Programmation – Pré-DM2 : Le langage PostScript

{ Jeremie.Detrey, Emmanuel.Jeandel } @ens-lyon.fr

à rendre à votre TDman favori pendant le TD du 27/28 octobre

La semaine prochaine, vous aurez un DM qui consistera à programmer un interpréteur PostScript. Mais avant d'en arriver là, il serait bon de savoir à quoi ressemble ce langage. Ce pré-DM, que vous devrez rendre pour la semaine prochaine, devrait vous apprendre les rudiments du langage, afin de faciliter ensuite le travail de programmation.

Ce pré-DM est à faire tout seul, et à rendre durant la prochaine séance de TD, sous forme d'un joli rapport (de préférence rédigé avec LaTeX).

Introduction

Bien que souvent peu connu, le langage PostScript est pourtant fort répandu. Ce langage, inventé en 1985 par Chuck Geschke et John Warnock, est en effet présent dans beaucoup d'imprimantes laser. Contrairement à PDF ou à PCL, qui décrivent à quoi ressemblent les pages (il y a un texte ici, il y a une image là), le langage PostScript dit comment construire la page. C'est donc un véritable langage de programmation. PostScript a été créé à une époque où les communications entre l'ordinateur et l'imprimante étaient très lentes (par cable parallèle, par exemple) et où il valait donc mieux envoyer à l'imprimante une liste d'instructions à faire, fût-elle compliquée, plutôt qu'un résultat fini, souvent plus gros.

Nous allons donner ici quelques détails sur comment marche PostScript.

1 Plongeon dans le petit bassin[1]

1.1 Les langages de pile

Remarque : pour ceux qui connaissent le RPN (Reverse Polish Notation), tout ce qui est dit dans cette partie devrait leur apparaître comme trivial. Lisez-la tout de même en entier.

PostScript est un langage interprété basé sur le concept de pile. Toutes les instructions sont donc à comprendre comme des modificateurs de la pile courante. Par la suite, nous choisissons de représenter les piles de telle sorte que leur sommet soit à *droite* (et donc le fond à gauche).

Donnons un exemple simple :

```
2 4 4 mul dup 1 add 3 mul 1 add mul mul
```

Au départ la pile n'est pas forcément vide. On dénote ce qu'elle contenait par ....

– Lorsqu'on lit un entier, le comportement normal est de le mettre sur la pile. Après la lecture des trois premiers entiers, la pile contient donc :

```
... 2 4 4
```

[1] Attention la tête !

- La fonction `mul` enlève les deux premiers éléments de la pile, les multiplie ensemble, et met le résultat sur la pile :
  `... 2 16`
- La fonction `dup` enlève le premier élément de la pile, et en met deux copies sur la pile :
  `... 2 16 16`
- L'entier 1 se dépose sur la pile :
  `... 2 16 16 1`
- La fonction `add` enlève les deux premiers éléments de la pile, les additionne ensemble, et met le résultat sur la pile :
  `... 2 16 17`

1. Terminez l'exécution.

2 Piège en eaux troubles

Afin de ne pas être bloqué par la syntaxe de PostScript, on va tout de suite fixer des conventions pour nommer les objets :

- la *variable* (aussi appelé *nom*) `x` sera notée `N(x)` ;
- la *chaîne de caractères* `x` sera notée `S(x)` ;
- l'*entier* `x` sera noté `x` ;
- le *tableau* contenant $\texttt{x}_1, \ldots, \texttt{x}_n$ sera noté `<` $\texttt{x}_1$ `...` $\texttt{x}_n$ `>`.

Ainsi, l'exemple précédent :

```
2 4 4 mul dup 1 add 3 mul 1 add mul mul
```

s'écrira :

```
2 4 4 N(mul) N(dup) 1 N(add) 3 N(mul) 1 N(add) N(mul) N(mul)
```

2.1 Définitions, concept d'exécutables

En PostScript, tout objet peut avoir deux statuts : *exécutable* ou *non exécutable* (on dit aussi *littéral*). Le comportement de l'objet dépend de ce statut. En règle générale, la lecture d'un objet non exécutable revient à le poser sur la pile. Lorsque l'objet est exécutable, le comportement est différent, et dépend de quand l'objet est rencontré. S'il est rencontré lors de la lecture, on dit qu'il s'agit d'une exécution *immédiate*. Dans le cas contraire, il s'agit d'une exécution *différée*.

