Programmation Fonctionnelle
Projet – Analyse statique d'un mini-langage

novembre 2021

`Version 1 (10/11/2021)`

Introduction

Dans le cadre de ce projet, nous allons manipuler des programmes dans un langage informatique idéalisé nommé Polish. Ce langage maison est conçu pour être simple à comprendre, avec très peu de constructions fournies, et sa syntaxe est faite pour être simple à lire automatiquement, même sans outils complexes de "lexing" et "parsing". Pour cela, nous utiliserons en particulier une notation polonaise [1] des expressions (dite aussi notation préfixe), d'où le nom de ce langage.

Malgré sa minimalité, ce langage nous suffira déjà pour étudier certaines propriétés non triviales de programmes Polish, et se livrer en particulier à de l'*analyse statique*. Une analyse de code est dite statique si elle s'effectue sans exécuter réellement le programme étudié, par opposition à une analyse dynamique faite, elle, pendant une véritable exécution du code. Lors d'une analyse statique, certains éléments seront donc inconnus, comme les données saisies par l'utilisateur, mais via des techniques dites d'*interprétation abstraite* [2] on pourra néanmoins réaliser parfois des approximations intéressantes. Par exemple, on pourra se demander si un programme donné risque ou non d'échouer sur une division par zéro, et l'analyse pourra nous répondre "non, jamais" (programme sûr, aucune de ses exécutions ne divisera par zéro) ou "oui, parfois" ou encore "oui, toujours" (programme défectueux divisant toujours par zéro lors de toutes ses exécutions).

1 Le langage Polish

La syntaxe exacte du langage Polish – ce qu'on appelle sa syntaxe *concrète* – ainsi que sa représentation en OCaml – la syntaxe *abstraite* du langage – seront présentées aux sections 1.1 et 1.2. En préliminaire, voici un premier exemple de programme Polish :

1. Voir `https://fr.wikipedia.org/wiki/Notation_polonaise`.
2. Voir `https://fr.wikipedia.org/wiki/Interprétation_abstraite`.

```
COMMENT Calcul de la factorielle
READ n
i := 1
r := 1
WHILE i <= n
  r := * i r
  i := + i 1
PRINT r
```

Dans les grandes lignes, on peut déjà dire que Polish est un langage impératif, proposant :

- des **variables**, uniquement entières, auxquelles on peut librement affecter la valeur d'une expression (opérateur `:=`) ou une valeur saisie par l'utilisateur (mot-clef `READ`).
- des **expressions arithmétiques** formées de constantes entières, de noms de variables et des opérateurs arithmétiques. Ces expressions s'écrivent en notation préfixe, ce qui permet de se dispenser de parenthèses. Par exemple l'expression usuelle $(2 \times (3 + 4))$ s'écrit en Polish : `* 2 + 3 4`. La valeur d'une expression peut être affichée à l'écran via le mot-clef `PRINT`.
- des **boucles** `WHILE` – mais aussi des **conditionnelles** `IF`, associées à des `ELSE` optionnels. La notion de bloc pour l'une ou l'autre de ces structures de contrôle est basée sur l'indentation, comme en Python. Chaque ajout d'exactement 2 espaces avant le premier caractère d'une instruction augmente de 1 la profondeur de bloc. Les conditions des `IF` et `WHILE` sont des comparaisons entre deux expressions arithmétiques, par exemple : `+ x y <= 10`.

1.1 Syntaxe concrète du langage Polish

Ce que nous appellerons ici *fichier Polish* est est un fichier texte, que l'on recommande de nommer en utilisant l'extension finale `.p`. Le contenu de ce fichier, lorsqu'il est bien formé au sens des contraintes ci-dessous, est ce que nous appellerons un *programme Polish*.

Votre projet devra être capable de lire un fichier Polish et de transformer son programme en une représentation OCaml qui sera décrite à la section suivante 1.2. Cette lecture doit être implémentée uniquement en OCaml, sans utiliser d'outil externe (`ocamllex`, `ocamlyacc`, `menhir`...). Elle doit échouer dans tous les cas où le contenu du fichier ne respecte pas l'une ou l'autre des contraintes présentées ci-dessous : nous réaliserons des tests automatiques sur cette partie.

