`

Commençons simplement : écrivez du code lex/yacc (et le C associé) pour pouvoir dessiner, c'est à dire capable de reconnaitre des nombres flottants simples (du style 1.543), et la commande `draw`, de telle sorte que les commandes du type `draw(10,10,20,20);` (ou une succession de commande de ce type) soient traduites en PostScipt.

2.2 Une compilation en deux phases

On pourrait faire une bonne partie du langage en utilisant uniquement des techniques comme celle que vous venez d'utiliser. Cependant, cela poserait des problèmes pour pouvoir définir des fonctions utilisant des variables (pouvez vous voir pourquoi ?)

On va donc utiliser une compilation en deux phases. Dans la première phase, on va se servir de yacc pour construire l'arbre du document, en répérant les erreurs de programmation. Dans une deuxième phase, on traduira cet arbre en PostScript.

Un noeud non terminal de l'abre est un opérateur du langage, une feuille de l'arbre est un des terminaux du langage. Au niveau où nous en sommes, un noeud non terminal ne peut être que `draw`, et une feuille ne peut être qu'un nombre.

- Un **noeud** devra donc contenir un `type` (ici réel ou opérateur), et une `union` de soit un réel soit un opérateur.

- Un **opérateur** est une structure contenant le type de l'opérateur (pour le moment, ce sera seulement DRAW, mais ça va vite changer), le nombre d'opérande(s) (ici, quatre, mais ça va vite changer) et un tableau ou une liste de noeuds correspondant aux opérandes de l'opérateur.

On aura également besoin de deux fonctions : l'une pour créer un noeud opérateur, et l'autre pour créer un noeud réel.

Note: pour ceux qui veulent progresser en C, vous pouvez éventuellement regarder les commandes `va_list` et `va_start`, qui permettent de créer des fonctions avec un nombre variable d'arguments. Les autres, faites comme vous voulez.

Il faudra également faire une fonction de traduction, qui prendra en arguement la racine de l'arbre, et traduira l'abre en PostScript.

N'oubliez pas de signaler ses erreurs à celui qui utilisera votre langage ! En particulier, il est de bon ton de signaler la ligne ou se produit l'erreur, et le caractère mis en cause.

2.3 Les variables

On va rajouter la possibilité d'utiliser des variables, pour pouvoir traduire des commandes comme

```
x = 3; // donner à x la valeur 3
x;     // mettre la valeur de x sur la pile
```

Il va falloir rajouter un nouveau type de feuille, le type identificateur.

2.4 Les expressions et les opérations basiques

Nous allons rajouter maintenant les opérateurs basiques de notre langage : +, - (dont le "moins unaire"), *, /, et les parenthèses. On aura également besoin des comparaisons (<, >, =<, >=, ==, !=) et c'est un bon moment pour les rajouter.

2.5 Les commentaires

On peut également ajouter les commentaires, comme en C et C++ :

```
// Ceci est un commentaire
/* Ceci est un commentaire
 sur plusieurs ligne
 qui finit ici */
```

Tant qu'à faire, il faut que les commentaires apparaissent dans le document PostScript généré.

2.6 Les boucles

Il existe plusieurs variantes des boucles, nous allons nous contenter de la boucle `for` du C, qui est de la forme `for( expr1 ; expr2; expr3) ... ;` où `expr1` est une expression evaluée en début de boucle, `expr2` est évaluée pour savoir si on a terminé la boucle et `expr3` est evaluée à chaque tour de boucle.

```
for(x=0; x<100; x = x + 10) {
  draw(0,0,100,x);
}
```

2.7 Les tests

Il s'agit de rajouter les tests `if` et `if ... else ...`.

2.8 Les fonctions : première version

Au niveau grammatical, les fonctions sont définies par la commande `def` suivi d'un identificateur, puis d'une liste d'identificateurs entre parenthèses. Elles sont appelées par l'identificateur correspondant à la fonction suivi d'une liste d'expressions entre parenthèses.

```
def untest(x,y) {
  draw(0,0,x,y);
}
untest(100,100);
```

2.9 Les fonctions : deuxième version

Si on se contente de la grammaire, on ne peut pas traduire les fonctions récursives en PostScript. Pour créer des fonctions récursives en PostScript, il faut utiliser des dictionnaires, comme on l'a vu plus haut. Pour pouvoir créer un dictionnaire, il faut savoir combien de variables sont entre les parenthèses de la définition de la fonction. Pour compter le nombre de variables, il va falloir descendre dans l'arbre.

Si vous faites cela, vous pouvez à présent traduire le code suivant.

```
def dragon(n,x,y,z,t) {
  if (n == 1) {
    draw(x,y,z,t);
  } else {
    dragon(n-1,x,y,(x+z+t-y)/2,(y+t-z+x)/2);
    dragon(n-1,z,t,(x+z+t-y)/2,(y+t-z+x)/2);
  }
}
dragon(10,20,20,220,220);
```

2.10 Un peu plus de vérification (facultatif, facile)

Rien dans notre impléméntation ne permet de vérifier que quelqu'un utilisant notre langage ne fait pas des erreurs du type suivant :

```
def func(x,y,z,t) {
  draw(x,y,z,t);
}
func(x,y,z);
```

Pour pouvoir vérifier cela, il faut construire une table des fonctions définies, et compter le nombre d'arguments avec lequel on appelle la fonction.

2.11 Les variables locales (facultatif, moins facile)

Pour pouvoir traduire le code donné en début de section, il faut rajouter la commande `local`.

La commande `local` définit une ou plusieurs variables comme locales au block dans lequel elles sont définies. Pour trouver le bloc, et donc savoir ou placer le dictionnaire, il faut remonter dans l'arbre.

2.12 Pour aller plus loin (facultatif, difficile)

Un document PostScript est optimisable de nombreuses manières. Documentez vous plus avant, et regardez comment vous pourriez intégrer quelques optimisations à votre langage. Cela peut éventuellement nécessiter une troisième phase de compilation.

3 Ce que vous devez rendre

J'attend de vous par email une archive en .tgz (tar compressé en gzip) contenant

- un fichier lex
- un fichier yacc
- les fichiers .h et .c que vous jugerez nécessaires
- un makefile permettant de générer l'exécutable
- des examples illustratifs

et un rapport décrivant votre projet, les difficultés rencontrées, les solutions envisagées, et celles que vous aurez mis en oeuvre. Le code devra tourner sous Linux. Une version papier du rapport devra également être déposée au secrétariat.

Vous serez noté sur la qualité du rapport et du code. Le projet devra se faire en binôme. Aucun point ne sera rajouté si vous faites le projet tout seul, par contre si vous le faites en trinôme, la note sera multipliée par 2/3.

Une recherche systématique des copies sera faite. Toute copie (même minime) vaudra la note de zéro aux élèves concernés, ceux qui pourront justifier qu'ils sont à l'origine du document copié n'auront pas d'autres sanctions que la note de zéro, les autres iront en plus en conseil de discipline.

Une soutenance orale sera demandée sur certains projets.