Programmation – DM2 : Un interpréteur PostScript

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

à rendre pour la semaine du 22 novembre

*Ce DM est à faire en **binôme**. Comme vous êtes un nombre impair, nous accepterons un unique trinôme, à condition que vous vous mettiez d'accord et que vous veniez nous voir avant pour nous dire qui en fera partie. Mis-à-part pour ce groupe, nous refuserons tout devoir qui n'a pas été fait en binôme.*

Pour ce DM, vous devrez écrire un interpréteur PostScript, en OCaml. Votre code devra être bien indenté, lisible et soigneusement commenté. Appliquez-vous à faire du code propre et robuste (gérez les erreurs) plutôt que d'implanter les dix mille fonctions avancées de PostScript. Pensez aussi à nous fournir des exemples de PostScript pour tester vos extensions.

Enfin, vous devrez aussi nous rendre un rapport imprimé (rédigé de préférence avec LaTeX[1]) dans lequel vous expliquerez comment vous avez réalisé votre interpréteur, les choix de conception que vous avez faits, ainsi que les extensions que vous lui avez apportées.

1 Point de départ

Rongés de remords devant ce sujet de DM, les TDmen ont décidé de vous mâcher le boulot, et se sont attelés à écrire les parties rigolotes de l'interpréteur. Ainsi, vous n'aurez pas à vous soucier du *parsing* des fichiers PostScript, ni de la gestion de l'affichage et des routines graphiques.

Allez récupérer l'archive `psi.tar.gz` contenant une ébauche de programme à l'adresse habituelle `http://perso.ens-lyon.fr/jeremie.detrey/04_prog/`. Décompressez-la avec la commande `tar xzvf psi.tar.gz`, et regardez ce qu'elle contient.

1.1 Le `Makefile` bien pratique

Votre programme va être découpé en plusieurs modules, et qui dit modules dit compilation séparée, et donc `Makefile`. Pour vous éviter de vous emmêler les pinceaux avec les dépendances, contentez-vous de reprendre celui-ci et de l'éditer pour rajouter vos fichiers, cela devrait suffire.

1.2 Les définitions de types : `defs.mli`

Ce fichier contient des définitions pour les types principaux que vous allez devoir manipuler dans votre interpréteur.

Ainsi, le type `num` désigne une valeur numérique, pouvant être soit un entier (`Int`) soit un nombre réel (`Real`) réprésenté en virgule flottante.

Un objet PostScript `obj` est donc la combinaison d'un objet de type `obj'` et d'un champ booléen indiquant s'il est exécutable ou non. Dans la version actuelle, un objet peut être soit

[1] Ou directement en PostScript dans le texte, si vous êtes masochistes[2].

[2] Et encodez-le en base 85 si ça ne vous suffit pas.

un nombre (`Num`), soit une chaîne de caractères (`Str`), soit un nom de variable (`Name`), soit un tableau d'objets (`Array`).

Il vous faudra bien entendu enrichir cette définition pour pouvoir gérer d'autres types d'objets PostScript, comme par exemple les booléens ou les dictionnaires.

Enfin, on définit le type `input` comme un flot d'objets PostScript : ce type désigne la file de lecture.

1.3 Quelques définitions globales : `globals.ml`, `globals.mli`

Dans le module `Globals`, une paire de définitions utiles :

- Un type d'exception `Error` permettant d'uniformiser les erreurs levées par l'interpréteur. Vous pourrez bien-sûr enrichir cette définition, et utiliser plusieurs exceptions dans votre interpréteur.
- Une fonction `print_obj : Defs.obj -> unit` qui affiche un objet sur la sortie standard. Cet affichage reprend les conventions du pré-DM.

1.4 Le *lexer* / *parser* PostScript : `lexer.mll`, `lexer.mli`

Le fichier `lexer.mll` contient des règles décrivant la grammaire d'un fichier PostScript. Vous n'avez pas à vous en soucier. Les plus curieux d'entre vous peuvent cependant y jeter un œil, c'est assez instructif.

Un appel à `ocamllex` génère alors, d'après ces règles, le fichier `lexer.ml` (qui contient en fait un énorme automate) qui va effectivement lire des fichiers PostScript et les transformer en flots d'objets.

Pour résumer, le module `Lexer` nous définit donc :

- Une référence sur une fonction, `prompt : (unit -> string) ref`, qui vous permet de fixer le prompt de l'interpréteur, en mode interactif. Le prompt est une chaîne de caractères affichée après chaque retour à la ligne dans l'interpréteur. Il s'agit ici d'une fonction, car le prompt peut afficher des informations, comme par exemple le nombre d'objets actuellement sur la pile (pour reprendre les bonnes idées de `gs`). Par défaut, le prompt est une chaîne vide.
- Une référence sur un entier, `line`, qui contient le numéro de la dernière ligne lue. Il vous servira à indiquer plus précisément une erreur, par exemple.
- Les fonctions de lecture proprement dites : `input_from_file`, `input_from_stdin` ainsi que `input_from_string`, qui lisent du PostScript à partir respectivement d'un fichier, de l'entrée standard ou d'une chaîne de caractères, et qui renvoient le flot des objets ainsi lus.

1.5 L'affichage : `window.ml`, `window.mli`

Votre interpréteur PostScript devra gérer quelques instructions graphiques histoire d'avoir des résultats qui ont de la gueule. Pour cela, vous allez utiliser Cairo, une bibliothèque graphique qui offre des fonctionnalités proches de celles de PostScript. Allez voir sa documentation à l'adresse `http://cairographics.org/`, et plus précisément la documentation des modules OCaml correspondants sur `http://oandrieu.nerim.net/ocaml/cairo/doc/`.

