projet de compilation

Petit Purescript

version 1 — 26 octobre 2023

L'objectif de ce projet est de réaliser un compilateur pour un fragment de Purescript [1], appelé Petit Purescript par la suite, produisant du code x86-64. Il s'agit d'un fragment 100% compatible avec Purescript, au sens où tout programme de Petit Purescript est aussi un programme Purescript correct. Ceci permettra notamment d'utiliser ce dernier comme référence. Le présent sujet décrit précisément Petit Purescript, ainsi que la nature du travail demandé.

1 Syntaxe

Dans la suite, nous utilisons les notations suivantes dans les grammaires :

$\langle règle \rangle^\star$	répétition de la règle $\langle règle \rangle$ un nombre quelconque de fois (y compris aucune)
$\langle règle \rangle^\star_t$	répétition de la règle $\langle règle \rangle$ un nombre quelconque de fois (y compris aucune), les occurrences étant séparées par le terminal t
$\langle règle \rangle^+$	répétition de la règle $\langle règle \rangle$ au moins une fois
$\langle règle \rangle^+_t$	répétition de la règle $\langle règle \rangle$ au moins une fois, les occurrences étant séparées par le terminal t
$\langle règle \rangle$?	utilisation optionnelle de la règle $\langle règle \rangle$ (*i.e.* 0 ou 1 fois)
($\langle règle \rangle$)	parenthésage ; attention à ne pas confondre ces parenthèses avec les terminaux `(` et `)`

1.1 Conventions lexicales

Espaces et tabulations constituent des blancs. Les commentaires peuvent prendre deux formes :

- débutant par `{-` et s'étendant jusqu'à `-}`, et ne pouvant pas être imbriqués ;
- débutant par `--` et s'étendant jusqu'à la fin de la ligne.

Les identificateurs obéissent à l'expression régulière $\langle ident \rangle$ suivante :

$$
\begin{array}{lcl}
\langle digit \rangle & ::= & \texttt{0–9} \\
\langle lower \rangle & ::= & \texttt{a–z} \mid \texttt{\_} \\
\langle upper \rangle & ::= & \texttt{A–Z} \\
\langle other \rangle & ::= & \langle lower \rangle \mid \langle upper \rangle \mid \langle digit \rangle \mid \texttt{'} \\
\langle lident \rangle & ::= & \langle lower \rangle \, \langle other \rangle^\star \\
\langle uident \rangle & ::= & \langle upper \rangle \, (\langle other \rangle \mid \texttt{.})^\star
\end{array}
$$

Les identificateurs suivants sont des mots clés :

```
case    class      data    do       else    false    forall  if      import
in      instance   let     module   of      then     true    where
```

Les constantes entières obéissent à l'expression régulière $\langle integer \rangle$ suivante :

1. https://www.purescript.org/

$$\langle \textit{integer} \rangle \quad ::= \quad \texttt{0} \mid \texttt{1–9}\ \langle \textit{digit} \rangle^\star$$

Une chaîne de caractères ⟨*string*⟩ s'écrit entre guillemets (`"`). Il y a trois séquences d'échappement : `\"` pour le caractère `"`, `\\` pour le caractère `\`, et `\n` pour un retour chariot. Par ailleurs, une chaîne peut contenir une séquence arbitraire de caractères blancs (espaces, tabulations et retours chariot) entre deux caractères `\` et tous ces caractères sont alors ignorés.

1.2 Indentation significative

Certaines constructions du langage Purescript utilisent l'alignement vertical (une même colonne dans le texte source) pour en déterminer la structure syntaxique. Plus précisément, les lexèmes `where`, `do`, `let` et `of` introduisent une structure de *bloc*. Ces blocs sont matérialisés dans la grammaire du langage par trois *lexèmes factices* : `{`, `}` et `;`. Ces lexèmes sont introduits au niveau de l'analyse lexicale et l'analyse syntaxique est réalisée de manière traditionnelle (voir section suivante).

Voici un exemple, avec à gauche le texte source et à droite la suite de lexèmes qui est produite par l'analyseur lexical :

texte source	lexèmes produits

`main = let x = "a"`
`y = "b" in`
`do`
`log x`
`log y`
	`main = let { x = "a" ;`
	`y = "b" } in`
	`do {`
	`log x ;`
	`log y }`

Explications :

- Le lexème `let` ouvre un bloc : d'une part, on émet le lexème `{` ; d'autre part, on lit le prochain lexème (`x`) et on note sa colonne (ici 11).
- Pour les lexèmes suivants, on insère un lexème `;` si la colonne prend de nouveau la valeur 11 ; c'est le cas ici pour le lexème `y`.
- Le lexème `in` ferme le bloc courant. On insère donc le lexème `}` avant `in`.
- Le lexème `do` se comporte comme le lexème `let`. On note que la colonne du bloc (ici 2) peut être inférieure à la colonne du mot-clé qui l'introduit (ici 4). On note également que c'est la fin de fichier qui a produit la fermeture de ce bloc.