Lignes et mots. Nous appellerons *ligne* d'un programme Polish toute suite maximale de caractères successifs dans son fichier ne contenant pas le caractère `'\n'`, et *mot* d'une ligne toute suite maximale de caractères successifs ne contenant aucun espace `' '` et aucun caractère `'\n'`.

Le nombre d'espaces qui précèdent le premier mot d'une ligne est ce qu'on appelle *l'indentation* de cette ligne. L'indentation doit toujours être une valeur paire. Le nombre d'espaces séparant deux mots ou qui suivent le dernier mot n'est pas significatif, et peut être quelconque pourvu qu'il soit non nul.

Les mots d'un programme Polish se divisent en trois catégories : les *mots-clefs* du langage, les *opérateurs*, les *nombres entiers* et les *noms de variables*.

- les mots-clefs sont : `READ PRINT IF ELSE WHILE COMMENT`
- les opérateurs se divisent en trois catégories :
  - *opérateur d'affectation* : `:=`
  - *opérateurs arithmétiques* : `+ - * / %`
  - *opérateurs de comparaison* : `= <> < <= > >=`
- les nombres entiers sont tous les mots reconnus par la fonction OCaml `int_of_string`, par exemple `42` ou `-2`.
- tout autre mot est considéré comme un nom de variable.

Remarques. Les lignes d'un fichier Polish sont à lire une à une. La fonction OCaml standard `String.split_on_char`[3] pourra être utilisée pour découper une ligne en mots, en supprimant ensuite les mots vides que cette fonction peut produire si la ligne contient plusieurs blancs consécutifs.

Les caractères d'espacement autres que l'espace et `'\n'` (*e.g.* `'\t'`) ne sont pas considérés comme des séparateurs. Cette règle combinée aux précédentes permet a priori des noms de variables très exotiques (par exemple remplis de parenthèses et de tabulations), mais il est recommandé d'utiliser dans vos exemples des noms de variables plus raisonnables, par exemple commençant par une lettre suivie d'une répétition de lettres, chiffres ou `'_'`.

Expressions arithmétiques. Une *expression arithmétique* est une suite de un ou plusieurs mots de l'une ou l'autre des formes suivantes :

- un nombre,
- un nom de variable,
- l'un des opérateurs arithmétiques `+ - * / %`,
  suivi de deux expressions arithmétiques.

3. Voir `https://ocaml.org/api/String.html#VALsplit_on_char`.

Remarques. Dans la suite, les opérateurs d'une expression arithmétique seront interprétés de la manière habituelle, l'opérateur `/` correspondant à la division entière et `%` au modulo. Attention à bien séparer chaque mot (symbole, nombre ou nom) par au moins un espace.

Conditions. Une *condition* est une suite de mots de la forme suivante :

- une expression arithmétique,
  l'un des opérateurs de comparaison =, <>, <, <=, >, >=,
  une expression arithmétique.

Remarques. Contrairement aux opérateurs arithmétiques, les opérateurs de comparaison se placent entre les deux expressions qu'ils comparent (notation infixe habituelle).

Instructions et blocs. Une *instruction* est une suite de une ou plusieurs lignes consécutives d'un programme Polish de l'une ou l'autre des formes décrites ci-dessous.

La *profondeur* d'une ligne est définie comme son indentation divisée par 2 (rappelons que cette indentation doit être paire). La *profondeur* d'une instruction est celle de sa première ligne. Un *bloc* est une suite maximale d'instructions successives de même profondeur. Voici à présent les différentes formes d'instructions :

