

Projet : Compilation vers Postscript

1 Motivation

1.1 Contexte

Le langage Postscript est un langage de description d'objets graphiques, dans un sens large : Les objets sont des figures géométriques (cercles, polygone, courbes etc.), mais aussi du texte ou des images digitalisées. Postscript peut être compris comme le prédécesseur du format PDF d'Acrobat. Il est un langage qui peut être interprété directement par un grand nombre d'imprimantes, mais aussi être visualisé à l'écran.

Postscript permet de décrire les objets directement à l'aide d'une série de commandes élémentaires. Postscript est un véritable langage de programmation, basé sur un modèle d'exécution à pile. Le but de ce projet est d'écrire un compilateur d'un langage impératif similaire à C, désormais appelé CPS, vers Postscript. Ce projet englobe donc toutes les étapes d'un compilateur : analyse lexicale et syntaxique, vérification de types, génération de code.

Ce projet illustre aussi l'utilité de la création d'un DSL (*domain specific language*) : Postscript est un langage de haut niveau (si comparé avec un assembleur), mais pourtant assez illisible de premier abord. Vous verrez que les programmes que vous pourrez écrire en CPS sont plus compréhensibles qu'un programme Postscript. La plus grande lisibilité vient d'une syntaxe plus familière, mais aussi du fait que nous n'avons pas à traiter toute la complexité du langage Postscript.

1.2 Un premier exemple

Deux des figures fractales que vous pourrez générer avec des programmes CPS sont affichés en Figure 1.



FIGURE 1 – Figures fractales avec rectangles

Les procédures CPS qui génèrent ces deux images se trouvent en Figure 2. Il s'agit de deux fonctions récursives dont la première (**fig1a**) imbrique un carré à la même position que le carré précédent, en divisant la longueur de son côté par 2. La deuxième fonction (**fig2a**) est similaire, mais le nouveau carré est placé à l'angle supérieur droit du carré précédent. Le processus continue jusqu'à ce que niveau de profondeur d soit 0.

Ces deux fonctions utilisent une procédure **square** pour dessiner un carré, définie de manière évidente à l'aide d'une fonction pour dessiner des rectangles. C'est cette dernière fonction (Figure 3 qui construit

```

void square(float x, float y, float a) {
    rectangle(x, y, a, a);
}

void figla(int d, float x, float y, float a) {
    if (d != 0) {
        square(x, y, a);
        figla(d - 1, x, y, a /. 2.);
    }
}

void figlb(int d, float x, float y, float a) {
    if (d != 0) {
        square(x, y, a);
        figlb(d - 1, x +. a, y +. a, a /. 2.);
    }
}

```

FIGURE 2 – Code générant les images de Figure 1

un rectangle explicitement, en faisant appel à des primitives de Postscript. Le coin inférieur gauche se trouve aux coordonnées x , y , et les côtés ont la longueur a respectivement b . Le dessin se fait en commençant un nouveau tracé (**newpath**), en plaçant un “crayon” virtuel à (x, y) , en dessinant les quatre traits (**rlineto**), en fermant le tracé et en affichant la figure (**stroke**). Les procédures appelées par **rectangle** sont prédéfinies par Postscript, il ne sera donc pas possible de les définir, mais il faut les déclarer pour pouvoir les utiliser. Notez que notre langage CPS utilise quelques opérations arithmétiques comme en Caml non présentes en C (par exemple une addition de flottants $+$) et n’offre pas tout le confort d’un langage de haut niveau (par exemple, la négation unaire $-a$ doit être simulée par $0 - a$).

```

/* Declaration of native Postscript procedures / functions */
void newpath ();
void moveto(float x, float y);
void rlineto(float x, float y);
void closepath();
void stroke ();

/* Function definition */
void rectangle(float x, float y, float a, float b) {
    newpath();
    moveto(x, y);
    rlineto(a, 0.);
    rlineto(0., b);
    rlineto(0. -. a, 0.);
    rlineto(0., 0. -. b);
    closepath();
    stroke();
}

```

FIGURE 3 – Code CPS de **rectangle**

2 Travail demandé

Le but du projet est d’implanter toute la chaîne de traitement :

- analyse lexicale et syntaxique de CPS, voir § 5. Les étudiants de la DLMI sont dispensés de cette partie ; elle compte pour la note de Langages et Automates.
- typage, voir § 4
- génération de Postscript, voir § 7

Les trois phases sont largement indépendantes. Nous recommandons de commencer à implanter la vérification de types ; un parser vous sera fourni avec le projet. Vous pouvez ensuite réaliser votre propre analyse lexicale et syntaxique, et la génération de code.