Algorithme. On utilise une *pile* qui contient soit une indication de bloc associé à une certaine colonne ($B\ n$) soit un marqueur d'expression (M). On lit les lexèmes un par un (avec un outil de son choix) et, pour chacun, on effectue une ou plusieurs actions selon le tableau de la figure 1. Dans cette figure, on désigne par « fermer (c) » l'action qui dépile des $B\ n$ en sommet de pile tant que $n > c$, en émettant un lexème `}` à chaque fois, puis conclut en émettant un lexème `;` si on termine avec $B\ c$ en sommet de pile. Par ailleurs, il y a deux modes possibles : le mode fort, par défaut, et le mode faible, qui est utilisé lorsqu'on traite récursivement un lexème dans l'algorithme. Dans le mode faible, l'opération « fermer (c) » n'a aucun effet.

Enfin, il convient de traiter la fin de fichier correctement. On peut le faire en considérant que le lexème `EOF` est sur la colonne -1 et en le traitant dans le cas général (dernière ligne).

Suggestion. Commencer par écrire un analyseur lexical traditionnel (avec un outil de type `lex`), puis construire par dessus un second analyseur lexical qui insère à la volée les lexèmes factices. On pourra par exemple utiliser une *file* pour stocker les lexèmes en attente d'être envoyés à l'analyseur syntaxique, afin de retrouver le type attendu d'un analyseur lexical qui envoie les lexèmes un par un. De nombreux tests, négatifs et positifs, sont fournis pour aider à comprendre ce mécanisme et à mettre au point l'analyseur lexical.

lexème (de colonne c)	action
`if`, `(`, `case`	empiler M et émettre le lexème
`)`, `then`, `else`, `in`	dépiler jusqu'à trouver M pour `then`, empiler M émettre le lexème
`where`, `do`, `let`, `of`	fermer(c) pour `let`, empiler M pour `of`, dépiler jusqu'à trouver M émettre le lexème suivi de `{` lire le lexème suivant t, de colonne c', et fermer(c') empiler B c' traiter le nouveau lexème t récursivement *en mode faible*
autre	fermer(c) émettre le lexème

FIGURE 1 – Traitement des lexèmes.

1.3 Syntaxe

La grammaire des fichiers sources considérée est donnée figure 2. Le point d'entrée est le non terminal ⟨*file*⟩. Les associativités et précédences des diverses constructions sont données par la table suivante, de la plus faible à la plus forte précédence :

opérateur ou construction	**associativité**
`in else`	—
`||`	gauche
`&&`	gauche
`==`, `!=`, `>`, `>=`, `<`, `<=`	—
`+`, `-`; `<>`	gauche
`*`, `/`	gauche
`-` (unaire)	—

2 Typage statique

Sera donné plus tard.

3 Production de code

Sera donné plus tard.

4 Travail demandé

Le projet est à faire seul ou en binôme. Il doit être remis par email[2], sous la forme d'une archive compressée (avec `tar` ou `zip`), appelée `vos_noms.tgz` ou `vos_noms.zip` qui doit contenir un répertoire appelé `vos_noms` (exemple : `dupont-durand`). Dans ce répertoire doivent se trouver les *sources* du compilateur (inutile d'inclure les fichiers compilés). Quand on se place dans ce