- Un **commentaire** est une ligne dont le premier mot est `COMMENT`.
- Une **lecture de variable** est une ligne réduite au mot `READ` suivi d'un nom de variable.
- Un **affichage** est une ligne dont le premier mot est `PRINT`, les mots suivants formant une expression arithmétique.
- Une **affectation** est une ligne dont le premier mot est un nom de variable, le second est l'opérateur `:=`, les mots suivants formant une expression arithmétique.
- Une **conditionnelle** de profondeur k est une suite de lignes de la forme suivante :
  - La première ligne a pour premier mot `IF`, les mots suivants formant une condition. Les lignes suivantes forment un bloc de profondeur $k+1$ appelé *bloc-if* de cette conditionnelle.
  - le bloc-if est, de façon facultative, suivi d'une ligne de profondeur k réduite au mot `ELSE`. Les lignes suivantes forment un bloc de profondeur $k+1$ appelé *bloc-else* de cette conditionnelle.
- Une **boucle** de profondeur k est une suite de lignes de la forme suivante :

— la première ligne a pour premier mot `WHILE`, les mots suivants formant une condition. Les lignes suivantes forment un bloc de profondeur $k+1$ appelé *bloc-while* de cette conditionnelle.

La structure d'un programme devra respecter la contrainte suivante : le programme tout entier doit former un bloc de profondeur nulle. Si le contenu d'un fichier lu ne respecte pas cette contrainte, vous devrez afficher un message d'erreur explicatif (donnant en particulier le numéro de la ligne où est située l'erreur), puis terminer votre programme avec un code d'erreur différent de 0 [4].

Remarques. Les contraintes sur les blocs font que la première ligne d'un programme doit être la première ligne d'une instruction de profondeur nulle, c'est-à-dire avoir une indentation de 0. La ligne qui suit immédiatement un `IF`, un `ELSE` ou un `WHILE` doit avoir une indentation d'exactement 2 de plus que cette ligne si le bloc-if, bloc-else ou bloc-while associé est non vide. Dans tout les autres cas, l'indentation d'une ligne doit être la même que celle de la ligne précédente (continuation du bloc courant) ou bien strictement moins (fin d'un ou plusieurs bloc(s) en cours, et retour à un ancien niveau d'indentation). La fin du fichier termine évidemment tous les blocs encore en cours.

1.2 La syntaxe abstraite de Polish

La syntaxe abstraite est ici la représentation d'un programme Polish comme une donnée OCaml prête à être manipulée (affichée, interprétée, analysée, etc.). Les types ci-dessous sont *imposés*. Ils ne pourront être modifiés que dans le cadre d'extensions à ce projet (typiquement via l'ajout de nouveaux constructeurs à `instr`, voir plus bas quelques suggestions d'extensions possibles, nous contacter si vous avez d'autres idées).

```
(** Numéro de ligne dans le fichier, débutant à 1 *)
type position = int

(** Nom de variable *)
type name = string

(** Opérateurs arithmétiques : + - * / % *)
type op = Add | Sub | Mul | Div | Mod

(** Expressions arithmétiques *)
```

4. Voir la fonction `exit : int -> 'a` décrite ici : `https://ocaml.org/api/Stdlib.html#VALexit`.

```
type expr =
  | Num of int
  | Var of name
  | Op of op * expr * expr

(** Opérateurs de comparaisons *)
type comp =
| Eq (* = *)
| Ne (* Not equal, <> *)
| Lt (* Less than, < *)
| Le (* Less or equal, <= *)
| Gt (* Greater than, > *)
| Ge (* Greater or equal, >= *)

(** Condition : comparaison entre deux expressions *)
type cond = expr * comp * expr

(** Instructions *)
type instr =
  | Set of name * expr
  | Read of name
  | Print of expr
  | If of cond * block * block
  | While of cond * block
and block = (position * instr) list

(** Un programme Polish est un bloc d'instructions *)
type program = block
```

Noter qu'il n'y a pas de représentation des commentaires, qui seront tout simplement ignorés. La représentation d'un bloc pourra être une liste vide, par exemple pour un `IF` sans `ELSE`, ou encore un `IF`, `WHILE` ou `ELSE` dont le bloc associé est vide. Dans la représentation d'un bloc, chaque représentation d'instruction est couplée avec sa position dans le fichier, c'est-à-dire le numéro de ligne où commence cette instruction, la première ligne ayant le numéro 1.