Le travail doit être réalisé par des équipes de deux ou trois personnes. Le code doit être déposé sur Moodle ou doit être disponible sur une plateforme de gestion de code comme Github ou Bitbucket et fait l’objet d’une soutenance par les équipes.

Dates à retenir :

1. Le projet est à rendre le **mercredi 5 avril 2023 à 23h** au plus tard.
2. Une présentation par les équipes du projet aura lieu le **vendredi 14 avril 2023** dans l’après-midi. L’organisation de cette présentation se fera plus tard.

Le travail demandé est précisé dans les boîtes “Travail à réaliser” de ce document. Dans la plupart des cas, il s’agit d’écrire du code Caml dans les fichiers désignés. Parfois, il s’agit d’exercices d’évaluation ou de réflexion. Dans ce cas, mettez vos observations comme commentaire dans le fichier Caml ou dans le **README**.

Format des fichiers à rendre : Vous avez deux possibilités :

1. Vous pouvez déposer une archive (**tarfile** ou **zip**, mais pas de format **rar** ou autres formats propriétaires) sur Moodle, *avant la date limite*.
2. Pour vous inciter à travailler avec des systèmes de gestion de version, vous pouvez aussi nous communiquer les coordonnées d’un dépôt (seuls formats admis : **git** ou Mercurial) où nous pouvons récupérer votre code. L’adresse du dépôt doit être envoyée à martin.strecker@irit.fr avant la date limite, et le dépôt doit rester stable pendant au moins 24h après la date limite, pour nous permettre de récupérer le contenu. Évidemment, le dépôt doit être accessible en mode lecture. Si vous avez des doutes, envoyez l’adresse du dépôt au moins 24h avant la date limite pour permettre des tests. Nous nous engageons à évaluer le code uniquement après la date limite.

Le travail peut être effectué en équipes de trois personnes maximum. Un travail individuel est aussi possible. Si vous travaillez en équipe, votre **README** doit en indiquer les membres. Votre projet ne doit pas être la copie (partielle ou intégrale) du code d’une autre équipe.

Fichiers à rendre : Le code déposée devra contenir :

- Les fichiers source, qui doivent compiler (par un simple **make** à la console).
- un fichier **README** contenant une description courte du travail effectué, éventuellement des difficultés rencontrées, et aussi des tests que vous avez faits, éventuellement avec vos observations (par exemple sur la taille du code, son optimalité etc.).

Un projet qui ne compile pas et qui ne peut pas être exécuté sur des exemples (même modestes) sera d’emblée exclu de toute évaluation.

2.1 Fichiers fournis

L’archive fournie contient les fichiers suivants :

- **lexer.ml** et **parser.ml** : Le lexer/parser pour le langage source. Voir aussi § 5.

- **comp.ml** : le fichier principal, contenant la fonction **main** qui permet d'exécuter le code de la console, voir annexe A.1.
- **lang.ml**, qui contient les types du langage source. Vous avez le droit de modifier ces types, mais il n'est pas nécessaire de le faire, et si vous le faites, il faut documenter et motiver ces modifications.
- **instrs.ml** : le type de données des instructions du langage cible (syntaxe abstraite de Postscript).
- **interf.ml** : une interface pour interagir avec les fonctions en Caml, voir annexe A.1.
- **gen.ml** (actuellement presque vide) : le fichier qui devra contenir vos fonctions de génération de code, voir § 7.
- **typing.ml** (actuellement presque vide), contenant des fonctions de typage, voir § 4.
- **use.ml** : Permet de charger les fichiers compilés, dans l'interpréteur de Caml, voir annexe A.1.
- un **Makefile** pour compiler le projet. Il faut décommenter la partie marquée **TODO** quand vous faites l'intégration du lexer / parser. Du reste, ne modifiez pas ce fichier sauf si vous savez exactement ce que vous faites.
- dans le sous-répertoire **Tests**, le fichier **rectangles.c**, un exemple de programme que vous pourrez utiliser comme cas de test et qui génère les figures 1.

