projet de compilation

Petit Purescript

version 4 — 7 décembre 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/

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

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 `{` et 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` ouvre un bloc, 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 ici 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.

Quand on exécute « dépiler jusqu'à trouver M », on émet un lexème `}` pour tout $B\ n$ dépilé. Une fois le M trouvé, il est également dépilé, mais sans émettre de `}` cette fois. V2

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

lexème (de colonne c)	action
`if`, `(`, `case`	1. fermer(c) 2. empiler M et émettre le lexème
`)`, `then`, `else`, `in`	1. dépiler jusqu'à trouver M (voir texte) 2. pour `then`, empiler M 3. émettre le lexème
`where`, `do`, `let`, `of`	1. fermer(c) 2. pour `let`, empiler M ; pour `of`, dépiler jusqu'à trouver M (idem) 3. émettre le lexème suivi de `{` 4. lire le lexème suivant t, de colonne c', et fermer(c') 5. empiler B c' 6. traiter le nouveau lexème t récursivement *en mode faible*
autre	1. fermer(c) 2. émettre le lexème

FIGURE 1 – Traitement des lexèmes.

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.

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)	—

$$
\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})^{+}_{\mid} \\
 & \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.

2 Typage statique

Une fois l'analyse syntaxique effectuée avec succès, on vérifie la conformité du fichier source. Un type τ représente un type que l'on peut donner à une expression. Les types sont de la forme suivante,

$$\tau \quad ::= \quad \alpha \mid T\ \tau \ldots \tau$$

où α est une variable de type (un identificateur commençant par une minuscule dans la syntaxe concrète) et T est un symbole de type (un identificateur commençant par une majuscule dans la syntaxe concrète). Par ailleurs, on désigne par C un symbole de *classe de types*. Chaque classe a une arité $k \geq 0$, c'est-à-dire qu'elle doit être appliquée à k types. Le résultat de cette application est une *instance*, notée I.

$$\begin{array}{rcl} I & ::= & C\ \tau \ldots \tau \\ S & ::= & (I, \ldots, I) \Rightarrow I \end{array}$$

Un schéma d'instance est noté S.

Environnement de typage. L'environnement de typage global contient des déclarations de types, de constructeurs, de fonctions, de classes de types, d'instances et de schémas d'instances. Il y a cinq symboles de types prédéfinis :

`Unit`, `Boolean`, `Int`, `String`, `Effect a`

Il y a une constante et quatre fonctions prédéfinies : V4

```
unit:: Unit
 not:: Boolean -> Boolean
 mod:: Int -> Int -> Int
 log:: String -> Effect Unit
pure:: forall a. a -> Effect a
```

Il y a une classe de types prédéfinie :

```
class Show a where
  show:: a -> String
```

Il y a deux instances prédéfinies :

```
instance Show Boolean
instance Show Int
```

À cet environnement global s'ajoute un environnement de typage local, noté Γ, qui contient

- un ensemble de variables de types $\alpha_1, \alpha_2, \ldots$;
- une suite ordonnée de déclarations de variables de la forme $x : \tau$;
- un ensemble d'instances $I_1, I_2, \ldots$

On note $\Gamma + x : \tau$ l'environnement Γ étendu avec une nouvelle déclaration de variable. Une précédente déclaration de x dans Γ est, le cas échéant, remplacée par cette nouvelle déclaration. Pour un motif p, on note $\Gamma + p$ l'environnement Γ étendu avec toutes les variables de p.

Type bien formé. Le jugement $\Gamma \vdash \tau\ bf$ signifie « le type τ est bien formé dans l'environnement Γ ». Il est défini ainsi :

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha\ bf} \qquad \frac{T \text{ d'arité } n \quad \forall i,\ \Gamma \vdash \tau_i\ bf}{\Gamma \vdash T\ \tau_1 \ldots \tau_n\ bf}$$

2.1 Typage des expressions

On introduit le jugement $\Gamma \vdash e : \tau$ signifiant « dans le contexte Γ, l'expression e est bien typée de type τ ». Ce jugement est défini par les règles d'inférence suivantes :

$$\frac{c \text{ constante de type } \tau}{\Gamma \vdash c : \tau} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash \tau \ bf \quad \Gamma \vdash e : \tau}{\Gamma \vdash (e :: \tau) : \tau}$$

$$\frac{\Gamma \vdash e : \texttt{Int}}{\Gamma \vdash \texttt{-}\ e : \texttt{Int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{Int} \quad \Gamma \vdash e_2 : \texttt{Int} \quad op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{\Gamma \vdash e_1 \ op \ e_2 : \texttt{Int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Int} \quad \Gamma \vdash e_2 : \texttt{Int} \quad op \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\Gamma \vdash e_1 \ op \ e_2 : \texttt{Boolean}}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \{\texttt{Boolean}, \texttt{Int}, \texttt{String}\} \quad op \in \{\texttt{==}, \texttt{/=}\}}{\Gamma \vdash e_1 \ op \ e_2 : \texttt{Boolean}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Boolean} \quad \Gamma \vdash e_2 : \texttt{Boolean} \quad op \in \{\texttt{\&\&}, \texttt{||}\}}{\Gamma \vdash e_1 \ op \ e_2 : \texttt{Boolean}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{String} \quad \Gamma \vdash e_2 : \texttt{String}}{\Gamma \vdash e_1 \ \texttt{<>}\ e_2 : \texttt{String}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Boolean} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