Conseils méthodologiques. Pour la lecture d'un fichier Polish et la transformation de son programme en donnée OCaml, il est recommandé de commencer par extraire toutes les lignes du fichier en couplant chaque ligne à son numéro de ligne, par exemple sous forme de `(int * string) list`. On pourra ensuite écrire des fonctions cherchant à lire dans cette liste de lignes une première instruction

(respectivement un premier bloc), et renvoyant un couple formé de la représentation de cette instruction (resp. de ce bloc) et des lignes restant à lire. La fonction de lecture d'instruction utilisera celle de lecture de bloc et vice-versa (récursivité mutuelle).

Pour le découpage d'une ligne en liste de mots, il a déjà été mentionné l'usage possible de `String.split_on_char`. Les mots vides de cette liste ne sont pas à conserver, mais permettent en début de liste de calculer l'indentation de la ligne. Il vous faudra en particulier pouvoir reconnaître dans cette liste le départ d'une expression arithmétique, et renvoyer sa représentation en même temps que la liste des mots restants après cette expression.

1.3 Évaluation d'un programme Polish

L'évaluation interactive d'un programme Polish – c'est-à-dire celle de sa représentation en OCaml par une valeur de type `program` – s'effectue sans surprise particulière pour un langage impératif aussi simple : les instructions du programme sont exécutées séquentiellement, et leur effet est celui intuitivement attendu.

Environnement. Chaque étape de l'évaluation d'un programme s'effectue dans un certain *environnement*, un mécanisme associant des noms de variables à une valeur entière. Cet environnement est susceptible d'être modifié par l'évaluation d'une instruction d'affectation ou de lecture : la fonction principale évaluant une instruction donnée devra donc permettre de récupérer cet environnement potentiellement modifié pour l'évaluation de l'instruction suivante.

Opérateurs. Le comportement des opérateurs arithmétiques et des comparaisons est identique à celui des opérateurs OCaml correspondants. Une division par 0 ou un modulo par 0 est une erreur qui doit terminer l'exécution de l'évaluation avec un message de votre choix et un code de retour différent de `0` (cf. `exit : int -> 'a`).

De façon facultative, au lieu du type `int`, vous pouvez choisir d'utiliser la bibliothèque `zarith` afin de faire toute l'évaluation des programmes Polish en utilisant le type des entiers arbitrairement grands fournis par `zarith`[5].

Expressions arithmétiques Lors de l'évaluation d'une expression arithmétique, les noms de variables seront remplacés par leurs valeurs associées dans l'environnement courant. La mention dans une expression d'une variable non encore associée à une valeur est une erreur qui doit terminer l'exécution de l'évaluation (toujours avec un message d'erreur et un code de retour non nul).

5. Voir `https://antoinemine.github.io/Zarith/doc/latest/Z.html`.

Affectation, lecture et affichage. L'évaluation d'un `Set(s, e)` commence par le calcul de la valeur de l'expression arithmétique `e`. Cette valeur devient celle associée au nom `s`.

L'évaluation d'un `Read s` commence par afficher sur la sortie standard le nom de variable `s` suivi d'un `?`, une valeur entière choisie est lue sur l'entrée standard du programme par un `read_int` ou équivalent. Cette valeur devient celle associée au nom `s`.

L'évaluation d'un `Print e` commence par le calcul de la valeur de l'expression arithmétique `e`, suivi de l'affichage de cette valeur sur la sortie standard du programme, suivi d'un saut de ligne.

Conditionnelles et Boucles. Un bloc-if (resp. bloc-else) n'est executé que si la condition du `If` est statisfaite (resp. non satisfaite), l'exécution d'un bloc-while est répétée tant que la condition du `While` est statisfaite.

Remarques. Il vous faudra faire le choix d'une structure OCaml permettant de représenter un environnement. Vous pouvez utiliser de simples listes d'association `(name * int) list` et vous servir de `List.assoc`, mais vous pouvez également définir des tables d'association plus efficaces via la ligne suivante :

```
module NameTable = Map.Make(String)
```

Ceci crée un nouveau type `'a NameTable.t`, celui des tables d'association indexées par des `string` – le paramètre `'a` est le type des valeurs associées à ces `string`. Les opérations disponibles sont alors `NameTable.empty`, `NameTable.add`, `NameTable.find`[6].