Souvent, les fichiers contiennent des définitions incomplètes, et qui existent uniquement pour permettre une compilation des sources Ocaml sans erreur.

3 Structures de données du langage source

La syntaxe abstraite (c'est à dire, les types de données) du langage source vous est fournie dans le fichier **lang.ml**. Vous avez le droit de la modifier, mais il faudra documenter les changements.

Expressions et commandes Dans notre présentation, nous nous limitons aux structures essentielles, à savoir les expressions

```
type expr =
  Const of value                (* constant *)
| VarE of vname                 (* variable *)
| BinOp of binop * expr * expr (* binary operation *)
| CondE of expr * expr * expr  (* conditional expr *)
| Calle of fname * (expr list) (* call expression *)
```

et les commandes

```
type com =
  Skip                (* no operation *)
| Exit                (* exit from loop *)
| Assign of vname * expr (* assign expression to var *)
| Seq of com * com    (* sequence of statements *)
| CondC of expr * com * com (* conditional com *)
| Loop of com          (* loop until exit *)
| CallC of fname * (expr list) (* call statement *)
| Return of expr        (* return from call *)
```

En général, une expression produit une valeur tandis qu'une commande modifie l'état du programme et est surtout exécuté pour ses effets de bord, par exemple l'affichage de graphiques.

Quelques éléments ont une version expression et commande, à savoir les conditionnels **CondE/CondC**, typiquement écrits comme **(cond ? t : e)** respectivement **if cond {...} else {...}** en C, et les appels de fonction respectivement procédure **Calle/CallC**. Nous n'avons que des opérations binaires (arithmétiques, booléennes et de comparaison, voir le type **binop**). Ceci requiert un codage légèrement maladroit par exemple de la négation arithmétique, par exemple **0 - 3** au lieu de l'opération unaire **-3**.

Vous êtes libre à modifier le parser et / ou les structures de données internes pour rajouter des opérateurs unaires.

L'affectation $x = e$ est représentée par le constructeur **Assign** et une séquence **c1; c2** de commandes par **Seq**. Une séquence de commandes vide, écrit **{}** en C, est représentée par **Skip**. Les boucles, par exemple un **while**, sont traitées par une combinaison de **Loop** et **Exit**, voir § 7.

Valeurs et types Nous avons des valeurs booléennes (**true/false**), des nombre flottants et entiers, des littéraux et des chaînes de caractère, voir le type **value**. Il y a aussi des types correspondants, et un type **VoidT** de plus pour des instructions, voir § 4.

Nous faisons la même distinction stricte entre entiers et flottants connue de Caml. Ainsi, tous les nombres qui s'écrivent sans point (par ex. **42** ou **2023**) sont des entiers, et avec point (par ex. **4.2** ou **2023.0**) des flottants. Les opérations de base $+$ $-$ $*$ $/$ sont pour les entiers, et aussi le modulo **%**; les variantes avec point $+$. $-$. $*$. $/$. sont pour les flottants. Pour les littéraux et les **string**, voir § 7.

Déclaration et définition de fonctions Un *programme* **prog** (voir par exemple la figure 3) est composé d'une liste de déclarations de fonction (typiquement de procédures ou fonctions prédéfinies de Postscript) et de définitions de fonction. Une *déclaration* de fonction **fundecl** est composée du type du résultat de la fonction, du nom de la fonction et une liste possiblement vide de paramètres (donc de déclarations de variables avec leur type). Une *définition* de fonction **fundefn** rajoute à cela le corps de la fonction, une commande.

```
type vardecl = Vardecl of tp * vname
```

```
type fundecl = Fundecl of tp * fname * (vardecl list)
```

```
type fundefn = Fundefn of fundecl * com
```

```
type prog = Prog of (fundecl list) * (fundefn list)
```

4 Typage

Les fonctions de typage seront à programmer dans le fichier **typing.ml**. Le but du typage est, globalement, d'assurer la cohérence d'opérateurs et fonctions avec leurs arguments.