Le module `Window` s'occupe de créer une fenêtre simple, ainsi que le contexte graphique nécessaire pour pouvoir appler Cairo. Il exporte donc uniquement :

– Le contexte graphique `cr` à passer à toutes les fonctions de Cairo.
– Une fonction `show : unit -> unit` servant à forcer la réactualisation de la fenêtre.

1.6 Une ébauche d'interpréteur : `psi.ml`

Le module `Psi`[3] est un exemple d'utilisation des autres modules :
– La fonction `do_input` lit le flot d'objets qui lui est passé et affiche sur l'entrée standard la liste des objets qu'il contenait.
– La fonction `draw_logo` est un exemple simple d'utilisation de Cairo et du module `Window`.
– Ces deux fonctions sont appelées dans le corps d'exécution principal du module. Remarquez la gestion des fichiers passés en ligne de commande à l'interpréteur, ainsi que la gestion des exceptions. Vous devrez vous en inspirer pour écrire votre propre interpréteur.

1.7 Le tout ensemble

Compilez le programme `psi` en tapant `make`, ou `make opt` pour obtenir le programme `psi.opt` compilé en langage machine. Tentez ensuite de l'exécuter. Par exemple :

```
./psi
```

ou encore :

```
./psi obfuscated.ps
```

Si vous n'êtes pas familier avec tout ça, amusez-vous un peu en modifiant `psi.ml` pour essayer de saisir l'articulation des différents modules.

[3] PSI pour *PostScript Interpreter*.

2 À vous maintenant !

Normalement, vous avez tout ce qu'il vous faut pour écrire votre interpréteur PostScript. Avant de vous lancer, allez télécharger le *PostScript Language Reference Manual*, dans lequel vous trouverez des informations utiles sur les différentes commandes PostScript existantes : `http://www.adobe.com/products/postscript/pdfs/PLRM.pdf`[4]. Cependant, faites attention : ce manuel couvre jusqu'aux dernières extensions du langage (level 3). Pour votre intepréteur, concentrez-vous en priorité sur les fonctions de base (level 1), vous aurez déjà largement à faire comme ça.

Voici une idée de progression que vous pouvez suivre (ou pas), avec les fonctions à implanter à chaque fois. Les trois premiers niveaux font partie du devoir, mais les deux suivants sont facultatifs, customisables et interchangeables.

2.1 Menu enfant

- gestion de la pile d'exécution ;
- dictionnaire basique ;
- `add`, `sub`, `mul` ;
- `eq`, `ne`, `lt`, `le`, `gt`, `ge` ;
- `true`, `false` ;
- `pop`, `dup`, `exch`, `def` ;
- `mark`, `[`, `]` ;
- `==`, `pstack` ;
- `if`, `ifelse`, `for`, `repeat`.

2.2 Formule express

- gestion de la liste de dictionnaires ;
- `dict`, `begin`, `end`, `currentdict`, `load` ;
- `string`, `get`, `put`, `length`, `forall` ;
- `cvn`, `cvi`, `cvr`, `cvx`, `cvlit`, `exec` ;
- `copy`, `index`, `roll`.

2.3 Formule gastronomique

- `translate`, `rotate`, `scale`, `setgray`, `setrgbcolor`, `setlinewidth` ;
- `newpath`, `closepath`, `moveto`, `rmoveto`, `lineto`, `rlineto`, `stroke`, `fill` ;
- `gsave`, `grestore`, `showpage` ;
- `loop`, `stopped`, `stop`, `quit`.

2.4 Fromage *et* dessert[5]

- `div`, `neg`, `abs`, `floor`, `ceil` ;
- `sqrt`, `exp`, `cos`, `sin`, `atan` ;

[4] Si vous non plus vous n'aimez pas le PDF, y'a la même chose avec juste le détail des instructions à l'adresse `http://atrey.karlin.mff.cuni.cz/ milanek/PostScript/Reference/`.

[5] C'est du bonus

- `and`, `or`, `not` ;
- `<<`, `>>` ;
- `sethsbcolor`, `curveto`, `rcurveto`, `clip`.

2.5 Double crème et pousse-café[6]

Vous avez quartier-libre pour implanter les fonctions que vous voulez. Typiquement, vous pouvez essayer de gérer le texte ou encore gérer l'exéuction immédiate des noms avec `//` (section 3.12.2 du PLRM). Vous pouvez aussi télécharger des exemples de PostScript, par exemple sur `http://web.mit.edu/PostScript/obfuscated-1993/`, et compléter votre interpréteur pour pouvoir les lire.

N'oubliez pas de nous joindre des exemples pour tester vos extensions.

3 Tester votre interpréteur

3.1 Jeux de tests fournis

Une archive de tests sous forme de fichiers PostScript sera maintenue à l'adresse habituelle. Vous pourrez ainsi vérifier sur ces exemples que votre interpréteur fonctionne bien, et donne bien le même résultat que `gs`.

3.2 Allez voir sur Internet

La communauté PostScript est assez discrète, mais propose de nombreux exemples tous plus tordus les uns que les autres (il y a même des jeux en PostScript, si si). Une rapide recherche sur Google vous permettra de trouver une tripotée de fichiers à tester.

3.3 PostScript élevé au grain

Nous vous encourageons fortement à écrire aussi vos propres fichiers de test, du moment qu'ils sont en PostScript correct (c'est-à-dire que `gs` les lit correctement). Vous verrez, c'est pas si dur !

[6] Ça aussi