Là encore, souvent, l'exécution différée d'un objet revient à le placer en position de lecture. Ce ne sera pas le cas pour les tableaux qu'on verra plus tard.

Détaillons ici l'exécution de deux éléments clés :

- s'il s'agit d'un nom de variable, on regarde à quoi correspond la variable, et on fait une exécution différée du résultat obtenu ;
- s'il s'agit d'une chaîne de caractères, on fait lire son contenu.

Dans la suite, les objets exécutables seront reconnaissables grâce au symbole `*` qui les suit.

Par défaut, un nom est exécutable, sauf s'il est précédé par le caractère `/`, comme `/toto` par exemple.

Ainsi, finalement, l'exemple initial :

```
2 4 4 mul dup 1 add 3 mul 1 add mul mul
```

sera écrit :

```
2 4 4 N(mul)* N(dup)* 1 N(add)* 3 N(mul)* 1 N(add)* N(mul)* N(mul)*
```

Donnons maintenant d'autres exemples. On écrit en bas la pile, et en haut ce qu'il reste à lire.

2.1.1 Premier exemple

On part de :

```
/toto 1664 def toto 35
```

En version objet, cela devient :

```
N(toto) 1664 N(def)* N(toto)* 35
```

L'exécution donne alors :

- `N(toto)` correspond à un nom non exécutable, donc on l'empile :
  `1664 N(def)* N(toto)* 35`
  `... N(toto)`
- La lecture du nombre correspond aussi à un empilement :
  `N(def)* N(toto)* 35`
  `... N(toto) 1664`
- `N(def)` est un nom exécutable, donc on l'exécute. L'exécution de `N(def)` (qu'on ne va pas détailler précisément) revient à dire qu'on définit le nom `N(toto)` comme valant 1664. La pile est alors dans son état initial :
  `N(toto)* 35`
  `...`
- `N(toto)` est un nom exécutable. On regarde ce qu'il vaut, et on voit qu'il vaut 1664. On fait donc une exécution différée de 1664, ce qui revient à le placer en position de lecture :
  `1664 35`
  `...`
- Les deux derniers nombres s'empilent :
  [rien à lire]
  `... 1664 35`

2.1.2 D'autres exemples

On part de :

```
/toto /toto cvx def toto 35
```

En version objet, cela devient :

```
N(toto) N(toto) N(cvx)* N(def)* N(toto)* 35
```

- La lecture des deux premiers mots revient à les empiler :
  ```
  N(cvx)* N(def)* N(toto)* 35
  ... N(toto) N(toto)
  ```
- `cvx` est une instruction qui prend l'objet en sommet de la pile et le met exécutable :
  ```
  N(def)* N(toto)* 35
  ... N(toto) N(toto)*
  ```
- `def` définit maintenant le nom `toto` comme quelque chose valant `N(toto)*` :
  ```
  N(toto)* 35
  ...
  ```
- `N(toto)` est exécutable. On regarde ce qu'il vaut, et on voit qu'il vaut `N(toto)*`. On fait donc une exécution différée de `N(toto)*`, ce qui revient à le lire :
  ```
  N(toto)* 35
  ...
  ```
- On boucle indéfiniment.

2.1.3 Un exemple différent, mais pas trop

On prend :

```
/toto 3 def /titi /toto def toto titi
```

qui s'écrit :

```
N(toto) 3 N(def)* N(titi) N(toto) N(def)* N(toto)* N(titi)*
```

Si vous avez bien suivi, vous voyez qu'à la fin, la pile contient :

```
... 3 N(toto)
```

En revanche, si on écrit :

```
/toto 3 def /titi /toto cvx def toto titi
```

c'est-à-dire, en termes d'objets :

```
N(toto) 3 N(def)* N(titi) N(toto) N(cvx)* N(def)* N(toto)* N(titi)*
```

la pile contient à la fin :

```
... 3 3
```

2.1.4 Un dernier exemple, la fonction `exec`

Considérons maintenant le code suivant :

```
/toto 3 def /toto cvx exec /toto exec
```

qui s'écrit en termes d'objets :

```
N(toto) 3 N(def)* N(toto) N(cvx)* N(exec)* N(toto) N(exec)*
```