Les types que nous utilisons sont :

```
type tp = BoolT | FloatT | IntT | LitT | StringT | VoidT
```

Dans un premier temps, vous pouvez ignorer les types **LitT** et **StringT**. Les **BoolT**, **FloatT**, **IntT** correspondent aux types habituels **bool**, **float**, **int**. Le type **VoidT** est un type (**void** en C) qui est le type de résultat des procédures, donc des fonctions qui ne renvoient pas de résultat, mais sont exécutées pour leurs effets de bord. Il n'y a pas de valeurs de type **VoidT** (voir le type **value**). Une variable ne peut pas avoir le type **void** (ce qui reviendrait à manipuler des valeurs **void**).

Travail à réaliser : Dans le fichier **typing.ml**, implantez des fonctions de vérification de type pour expressions et commandes **tp_expr** et **tp_cmd**. Vous avez évidemment le droit d'utiliser des fonctions auxiliaires.

La fonction **tp_expr** prend un environnement et une expression et renvoie son type. Pour des expressions simples, vous pouvez supposer que l'environnement est une liste d'association de noms de variables **vname** et de types. La fonction est alors assez similaire à la fonction pour le langage Caml déjà implantée en TP. Pour une expression conditionnelle **CondE**, dont la syntaxe concrète est **(c ? t : e)**, il faut que la condition **c** soit booléenne que les branches **t** et **e** aient le même type.

Le plus complexe est la vérification d'un appel de fonction **CallE**. Il faut vérifier le suivant :

- il faut que la fonction ait été déclarée ou définie ;
- les types des arguments (de l'appel) doivent correspondre aux types des paramètres (dans la définition de la fonction) ; surtout, il faut avoir autant d'arguments que de paramètres.

Le type de l'appel de fonction est alors le type de retour de la fonction.

Pour vérifier un appel de fonction, il ne suffit donc pas d'avoir un environnement qui enregistre uniquement le type des variables, il faut aussi connaître le type des fonctions. Pour cela, nous proposons d'utiliser le type suivant d'environnement :

```
type environment =  
  {localvars: (vname * tp) list;  
   funbind: fundecl list  
  }
```

dont le premier composant **localvars** correspond à la liste d'association utilisée précédemment comme environnement, et le deuxième composant **funbind** est pour la vérification des appels de fonction.

La fonction **tp.cmd** renvoie **VoidT** pour la plupart des commandes, mais doit pourtant vérifier que les sous-commandes soient bien typées. Le type d'un **Return e** est le type de l'expression **e** (donc normalement autre que **VoidT**), et le résultat d'une séquence **c1; c2** est le type de **c2**, et **c1** doit être de type **VoidT**. Le typage de **CondC** est similaire au typage de **CondE**.

Pour illustration, considérez le code :

```
int incorrect(int n) {  
  if (n > 0)  
    return(n + 1);  
  else  
    arc(3.0, 4.2, 5.0, 200, 300);  
}
```

Dans sa branche *then*, il renvoie une valeur (en conformité avec le type de la fonction), mais dans la branche *else*, il exécute un appel de procédure (**arc**) qui ne renvoie pas de valeur. Cette branche est donc mal typée mais devient bien typée si on rajoute un *return* :

```
else {  
  arc(3.0, 4.2, 5.0, 200, 300);  
  return(0);  
}
```

Travail à réaliser : Dans le fichier **typing.ml**, écrivez la fonction **tp.fundefn** qui prend un argument de type **fundefn**, effectue les vérifications mentionnées, et ensuite construit un environnement initial et vérifie la commande (avec **tp.stmt**) qui constitue le corps de la fonction.

Il reste à mettre en oeuvre la vérification de fonctions et de programmes. Une seule **Fundefn** est vérifiée en construisant un environnement initial (comment ?) et en vérifiant le corps de la fonction (une commande). A noter qu'une déclaration d'une fonction n'est pas vérifiée, mais la déclaration de la fonction est rajoutée à l'environnement initial. Un programme est correct si toutes ses fonctions sont correctes.

A côté de la vérification de types proprement dite, il faut aussi vérifier d'autres règles de bonne formation souhaitable, par exemple qu'il n'y a pas de double définition d'une fonction ou que les noms des paramètres d'une fonction sont disjoints. Votre implantation peut gagner en compréhensibilité si vos messages d'erreur permettent facilement d'identifier une erreur de typage.