$$\frac{\forall i,\ \Gamma \vdash e_i : \texttt{Effect Unit}}{\Gamma \vdash \texttt{do}\ e_1\ \ldots\ e_n : \texttt{Effect Unit}}$$

$$\frac{\forall i,\ \Gamma_{i-1} \vdash e_i : \tau_i \quad \Gamma_i \stackrel{\text{def}}{=} \Gamma_{i-1} + x_i : \tau_i \quad \Gamma_n \vdash e : \tau}{\Gamma_0 \vdash \texttt{let}\ x_1 = e_1\ \ldots\ x_n = e_n\ \texttt{in}\ e : \tau}$$

$$\frac{\texttt{data}\ T\ \overline{\alpha} = \ldots X\ \tau_1 \ldots \tau_n \ldots \quad \forall i,\ \Gamma \vdash e_i : \sigma\tau_i}{\Gamma \vdash X\ e_1\ \ldots\ e_n : T\ \sigma\overline{\alpha}}$$

$$\frac{f :: \forall \overline{\alpha},\ I_1 \Rightarrow \cdots \Rightarrow I_\ell \Rightarrow \tau_1 \to \ldots \to \tau_n \to \tau \quad \forall i,\ \Gamma \vdash e_i : \sigma\tau_i \quad \forall i,\ \Gamma \vdash \sigma I_i}{\Gamma \vdash f\ e_1\ \ldots\ e_n : \sigma\tau}$$

Dans les deux dernières règles, on note σ une substitution des variables $\overline{\alpha}$ vers autant de types bien formés dans Γ.

$$\frac{\Gamma \vdash e : \tau \quad \forall i,\ \Gamma \vdash p_i : \tau \quad \Gamma + p_i \vdash e_i : \tau'}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ p_1\texttt{->}e_1 \ldots \texttt{of}\ p_n\texttt{->}e_n : \tau'}$$

Pour cette dernière règle (`case`), il faut également vérifier que le filtrage est exhaustif. En revanche, il peut y avoir des clauses inutiles (c'est-à-dire un motif déjà couvert par un motif précédent).

Le jugement $\Gamma \vdash p : \tau$ signifie « dans le contexte Γ, le motif p est bien typé de type τ ». Ce jugement est défini par les règles d'inférence suivantes :

$$\frac{c \text{ constante de type } \tau}{\Gamma \vdash c : \tau} \qquad \frac{}{\Gamma \vdash x : \tau}$$

$$\frac{\texttt{data}\ T\ \overline{\alpha} = \ldots X\ \tau_1 \ldots \tau_n \ldots \quad \forall i,\ \Gamma \vdash p_i : \sigma\tau_i}{\Gamma \vdash X\ p_1\ \ldots\ p_n : T\ \sigma\overline{\alpha}}$$

Par ailleurs, un motif ne doit pas contenir deux fois la même variable.

Le jugement $\Gamma \vdash I$ signifie « dans le contexte Γ, l'instance I est résolue ». Ce jugement est défini par les règles d'inférence suivantes :

$$\frac{I \in \Gamma \text{ ou } I \text{ globale}}{\Gamma \vdash \sigma I} \qquad \frac{(I_1, \ldots, I_\ell) \Rightarrow I \quad \forall i,\ \Gamma \vdash \sigma I_i}{\Gamma \vdash \sigma I}$$

Dit autrement, on résout une instance en la trouvant dans l'environnement global ou local, ou en la construisant récursivement à partir de schémas. Ici, la substitution σ autorise à instancier une instance polymorphe pour résoudre une instance plus précise. Ainsi, si on dispose d'une instance `C a`, alors on peut résoudre l'instance `C Int`.

2.2 Typage d'un fichier

Les déclarations constituant un fichier sont traitées dans l'ordre d'apparition. Les types, fonctions, classes et instances sont ajoutées à l'environnement global au fur et à mesure de leur analyse. Une déclaration ne peut faire référence qu'à des déclarations antérieures (ou à la déclaration courante). Chaque type, constructeur, fonction ou classe ne peut être défini qu'une seule fois.