Lorsqu'on lit tous les premiers objets (jusqu'à `N(cvx)`), on définit `toto` comme valant 3 et on arrive à :

```
N(exec)* N(toto) N(exec)*
---------------------------
... N(toto)*
```

La fonction `exec` a un principe très simple : elle revient à faire une exécution différée de l'élément présent sur la pile. L'exécution différée d'un nom revient à le lire, ainsi, on arrive à :

```
N(toto)* N(toto) N(exec)*
---------------------------
...
```

Ce qui donne :

```
N(toto) N(exec)*
------------------
... 3
```

puis :

```
N(exec)*
----------
... 3 N(toto)
```

et :

```
N(toto)
---------
... 3
```

pour finir par :

```
----------
... 3 N(toto)
```

2.2 Tableaux

Un tableau correspond à une liste d'objets. Jusque là, pas de mystère. Une des vraies difficultés tient en revanche à son comportement lorsqu'il est exécutable : l'exécution immédiate d'un tableau exécutable revient à le poser sur la pile tandis que l'exécution différée revient à lire son contenu (les objets qu'il contient).

Supposons par exemple que nous soyons dans la situation suivante :

```
N(def)* N(a)* 52 N(b)* 32
---------------------------
... N(a) <N(b) 3 N(def)*>*
```

La lecture de `N(def)` définit donc `N(a)` comme égal à un certain tableau (exécutable) :

```
N(a)* 52 N(b)* 32
-------------------
...
```

`N(a)` est exécutable, donc on fait une exécution différée de son contenu, donc une exécution différée de `<N(b) 3 N(def)*>*`. Cela revient à lire le contenu du tableau, c'est-à-dire :

```
N(b) 3 N(def)* 52 N(b)* 32
----------------------------
...
```

1. Finissez l'exécution.

Contrairement aux autres exemples, on a fourbé ici, puisqu'il n'a pas été indiqué *comment* on construisait un tableau.

Il y a deux méthodes pour ceci. Dans la syntaxe PostScript, le bloc `{ a b c }` correspond à un tableau exécutable contenant `a`, `b` et `c`.

De sorte que le programme suivant, en syntaxe PostScript :

```
/a {/b 3 def} def a 52 b 32
```

est équivalent à :

```
N(a) <N(b) 3 N(def)*>*  N(def)* N(a)* 52 N(b)* 32
```

dans notre pseudo-langage.

La deuxième méthode consiste à utiliser les instructions `[` et `]`. `[` est un nom exécutable dont l'exécution revient à poser une marque (que l'on notera `-mark-`) sur la pile. `]` est un nom exécutable dont l'exécution cherche le dernier `-mark-` sur la pile, et met tout ce qu'il trouve entre les deux dans un tableau (non exécutable).

Par exemple, le code suivant :

```
[ 3 4 mul 8 ]
```

que l'on va traduire :

```
N([)* 3 4 N(mul)* 8 N(])*
```

va provoquer successivement :

```
N([)* 3 4 N(mul)* 8 N(])*
-------------------------
...

3 4 N(mul)* 8 N(])*
-------------------
... -mark-

3 4 N(mul)* 8 N(])*
-------------------
... -mark- 3
```

(... Pendant ce temps-là, à Vera-Cruz ...)

```
N(])*
-----
... -mark- 12 8


-------
... <12 8>
```

On remarque ainsi que le code PostScript suivant :

```
{ 3 4 mul 8 }
```

produit un tableau exécutable contenant 4 objets, alors que :

```
[ 3 4 mul 8 ]
```

est un tableau non exécutable contenant 2 objets.

Un dernier petit exemple :

```
/a {32 mul} def  52 a
```

qu'on écrit :

```
N(a) <32 N(mul)*>* N(def)* 52 N(a)*
```

Les trois premières instructions définissent `a` comme étant le tableau exécutable `<32 N(mul)*>` :

```
52 N(a)*
---
...
```

On empile ensuite 52 :

```
N(a)*
---
... 52
```

`N(a)` est un nom exécutable, donc on l'exécute. On regarde à quoi il correspond et on voit que c'est un tableau. On fait donc une exécution différée du tableau exécutable `<32 N(mul)*>`, ce qui revient à lire son contenu, c'est-à-dire `32` et `N(mul)*>` :

```
32 N(mul)*
---
... 52
```

Il n'est sans doute pas nécessaire de dire comment tout ça va finir...