5 Analyse lexicale et syntaxique

Ce travail est à réaliser dans le cadre du TP Langages et Automates. Nous recommandons d'attendre la présentation complète des outils **ocamllex** et **ocamlyacc** et de leur interaction avant de commencer ce travail. Entre-temps, une analyse syntaxique complète vous sera fournie.

Travail à réaliser : Écrivez un lexer (Fichier **lexer.mll**) et un parser (fichier **parser.mly**) pour le langage C.

Pour la syntaxe, nous nous inspirons du langage C, qui a été normalisé par l'ISO et est décrit dans un standard très volumineux¹. L'annexe A du standard donne la syntaxe complète que nous n'allons évidemment pas reproduire en intégralité, mais vous pouvez vous en inspirer. Votre syntaxe peut être légèrement plus restrictive que la syntaxe de C ; par exemple, vous pouvez imposer que l'instruction qui suit un **if** soit toujours inclus dans des accolades `{ ... }`, contrairement à la pratique courante d'écrire une seule commande sans accolades. Par contre, la syntaxe ne devrait pas obliger à des contorsions si on peut les éviter facilement. Par exemple, il devrait être possible d'écrire un **if** sans **else** (comment ?).

Vous avez le droit de modifier les structures de données des expressions et commandes (**expr** et **com**), mais il faut documenter les modifications.

Quelques remarques spécifiques :

- L'analyse d'expressions binaires devrait respecter les priorités et associativités habituelles de C. Ainsi, les opérations multiplicatives sont prioritaires sur les opérations additives, qui sont prioritaires sur les opérateurs de comparaison et les opérateurs booléens. Ainsi, $2+3 < 5 \&\& 3*x+4 \geq 5$ est parenthésée comme suit : $((2+3) < 5) \&\& ((3*x+4) \geq 5)$.
- Le prototype d'une boucle s'écrit **while e { c }**. Cette boucle est à traduire (lors de l'analyse syntaxique) en une combinaison des constructeurs **Loop**, **CondC** et **Exit**.
- Une séquence de plusieurs actions doit être décomposée en actions binaires (constructeur **Seq**).

Inspirez-vous des exemples en Figure 1 et 2.

6 Structures de données du langage cible

La génération de code a pour but de créer des instructions du langage Postscript. Ces instructions ne sont pas écrites directement dans un fichier, mais une bonne pratique est de générer des instances d'un type de données d'un langage cible, qui est dans notre cas le type suivant, voir fichier **instrs.ml** :

```
type instr =  
  IVal of value  
| IVar of int  
| IOper of string  
| IBloc of instr  
| ISeq of instr list  
| ILoop of instr  
| IDef of fname * instr
```

Le constructeur **IVal** correspond à **Const** des **expr**, et **IVar** à **VarE** ; par contre, les noms de variables sont traduits en indices (entiers). Les opérations de **BinOp** deviennent des opérateurs **IOper** (traduits directement en chaînes de caractères). Les blocs d'instructions **IBloc** sont surtout utilisés pour délimiter les branches d'un conditionnel. Les séquences de commandes **Seq** deviennent ici **ISeq**, des listes d'instructions. Enfin, **ILoop** désigne le constructeur spécifique pour boucles, et **IDef** le mécanisme de définition d'une fonction de Postscript.

1. https://web.archive.org/web/20181230041359if_/http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf

Écrivez, dans le fichier `instrs.ml`, la fonction `string_of_instr` qui affiche les instructions du type `instr`.

Pour en savoir plus sur la syntaxe du langage Postscript, référez-vous aux documents indiqués dans l'annexe [B.1](#).

7 Génération de Postscript

Le langage Postscript exécute des commandes en manipulant des valeurs qui sont stockés sur une pile. Par exemple, pour calculer $(2 + 5) - 1$, on empile d'abord les arguments 1 et 5 de l'addition, appelle l'opération d'addition, ensuite, on empile 1 et appelle l'opération de soustraction. La séquence d'opérations de ce calcul, en syntaxe concrète de Postscript, sera donc `2 5 add 1 sub`. Dans notre syntaxe abstraite, le code est `ISeq [IVal (IntV 2); IVal (IntV 5); IOper "add"; IVal (IntV 1); IOper "sub"]`.