2. à Jean-Christophe.Filliatre@cnrs.fr

$$
\begin{array}{lcl}
\langle \mathit{file} \rangle & ::= & \texttt{module Main where \{} \ \langle \mathit{imports} \rangle \ \langle \mathit{decl} \rangle^{+}_{\texttt{;}} \ \texttt{\}} \ \texttt{EOF} \\
\langle \mathit{imports} \rangle & ::= & \texttt{import Prelude ;} \\
 & & \texttt{import Effect ;} \\
 & & \texttt{import Effect.Console ;} \\
\langle \mathit{decl} \rangle & ::= & \langle \mathit{defn} \rangle \mid \langle \mathit{tdecl} \rangle \\
 & \mid & \texttt{data} \ \langle \mathit{uident} \rangle \ \langle \mathit{lident} \rangle^{\star} \ \texttt{=} \ (\langle \mathit{uident} \rangle \ \langle \mathit{atype} \rangle^{\star})^{+}_{\texttt{|}} \\
 & \mid & \texttt{class} \ \langle \mathit{uident} \rangle \ \langle \mathit{lident} \rangle^{\star} \ \texttt{where \{} \ \langle \mathit{tdecl} \rangle^{\star}_{\texttt{;}} \ \texttt{\}} \\
 & \mid & \texttt{instance} \ \langle \mathit{instance} \rangle \ \texttt{where \{} \ \langle \mathit{defn} \rangle^{\star}_{\texttt{;}} \ \texttt{\}} \\
\langle \mathit{defn} \rangle & ::= & \langle \mathit{lident} \rangle \ \langle \mathit{patarg} \rangle^{\star} \ \texttt{=} \ \langle \mathit{expr} \rangle \\
\langle \mathit{tdecl} \rangle & ::= & \langle \mathit{lident} \rangle \ \texttt{::} \ (\texttt{forall} \ \langle \mathit{lident} \rangle^{+} \texttt{.})? \\
 & & (\langle \mathit{ntype} \rangle \ \texttt{=>})^{\star} \ (\langle \mathit{type} \rangle \ \texttt{->})^{\star} \ \langle \mathit{type} \rangle \\
\langle \mathit{ntype} \rangle & ::= & \langle \mathit{uident} \rangle \ \langle \mathit{atype} \rangle^{\star} \\
\langle \mathit{atype} \rangle & ::= & \langle \mathit{lident} \rangle \mid \langle \mathit{uident} \rangle \mid \texttt{(} \ \langle \mathit{type} \rangle \ \texttt{)} \\
\langle \mathit{type} \rangle & ::= & \langle \mathit{atype} \rangle \mid \langle \mathit{ntype} \rangle \\
\langle \mathit{instance} \rangle & ::= & \langle \mathit{ntype} \rangle \\
 & \mid & \langle \mathit{ntype} \rangle \ \texttt{=>} \ \langle \mathit{ntype} \rangle \\
 & \mid & \texttt{(} \ \langle \mathit{ntype} \rangle^{+}_{\texttt{,}} \ \texttt{)} \ \texttt{=>} \ \langle \mathit{ntype} \rangle \\
\langle \mathit{patarg} \rangle & ::= & \langle \mathit{constant} \rangle \mid \langle \mathit{lident} \rangle \mid \langle \mathit{uident} \rangle \mid \texttt{(} \ \langle \mathit{pattern} \rangle \ \texttt{)} \\
\langle \mathit{pattern} \rangle & ::= & \langle \mathit{patarg} \rangle \mid \langle \mathit{uident} \rangle \ \langle \mathit{patarg} \rangle^{+} \\
\langle \mathit{constant} \rangle & ::= & \texttt{true} \mid \texttt{false} \mid \langle \mathit{integer} \rangle \mid \langle \mathit{string} \rangle \\
\langle \mathit{atom} \rangle & ::= & \langle \mathit{constant} \rangle \\
 & \mid & \langle \mathit{lident} \rangle \\
 & \mid & \langle \mathit{uident} \rangle \\
 & \mid & \texttt{(} \ \langle \mathit{expr} \rangle \ \texttt{)} \\
 & \mid & \texttt{(} \ \langle \mathit{expr} \rangle \ \texttt{::} \ \langle \mathit{type} \rangle \ \texttt{)} \\
\langle \mathit{expr} \rangle & ::= & \langle \mathit{atom} \rangle \\
 & \mid & \texttt{-} \ \langle \mathit{expr} \rangle \\
 & \mid & \langle \mathit{expr} \rangle \ \langle \mathit{binop} \rangle \ \langle \mathit{expr} \rangle \\
 & \mid & (\langle \mathit{lident} \rangle \mid \langle \mathit{uident} \rangle) \ \langle \mathit{atom} \rangle^{+} \\
 & \mid & \texttt{if} \ \langle \mathit{expr} \rangle \ \texttt{then} \ \langle \mathit{expr} \rangle \ \texttt{else} \ \langle \mathit{expr} \rangle \\
 & \mid & \texttt{do \{} \ \langle \mathit{expr} \rangle^{+}_{\texttt{;}} \ \texttt{\}} \\
 & \mid & \texttt{let \{} \ \langle \mathit{binding} \rangle^{+}_{\texttt{;}} \ \texttt{\}} \ \texttt{in} \ \langle \mathit{expr} \rangle \\
 & \mid & \texttt{case} \ \langle \mathit{expr} \rangle \ \texttt{of \{} \ \langle \mathit{branch} \rangle^{+}_{\texttt{;}} \ \texttt{\}} \\
\langle \mathit{binding} \rangle & ::= & \langle \mathit{lident} \rangle \ \texttt{=} \ \langle \mathit{expr} \rangle \\
\langle \mathit{branch} \rangle & ::= & \langle \mathit{pattern} \rangle \ \texttt{->} \ \langle \mathit{expr} \rangle \\
\langle \mathit{binop} \rangle & ::= & \texttt{==} \mid \texttt{/=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{<>} \mid \texttt{\&\&} \mid \texttt{||}
\end{array}
$$