2. Décrire le déroulement des codes suivants (en particulier le dernier) :

2.a. PostScript : `/a {def} def /b 32 a b 52 mul`
objets : `N(a) <N(def)*>* N(def)* N(b) 32 N(a)* N(b)* 52 N(mul)*`

2.b. PostScript : `/a [ /mul cvx ] cvx def 32 52 a`
objets : `N(a) N([)* N(mul) N(cvx)* N(])* N(cvx)* N(def)* 32 52 N(a)*`

2.c. PostScript : `/a {[} def /b {]} def /c a /mul cvx b cvx def 32 52 c`
objets :
```
N(a) <N([)*>* N(def)* N(b) <N(])*>* N(def)* N(c) N(a)* N(mul)
N(cvx)* N(b)* N(cvx)* N(def)* 32 52 N(c)*
```

2.3 Chaînes de caractères

Les chaînes de caractères sont représentées en PostScript entre parenthèses `(` et `)`. Ainsi la chaîne `S(Vive Caml)` s'écrit `(Vive Caml)` en PostScript. Un caractère est représenté par un entier, qui est son code ASCII (allez voir `http://www.asciitable.com/`).
Attention : mis à part les '\' (utiles pour protéger un caractère ou définir un caractère par son code ASCII en octal, par exemple '\151' qui équivaut à 'i') et les ')', une chaîne PostScript contient *exactement* tous les caractères situés entre les parenthèses.

Il faut aussi faire attention car en PostScript, une chaîne de caractères ne change jamais de taille. On ne peut donc pas simplement lui concaténer une autre chaîne sans refaire une allocation. De même, si on `dup` une chaîne de caractères et que l'on modifie la copie, l'originale sera aussi modifiée car il s'agit physiquement de la même chaîne (comme pour les tableaux en fait). Les fonctions de manipulation de chaînes sont décrites plus loin.

2.4 Erreurs

Evidemment, il peut y avoir des erreurs. Ainsi, si on demande à faire un `pop` alors que la pile est vide, `pop` va avoir un peu de mal. De même, faire `add` sur autre chose que des nombres n'est pas top. L'interpréteur PostScript de base a tendance à signaler ce comportement par une erreur et à arrêter l'exécution. Les imprimantes doivent sûrement faire différemment.

3 Le saut de l'ange

On va décrire ici plusieurs instructions PostScript. On utilisera pour cela la syntaxe suivante :

$$\mathrm{x}_n \ldots \mathrm{x}_1 \texttt{ instr } \mathrm{y}_1 \ldots \mathrm{y}_p$$

signifiant : l'instruction `instr` prend les n premiers éléments $\mathrm{x}_1 \ldots \mathrm{x}_n$ de la pile, fait quelque chose, puis pose sur la pile les éléments $\mathrm{y}_1 \ldots \mathrm{y}_p$ dans cet ordre (de sorte que l'élément en haut de la pile est y_p).