Vous pouvez directement exécuter les commandes dans l'interpréteur *ghostscript* de Postscript et afficher l'état de la pile, voir une trace d'exécution dans l'annexe [B.2](#). Par ce mécanisme, vous pourrez facilement tester votre génération de code.

La démarche est la même pour des appels de fonctions (ou procédures), que ce soient des fonctions prédéfinies comme `arc` (voir les *déclarations* dans un fichier CPS) ou des fonctions que vous venez de définir (les *définitions* dans un fichier CPS).

Par exemple, `arc` permet de tracer un arc avec centre (x, y) , rayon x et un angle entre a et b :

```
void arc(float x, float y, float r, int a, int b);
```

En CPS, un appel de cette fonction est : `arc(50., 60., 30., 30, 120)`. En Postscript, on place les arguments sur la pile ; ensuite, on appelle la fonction : `50. 60. 30. 30 120 arc`.

Nous faisons une distinction entre *fonction* et *procédure*. Au niveau du langage source, une *fonction* a un type de résultat non `void`, et une *procédure* a un type `void`. Cette distinction est aussi très nette pour le langage Postscript, bien qu'il n'y ait pas de mécanismes comme le typage pour distinguer les deux formellement : une fonction remplace ses arguments par un résultat sur la pile ; une procédure a un effet de bord. Ainsi, `add` est une fonction (2 et 5 sont remplacés par 7), tandis que `arc` est une procédure (les arguments sont supprimés de la pile, avec l'effet de tracer un arc).

Travail à réaliser : Dans le fichier `gen.ml`, écrivez la fonction `gen_expr` qui génère une `instr` du langage cible à partir d'une `expr` du langage source.

A Utilisation de Caml

Voici quelques indications sur des éléments les plus utilisés. Pour des questions plus pointues, référez-vous au manuel de Caml disponible sur Moodle.

A.1 Compilation de fichiers

En Caml, vous avez le choix entre

- des fichiers source `*ml`
- des fichiers compilés / objet `*cmo`

Pour lancer votre code à partir de la console, il est impératif de le compiler. Pour le développement de votre code, il est préférable d'interagir avec l'interpréteur de Caml et de travailler avec les fichiers `*ml`. Vous pouvez facilement compiler tous les fichiers avec un `make` dans la console. Dans l'évaluateur Caml,

- pour charger un fichier `f` compilé, lancez `#load "f.cmo";;`²
- pour charger un fichier `f` source, lancez `#use "f.ml";;`

Problèmes typiques :

- Modules : après compilation et chargement avec `#load "nom de fichier.cmo";;`, les fonctions sont définies dans un module. Vous pouvez ou bien les appeler avec leur nom complet, par exemple `Typing.tp_expr`, ou bien ouvrir le fichier pour omettre le préfixe : `open Typing;;` et ensuite appeler `tp_expr`.
- En cas de blocage complet : Sortez de l'interpréteur interactif de Caml (avec `Ctrl-d`). Recompilez tous les fichiers avec `make`. Évaluez de nouveau le fichier `use.ml`.

Le fichier `use.ml` regroupe tous les `#load` à effectuer. Pour démarrer avec un état bien défini, vous pourrez lancer Caml et évaluer `#use "use.ml";;`

Utilisation du parser Pour développer votre code progressivement et tester vos fonctions de typage et de génération de code, il est utile d'utiliser le parser en mode interactif, pour récupérer l'arbre syntaxique correspondant au code que vous avez écrit dans un fichier, par exemple `even.c`.

Pour utiliser le parser, procédez comme suit :

1. Assurez vous que les fichiers Caml sont correctement compilés, avec un `make`.
2. Lancez une session interactive de Caml et évaluez le fichier `use.ml`, avec `#use "use.ml";;`
3. Pour avoir accès au parser : `open Interf;;`
4. Ensuite, lancez `parse "Tests/rectangles.c" ;;` (ou choisissez le nom de fichier approprié)
5. Pour éviter le préfixe `Lang` devant les constructeurs : `open Lang;;`

Aussi la génération de code, du graphe de flot de contrôle etc. peuvent se faire à partir de la session interactive.