1.4 Analyse statique d'un programme Polish

Cette section décrit les analyses statiques de code Polish à réaliser pour le second rendu de projet. Pour plus de détails sur ce second rendu, voir la section 2.2. Pour le moment, les descriptions qui suivent sont volontairement succinctes, et pourront être précisées ultérieurement si nécessaire.

1.4.1 Simplifications de code

Lorsqu'on le lancera avec l'option `-simpl`, votre programme devra être en mesure d'effectuer les simplifications élémentaires d'un programme Polish décrites ci-dessous, puis d'afficher le code de ce programme après ces simplifications.

6. Pour plus de détails, voir `https://ocaml.org/api/Map.S.html`.

- **Propagation des constantes** : dans toutes les expressions arithmétiques apparaissant dans un programme, remplacer les opérations portant sur deux nombres par le résultat du calcul. Appliquer également les simplifications évidentes concernant les constantes `0` ou `1`, par exemple `+ 0 e` est à simplifier en `e`, de même pour `* e 1`, tandis que `* e 0` ou `/ 0 e` deviennent `0`. Propager ces simplifications autant que possible. Ces pré-calculs seront fait sur le type `int` OCaml avec les opérations usuelles d'OCaml, les débordements de capacité ("overflow") qui peuvent se produire sont acceptés ici. On notera que certaines sous-expressions peuvent disparaître lors de ces simplifications, donc un programme dont l'évaluation donnait une erreur due à une division par zéro pourra ne plus faire cette erreur après cette phase de simplification. Par contre les expressions `/ e 0` et `% e 0` sont à simplifier seulement en simplifiant `e` : ces expressions mènent à une erreur à l'évaluation, et ce comportement doit être conservé.
- **Élimination du code mort** : après propagation des constantes, il se peut que l'on obtienne des conditions comparant deux constantes. S'il s'agit de la condition d'un `IF`, remplacer alors ce `IF` par les instructions du bloc-if ou du bloc-else correspondant, selon le résultat de la comparaison. Pour le `WHILE`, supprimer entièrement ce `WHILE` si la condition est invalide. Par contre dans le cas où la condition est valide, on a alors une boucle infinie qu'il faut alors conserver faute d'équivalent plus simple en Polish.

1.4.2 Variables non-initialisées

Une analyse statique classique sur un programme est de vérifier si chaque accès à une variable se fait obligatoirement après une première écriture dans cette variable (soit via une affectation `:=` soit via un `READ`).

Lorsqu'on le lancera avec l'option `-vars`, votre programme devra afficher deux lignes :

- Sur la première ligne, le nom de toutes les variables du programme Polish étudié
- Sur la seconde ligne, le nom des variables pouvant être accédées avant leur première écriture.

Pour chacune des lignes, les noms de variables seront séparés par un espace, et viendront dans un ordre quelconque mais sans redondances.

Le calcul des variables non-initialisées est à faire via une sur-approximation : on ne cherche pas à calculer le résultat d'une condition de `IF` et de `WHILE` et on considère donc que tout `IF` peut aussi bien effectuer son bloc-if que son bloc-else, et que tout `WHILE` peut aussi bien effectuer son bloc-while (une ou plusieurs fois) que ne pas l'effectuer du tout. Ainsi, une variable est considérée comme initialisée

à la fin d'un `IF` si elle l'était déjà avant le `IF` ou bien si elle est initialisée durant les *deux* branches du `IF`. Et les variables initialisées après un `WHILE` sont les mêmes qu'avant ce `WHILE` !

Ces sur-approximations peuvent conduire à considérer une variable comme risquant d'être accédée non-initialisée, alors que les exécutions effectives du programme se dérouleront pourtant correctement. Par contre, si une variable n'est pas dans cette liste, on est certain qu'elle sera bien toujours initialisée avant d'être lue.