Instruction			Fonctionnement	Exemple	Résultat
$x\ y$	`add`	z	$z = x + y$	`1662 2 add`	`1664`
$x\ y$	`sub`	z	$z = x - y$	`1666 2 sub`	`1664`
$x\ y$	`mul`	z	$z = x * y$	`52 32 mul`	`1664`
$x\ y$	`eq`	b	b = true si $x = y$, false sinon	`1 2 eq`	`false`
$x\ y$	`lt`	b	b = true si $x < y$, false sinon	`1 2 lt`	`true`
$x\ y$	`gt`	b	b = true si $x > y$, false sinon	`1 2 gt`	`false`
x	`pop`	-		`1664 35 pop`	`1664`
x	`dup`	x x		`1664 dup`	`1664 1664`
x y	`exch`	y x			
var obj	`def`	-	voir plus haut	`/a 1664 def a`	`1664`
obj	`cvn`	obj	convertit obj en un nom	`(a) cvn`	`N(a)`
obj	`cvi`	obj	convertit obj en un entier	`(1664) cvi`	`1664`
obj	`cvx`	obj	rend obj exécutable		
obj	`cvlit`	obj	rend obj non exécutable		
obj	`exec`	-	exécution différée de obj	`{52 32 mul} exec`	`1664`
-	`[`	`-mark-`	ajoute une marque sur la pile	`[`	`-mark-`
`-mark-` $\mathrm{x}_1 \ldots \mathrm{x}_n$	`]`	<$\mathrm{x}_1 \ldots \mathrm{x}_n$>	crée un tableau	`[ 3 5 ]`	`<3 5>`
obj	`==`	-	affiche obj		
n	`string`	str	alloue une chaîne de caractères de taille n		
str i c	`put`	-	place le caractère c en position i dans str (attention, effet de bord !)	`( ) dup 0 105 put`	`S(i)`
$\mathrm{x}_n \ldots \mathrm{x}_0\ i$	`index`	$\mathrm{x}_n \ldots \mathrm{x}_0\ \mathrm{x}_i$		`1 6 64 2 index`	`1 6 64 1`
$\mathrm{x}_{n-1} \ldots \mathrm{x}_0\ n\ i$	`roll`	$\mathrm{x}_{i-1} \ldots \mathrm{x}_0\ \mathrm{x}_{n-1} \ldots \mathrm{x}_i$	rotation à droite de i places	`6 7 8 9 4 2 roll`	`8 9 6 7`
<$\mathrm{x}_1 \ldots \mathrm{x}_n$>	`length`	n		`[ 1 3 ] length`	`2`
obj	`xcheck`	b	b = true si obj est exécutable	`[ 1 ] xcheck` `{ 1 } xcheck`	`false` `true`

TAB. 1 – Commandes déjà vues ou pas loin. Les commandes situées à la fin ne sont pas à connaître.

Vous trouverez dans le tableau 1 les instructions déjà rencontrées (ou des variantes) et dans les tableaux suivants de nouvelles instructions.

3.1 Structures de contrôle

Le langage PostScript dispose évidemment de plusieurs structures de contrôle :

Instruction	Fonctionnement	Exemple	Résultat
b x `if` -	exécution différée de x si b est vrai	`true {32 52 mul} if`	`1664`
b x y `ifelse` -	exécution différée de x si b est vrai, de y sinon	`true 51 1664 ifelse`	`51`
s n e x `for` -	exécution différée, pour i allant de s à e par pas de n, de x avec i sur la pile.	`30 2 34 {2 mul} for`	`60 64 68`
tab x `forall` -	exécution différée, pour i parcourant les objets du tableau tab, de x avec i sur la pile ; tab peut aussi être une chaîne de caractères, taitée alors comme le tableau des codes ASCII des caractères.	`[ 10 17 ] {3 mul} forall` `0 (Toto) {add} forall`	`30 51` `422`
n x `repeat` -	exécution différée n fois de x	`3 /a /b 2 {pop} repeat`	`3`

TAB. 2 – Commandes pas déjà vues : structures de contrôles.

1. Que fait le code suivant :

```
/f { dup 2 lt { pop 1 } { dup 1 sub f mul } ifelse } def 4 f
```

2. Même question avec (on reprend la définition de `f`) :

```
[ 3 5 6 2 ] { f dup add } forall sub add exch sub
```

3. Sur le même principe, coder la fonction de Fibonacci.

3.2 Dictionnaires

Il serait temps d'aborder comment les définitions sont faites en PostScript. Un dictionnaire est un ensemble de couples $(\texttt{obj}, \texttt{val})$ signifiant : l'objet `obj` a pour valeur `val`.

La "mémoire" de l'interpréteur est constituée par un pile (encore une) de dictionnaires $d_1, \ldots, d_n$. On va plutôt dire "liste" que pile pour éviter les problèmes.

Lorsque l'on doit exécuter un nom `x` , l'interpréteur PostScript doit aller chercher successivement dans chaque dictionnaire si jamais il y a un couple $(\texttt{x}, \texttt{y})$ qui traîne. Ensuite, il fait une exécution différée de `y`.

Dans un sens, `/x y def` revient à ajouter au dictionnaire en tête de la liste un couple $(\texttt{x}, \texttt{y})$.

On trouvera dans le tableau 3 des instructions sur les dictionnaires.