A.2 Chaînes de caractères

- Les chaînes de caractères peuvent être concaténées avec `^` (accent circonflexe).
- Pour convertir un nombre en chaîne de caractères, utiliser `string_of_int` respectivement `string_of_float`.

A.3 Fichiers

Pour écrire un fichier Postscript, nous conseillons de construire une chaîne de caractères comme décrit en § A.2 et ensuite écrire cette chaîne sur fichier, par exemple en utilisant la fonction suivant :

```
let write_to_file str outfile =
  let outf = open_out outfile in
  output_string outf str ; flush outf ;;
```

Pour écrire `"foo"` sur le fichier `test.ps`, il faut évaluer

```
write_to_file "foo" "test.ps"
```

Attention, le contenu du fichier sera remplacé / écrasé complètement par la nouvelle chaîne.

2. Le dièse `#` fait partie de la commande

A.4 Exceptions

Une exception arrête immédiatement l'évaluation d'une expression de Caml. (Il existe aussi la possibilité de capturer une exception et continuer l'exécution, voir manuel de Caml).

Dans sa forme la plus simple, on lève une exception avec **failwith**, suivi d'une chaîne de caractères qui permet d'expliquer la cause. Par exemple,

```
failwith ("Variable undefined: " ^ v)
```

peut être utilisé lors de la vérification de types en cas d'une variable **v** indéfinie.

B Postscript

B.1 Documentation

Une introduction au langage Postscript est donnée sur Wipedia³. Le *cookbook*⁴ et le tutoriel⁵ décrivent le langage plus en détail. La définition complète (Postscript Language Reference Manual, *PLRM*) se trouve sur le site d'Adobe⁶.

B.2 Exécution d'instructions

On peut exécuter le langage Postscript interactivement, avec l'interpréteur Ghostscript (sous Linux : **gs**). Voici une trace d'interaction :

```
> gs
GS>2
GS<1>5
GS<2>pstack
5
2
GS<2>add
GS<1>pstack
7
GS<1>1
GS<2>sub
GS<1>pstack
6
```

Entrer une constante (2, 5) rajoute cette commande à la pile, dont le contenu peut être affiché avec **pstack** et la taille actuelle est indiquée entre **<.>**. Des opérateurs (par exemple **add**, **sub**) modifient la pile. Par exemple, **add** enlève les deux éléments au sommet de la pile et les remplace par leur somme. Une documentation exhaustive de toutes les commandes se trouve dans le manuel *PLRM*.

Bien évidemment, la plupart des commandes de Postscript ont pour but d'afficher des objets graphiques. La démarche est similaire : On positionne certains objets / valeurs sur la pile et fait ensuite appel à des opérations élémentaires qui manipulent la pile (comme les opérateurs arithmétiques) mais qui en plus ont comme effet de bord l'affichage des objets graphiques. Par exemple, pour dessiner un cercle :

1. Commencer avec un **newpath**
2. mettre les coordonnées du centre, ensuite le rayon,

3. <http://en.wikipedia.org/wiki/PostScript>

4. Charles Geschke : *Postscript Language : Tutorial and Cookbook*. Adobe Systems Inc., 1985.

5. Glenn Reid : *Thinking in Postscript*, 1990, <http://w3-o.cs.hm.edu/~ruckert/compiler/ThinkingInPostScript.pdf>

6. <https://www.adobe.com/jp/print/postscript/pdfs/PLRM.pdf>

3. ensuite dessiner le cercle (utiliser `0 360 arc`, on peut aussi dessiner un arc partiel).
4. Terminer avec `stroke`

```
> gs
GS>newpath
GS>100. 200. 50. 0 360
GS<5>pstack
360
0
50.0
200.0
100.0
GS<5>arc
GS>stroke
```

B.3 Affichage de fichiers

Visualiser un fichier Postscript (par exemple `foo.ps`) :

- Sous Unix / Linux : lancer `evince foo.ps &`
- Alternativement : conversion vers PDF : `ps2pdf foo.ps` et ensuite visualiser `foo.pdf`
- Sous Windows : Utiliser l'émulation `cygwin`

Fichiers Postscript :

- Simples fichiers en format texte (éditer, par exemple, avec Emacs)
- En première ligne : mettre un commentaire
`%!PS-Adobe-2.0`
- Pour afficher les objets dessinés, le fichier doit se terminer avec un `showpage`