FIGURE 2 – Grammaire de Petit Purescript.

répertoire, la commande `make` doit créer le compilateur, qui sera appelé `ppurs`. La commande `make clean` doit effacer tous les fichiers que `make` a engendrés et ne laisser dans le répertoire que les fichiers sources. Bien entendu, la compilation du projet peut être réalisée avec d'autres outils que `make` (par exemple `dune` si le projet est réalisé en OCaml) et le `Makefile` se réduit alors à quelques lignes seulement pour appeler ces outils.

L'archive doit également contenir un court rapport expliquant les différents choix techniques qui ont été faits et, le cas échéant, les difficultés rencontrées ou les éléments non réalisés. Ce rapport pourra être fourni dans un format texte (par exemple Markdown) ou PDF.

Partie 1 (à rendre pour le dimanche 10 décembre 18:00). Dans cette première partie du projet, le compilateur `ppurs` doit accepter sur sa ligne de commande une option éventuelle (parmi `--parse-only` et `--type-only`) et exactement un fichier Petit Purescript portant l'extension `.purs`. Il doit alors réaliser l'analyse syntaxique du fichier. En cas d'erreur lexicale ou syntaxique, celle-ci doit être signalée le plus précisément possible, par sa nature et sa localisation dans le fichier source. On adoptera le format suivant pour cette signalisation :

```
File "test.purs", line 4, characters 5-6:
syntax error
```

L'anglicisme de la première ligne est nécessaire pour que la fonction `next-error` d'Emacs[3] puisse interpréter la localisation et placer ainsi automatiquement le curseur sur l'emplacement de l'erreur. En revanche, le message d'erreur proprement dit pourra être écrit en français si vous le souhaitez. En cas d'erreur, le compilateur doit terminer avec le code de sortie 1 (`exit 1`).

Si le fichier est syntaxiquement correct, le compilateur doit terminer avec le code de sortie 0 si l'option `--parse-only` a été passée sur la ligne de commande. Sinon, il doit poursuivre avec le typage du fichier source. Lorsqu'une erreur de typage est détectée par le compilateur, elle doit être signalée le plus précisément possible, de la manière suivante :

```
File "test.purs", line 4, characters 5-6:
this expression has type int but is expected to have type String
```

Là encore, la nature du message est laissée à votre discrétion, mais la forme de la localisation est imposée. Le compilateur doit alors terminer avec le code de sortie 1 (`exit 1`). Si en revanche il n'y a pas d'erreur de typage, le compilateur doit terminer avec le code de sortie 0. En cas d'erreur du compilateur lui-même, le compilateur doit terminer avec le code de sortie 2 (`exit 2`). L'option `--type-only` indique de stopper la compilation après l'analyse sémantique (typage). Elle est donc sans effet dans cette première partie, mais elle doit être honorée néanmoins.

Partie 2 (à rendre pour le dimanche 21 janvier 18:00). Si le fichier d'entrée est conforme à la syntaxe et au typage de ce sujet, votre compilateur doit produire du code x86-64 et terminer avec le code de sortie 0, sans rien afficher. Si le fichier d'entrée est `file.purs`, le code assembleur doit être produit dans le fichier `file.s` (même nom que le fichier source mais suffixe `.s` au lieu de `.purs`). Ce fichier x86-64 doit pouvoir être exécuté avec la commande

```
gcc -no-pie file.s -o file
./file
```

Le résultat affiché sur la sortie standard doit être identique à celui donné par la commande

```
spago run
```

lorsque le fichier `src/Main.purs` est identique à `file.purs` (en supposant se trouver dans un projet correctement initialisé avec `spago init`).

3. Le correcteur est susceptible d'utiliser cet éditeur.

Remarque importante. La correction du projet est réalisée en partie automatiquement, à l'aide d'un jeu de petits programmes réalisant des affichages avec l'instruction `log`, qui sont compilés avec votre compilateur et dont la sortie est comparée à la sortie attendue. Il est donc très important de correctement compiler les appels à `log`.

Conseils. Il est fortement conseillé de procéder construction par construction que ce soit pour le typage ou pour la production de code, dans cet ordre : affichage (`log`), arithmétique, variables locales, `if`, `case`, fonctions, classes de types.