Pourquoi plusieurs dictionnaires ? L'idée est de permettre de déclarer des variables locales.

Considérons par exemple le code suivant :

```
20 dict % création d'un nouveau dictionnaire
begin   % ce dictionnaire est ajouté à la mémoire
/a 3 def
a a add
% ...
% ...
end
```

Instruction			Fonctionnement	Exemple	Résultat
n	`dict`	y	crée un nouveau dico de taille n et le pose sur la pile		
d	`begin`	-	ajoute à la liste de dico le dico d		
-	`end`	-	enlève le dico en tête de la liste de dico		
-	`currentdict`	d	pose le dico en tête de la liste de dico sur la pile		
obj	`load`	val	renvoie la valeur de obj	`/a {3 mul} def /a load`	`{3 mul}`

TAB. 3 – Commandes nouvelles : dictionnaires (dico).

Cela permet en un sens de rendre la définition de `a` locale à ce qui se trouve entre le `begin` et le `end`. Après le `end`, le nom `a` reprend la valeur qu'il avait auparavant (s'il jamais en avait une).

Remarque : au passage, les commentaires PostScript commencent par un `%` et s'étendent jusqu'à la fin de la ligne.

4. Que fait le code suivant :

```
/f {
  1 dict begin
    /n exch def
    n 2 lt { 1 }
           { n 1 sub f n mul }
           ifelse
  end
} def
```

4 Brasse coulée

Les gens qui écrivent du PostScript à la main sont généralement des fous, et tant qu'à être fous, ils font des concours pour désigner les plus fous d'entre eux. Ainsi, l'*Obfuscated PostScript Contest*, où des gens encore plus fous[2] que les autres essayent de rendre leur code PostScript le plus court / astucieux / illisible possible (les trois critères sont importants !).

En voici un exemple soigneusement concocté par nos soins[3] :

```
/p/dup/m/exch/l/pop/0/sub/Y{repeat}def/h/eq/z/add/x/index/I/ifelse/s/def/o/roll
/k/gt/n/cvi/i/cvn 13{load def}Y/C{a}s/a(&mp0h'l1z .mp0h'l101a"%mp10m2x10Caml!I!I
){( )p 0 4 3 o put p(/)k{p(9)k{i}{n}I}{($)k{[}{]}I}I cvx}forall s
```

Pour vous éviter de retaper tout ça et d'y faire 15 fautes de frappe, allez chercher le fichier `obfuscated.ps` sur `http://perso.ens-lyon.fr/jeremie.detrey/04_prog/`.

[2] Tellement fous qu'il n'y a eu qu'une seule édition de leur concours. Ils ont surement du par la suite être internés en hôpital psychiatrique sur demande de leur famille.

[3] L'idée originale pour l'*obfuscation* est de Takashi Hayakawa, qui a remporté l'*Obfuscated PostScript Contest* en 1993, section meilleurs graphismes. Allez voir son code pour prendre peur (et admirer) :
`http://web.mit.edu/PostScript/obfuscated-1993/Tiny_RayTracing.ps`

Et maintenant, les questions tant redoutées :

1. Que fait cette horreur ?
2. Comment ça marche ?

5 Sous-marinier

1. Réfléchissez aux structures de données qui peuvent être nécessaires pour écrire un interpréteur PostScript en Caml. En particulier, la lecture de la documentation des modules `Hashtbl` et `Stack` peut s'avérer pertinente.

6 Terre ferme

L'interpréteur PostScript disponible sous Unix s'appelle Ghostscript (commande `gs`).

Pour essayer les exemples proposés plus haut, ou pour en concevoir des nouveaux, vous pouvez taper :

```
gs -dNODISPLAY
```

et vous amuser dans l'environnement, ou taper vos exemples dans un fichier `toto.ps` et faire :

```
gs -dNODISPLAY toto.ps
```

Le `-dNODISPLAY` est là pour que l'interpréteur ne vous affiche pas une fenêtre graphique toute vide.

7 Hum, vous n'oubliez pas quelque chose ?

Q : PostScript, à la base c'est quand même fait pour faire des graphismes, non ?

R : Certes. Pour ne pas surcharger ce pré-DM, on ne parlera des instructions graphiques de PostScript que la semaine prochaine.