Conseil d'implémentation. De façon similaire aux "Map" suggérées dans la section 1.3, il est recommandé ici d'utiliser une structure efficace d'ensembles pour coder les ensembles de variables :

```
module Names = Set.Make(String)
```

Ceci crée un nouveau type `Names.t` des ensembles de chaînes de caractères (et donc en particulier de noms de variables), ainsi que des opérations telles que `Names.empty`, `Names.add`, `Names.union`[7].

1.4.3 Signes des variables

Lorsqu'on le lancera avec l'option `-sign`, votre programme devra afficher une ligne par variable du programme Polish étudié, avec sur chaque ligne le nom de la variable puis un espace puis un ou plusieurs caractères parmi `- 0 + !`. Ces caractères doivent indiquer les signes possibles de cette variable à la fin de l'exécution du programme, valeur nulle pour `0`, strictement positive pour `+`, strictement négative pour `-` ou encore erreur due à une division par zéro pour `!`. Une réponse à plusieurs caractères indique une variable pouvait prendre chacun de ces signes, par exemple `0+!` pour une variable positive ou nulle ou possiblement issue d'une division par zéro, ou encore `-0+` pour une variable à valeur entière quelconque mais non erronée (cas d'une variable après un `READ` par exemple).

Enfin une dernière ligne devra indiquer `safe` si le programme ne pourra jamais faire de division par zéro ou bien un message de la forme `divbyzero 42` si une division par zéro risque de se produire pour la première fois ligne 42.

Signes. Il est recommandé de définir un type tel que :

```
type sign = Neg | Zero | Pos | Error
```

7. Pour plus de détails, voir `https://ocaml.org/api/Set.S.html`.

On pourra ensuite associer à chaque variable Polish un `sign list`, ce qu'on appellera un environnement abstrait de variables. L'analyse demandée consiste à déterminer l'environnement abstrait des variables tout au long du programme Polish, et donc en particulier à la fin.

Expressions. Les opérations arithmétiques seront approchées au mieux, par exemple l'addition d'un `Pos` et d'un `Pos` donne un `Pos` par contre pour l'addition d'un `Pos` et d'un `Neg` on ne peut pas dire mieux que `[Neg;Zero;Pos]`.

Conditions. On se place dans la situation où une condition a été satisfaite. Déjà, en regardant les signes possibles des expressions qui ont été comparées, on peut vérifier s'il est effectivement possible de satisfaire cette condition, ou si au contraire on est dans une situation impossible. Ensuite, on pourra se demander si satisfaire cette condition nous apprend de nouvelles contraintes de signe sur les éventuelles variables présentes dans la condition. Par exemple, si un `x > 3` a été satisfait, on sait alors maintenant que `x` est strictement positif. Cela sera aussi le cas si un `x > y` est satisfait alors qu'on savait déjà que `y` a pour signes possibles `[Zero;Pos]`. On appellera cette opération la "propagation" de la condition.

Analyse des conditionnelles `IF`. Pour analyser un `IF`, on considère séparément son bloc-if et son bloc-else. Pour le bloc-if, l'environnement abstrait précédent le `IF` subit la propagation de la condition du `IF`, puis chacune des instructions de ce bloc-if (sauf si la condition était impossible). Pour le bloc-else, l'environnement abstrait précédent le `IF` subit la propagation de la condition inverse, puis chacune des instructions de ce bloc-else (sauf si la condition inverse était impossible). Enfin on réunifie les deux environnements obtenus dans les deux branches par une opération de "join" : si l'une des branches était impossible on ne garde que l'autre environnement, sinon on fait pour chaque variable l'union des signes possibles issus de ces deux branches.

Analyse des boucles `WHILE`. Pour analyser un `WHILE` on va procéder récursivement et faire ce qu'on appelle un "calcul de point-fixe". Appelons E_0 l'environnement avant le `WHILE`. On itère alors le processus suivant : sur le dernier environnement obtenu E_n, on propage la condition du `WHILE`, puis chacune des instructions du bloc-while, puis on "join" ce nouvel environnement et E_n pour obtenir E_{n+1}. On arrête ce processus lorsque $E_{n+1} = E_n$, ce qui est assuré de se produire ici vu que chaque variable ne peut avoir qu'un nombre fini de signes. Enfin, l'environnement à la sortie de ce `WHILE` est le dernier E_n calculé, sur lequel on effectue en plus la propagation de la condition inverse de celle du `WHILE`.

2 Travail à réaliser

Les rendus se feront impérativement en créant un "fork" du dépôt git fourni `pf5-projet`, en le rendant privé mais accessible aux enseignants du cours. Dans ce dépôt git fourni, voir en particulier le fichier `CONSIGNES.md` pour plus de détails sur les modalités de ce projet, ainsi qu'une aide sur git et GitLab dans le fichier `GIT.md`.

2.1 Rendu 1

D'ici au **15 décembre 2021**, vous devrez réaliser un programme pouvant :

- lire un fichier contenant un programme Polish et le traduire en syntaxe abstraite selon les types indiqués ci-dessus, puis, selon les cas :
- ou bien réafficher ce programme en partant de la syntaxe abstraite obtenue et suivant les règles indiquées de syntaxe concrète,
- ou bien évaluer ce programme.

Plus précisément votre programme devra pouvoir se lancer depuis un terminal, et accepter un premier argument sur la ligne de commande indiquant l'action à effectuer et un second argument indiquant le nom du fichier contenant le programme Polish à traiter. Pour ce premier rendu, le premier argument devra pouvoir être la chaîne de caractère `-reprint` ou bien `-eval`. Ainsi, si votre programme compilé s'appelle `polish`, un exemple de lancement sera par exemple `./polish -reprint fact.p` ou encore `./polish -eval fact.p`. Cette gestion des arguments pourra se faire soit en lisant directement le tableau `Sys.argv`[8], soit en utilisant des fonctionnalités plus avancées telles que `Arg.parse`[9].

Note : l'usage de l'outil `dune` pour compiler votre programme place le binaire compilé dans un endroit malcommode (dans un sous-répertoire de `_build`). Vous trouverez dans le code fourni un petit script nommé `run` pour aider le lancement de votre programme avec les bonnes options, par exemple `./run -reprint fact.p`.

Attention, votre programme devra afficher juste ce qui est demandé (le réaffichage du programme lu ou bien les affichages induits par l'évaluation du programme lu), afin de permettre des tests automatiques.

Extension possible (optionnelle) pour ce rendu 1 : utilisation de la bibliothèque `zarith` lors de l'évaluation pour permettre des calculs en précision arbitraire.

8. Voir `https://ocaml.org/api/Sys.html`.
9. Voir `https://ocaml.org/api/Arg.html`.

2.2 Rendu 2

Pour le **10 janvier 2022**, vous devrez compléter votre programme avec les fonctionnalités suivantes, en acceptant trois nouvelles options en premier argument de votre programme (et toujours un second argument donnant le nom du fichier Polish à trairer) :

- option `-simpl` : simplification d'un programme effectuant une propagation des constantes et l'élimination des blocs "morts".
- option `-vars` : calcul statique des variables risquant d'être accédées avant d'être écrites
- option `-sign` : analyse statique du signe possible des variables lors du déroulement du programme, et application à la détermination du risque de division par zéro.

Voir la section précédente 1.4 "Analyse statique" pour la description précise de ces fonctionnalités.

Extensions possibles (optionnelles) pour ce rendu 2 :

- Une analyse statique supplémentaire utilisant cette fois des intervalles au lieu de juste des signes. Attention à la convergence lors de l'analyse des `WHILE` (opération dite de "widening" : si une borne ne converge pas au bout de quelques tours d'analyse, l'approximer à $+\infty$ ou $-\infty$ selon les cas).
- Des enrichissements divers du langage : boucles `FOR` (et une option permettant leur traduction vers des `WHILE`s), conditions booléennes plus complètes (et leur traduction vers des suites de `IF`s), tableaux, fonctions, ...