Déclaration de fonction. Une fonction f est introduite par une déclaration de type, suivie d'une définition décomposée en une ou plusieurs équations.

$$\begin{array}{l} f :: \texttt{forall}\ \alpha_1 \dots \alpha_k .\ I_1 \Rightarrow \dots \Rightarrow I_\ell \Rightarrow \tau_1 \to \dots \to \tau_n \to \tau \\ f\ p_{1,1}\ \dots\ p_{1,n} = e_1 \\ \vdots \\ f\ p_{m,1}\ \dots\ p_{m,n} = e_m \end{array}$$

Les instances I_i, les types τ_i et le type τ doivent être bien formés dans l'environnement $\Gamma_0 \stackrel{\text{def}}{=} \{\alpha_1, \dots, \alpha_k\}$. Par ailleurs, les motifs $p_{i,j}$ et les expressions e_i doivent être bien typées dans l'environnement $\Gamma \stackrel{\text{def}}{=} \Gamma_0 + \{I_1, \dots, I_\ell\}$, c'est-à-dire : V3

$$\forall i, j,\ \Gamma \vdash p_{i,j} : \tau_j \qquad \text{et} \qquad \forall i,\ \Gamma + p_{i,1} + \dots + p_{i,n} \vdash e_i : \tau$$

Dans une équation, les motifs doivent introduire des variables différentes. Enfin, tous les motifs doivent être des variables, à l'exception d'une colonne au plus, qui doit alors constituer un filtrage exhaustif (mais possiblement redondant, comme pour l'expression `case`).

Déclaration de type. On introduit un nouveau symbole de type T avec la déclaration suivante :

$$\texttt{data}\ T\ \alpha_1 \dots \alpha_k = X_1\ \tau_{1,1} \dots \tau_{1,n_1}\ \texttt{|}\ \dots\ \texttt{|}\ X_m\ \tau_{m,1} \dots \tau_{m,n_m}$$

Les variables de types α_i doivent porter des noms différents. Les constructeurs X_i doivent porter des noms différents. Les types $\tau_{i,j}$ doivent être bien formés dans l'environnement $\Gamma \stackrel{\text{def}}{=} \{\alpha_1, \dots, \alpha_k\}$.

Déclaration de classe. On introduit une nouvelle classe de types C avec la déclaration suivante :

$$\begin{array}{l} \texttt{class}\ C\ \alpha_1 \dots \alpha_k\ \texttt{where} \\ \quad f_1 :: \tau_{1,1} \to \dots \to \tau_{1,n_1} \\ \quad \vdots \\ \quad f_m :: \tau_{m,1} \to \dots \to \tau_{m,n_m} \end{array}$$

Les variables de types α_i doivent porter des noms différents. Les fonctions f_i doivent porter des noms différents et nouveaux. Les types $\tau_{i,j}$ doivent être bien formés dans l'environnement $\Gamma \stackrel{\text{def}}{=} \{\alpha_1, \dots, \alpha_k\}$. On ajoute à l'environnement global les m fonctions suivantes :

$$f_i\ ::\ \texttt{forall}\ \alpha_1 \dots \alpha_k\ .\ C\ \alpha_1 \dots \alpha_k \Rightarrow \tau_{i,1} \to \dots \to \tau_{i,n_i}$$

Déclaration d'instance. Pour une certaine classe de types C d'arité k, on introduit une nouvelle instance (ou un nouveau schéma) pour cette classe avec la déclaration suivante :

$$\begin{array}{l}\texttt{instance (}I_1, \ldots, I_\ell\texttt{) => }C\ \tau_1 \ldots \tau_k\ \texttt{where}\\ \quad f_1\ p_{1,1}\ \ldots\ p_{1,n_1}\ \texttt{=}\ e_1\\ \quad \vdots\\ \quad f_m\ p_{m,1}\ \ldots\ p_{m,n_m}\ \texttt{=}\ e_m\end{array}$$

Les différentes équations doivent définir exactement les fonctions de la classe C. Chaque définition de fonction doit être composée d'équations successives dans la déclaration (c'est-à-dire qu'on ne mélange pas les équations de différentes fonctions). Chaque définition doit être conforme à sa déclaration dans la classe $C\ \overline{\alpha}$ une fois qu'on a appliqué à cette déclaration la substitution $\sigma \stackrel{\text{def}}{=} \overline{\alpha} \mapsto \overline{\tau}$. Chaque définition est typée dans un environnement qui contient d'une part les variables de types introduites par $I_1, \ldots, I_\ell, C\ \tau_1 \ldots \tau_k$ et d'autre part les instances $I_1, \ldots, I_\ell$.

Par ailleurs, deux instances différentes ne doivent pas pouvoir être unifiées. Ainsi, pour une classe `C a b`, on ne peut pas déclarer une instance pour `C Int b` et une instance pour `C a Int`.

2.3 Limitations par rapport à Purescript

Le langage Petit Purescript souffre d'un certain nombre de limitations par rapport à Purescript, à savoir :

- La comparaison `==` est limitée aux types `Boolean`, `Int` et `String`. (En Purescript, `==` est un alias pour la fonction `eq` de la classe `Eq`.)
- Dans une définition de fonction, le filtrage est limité à un seul paramètre.
- Les définitions ne sont pas mutuellement récursives (dans le fichier comme dans un `let`), mais introduites au fur et à mesure dans l'environnement.
- Petit Purescript possède moins de mots clés que Purescript.

Cependant, votre compilateur ne sera jamais testé sur des programmes incorrects au sens de Petit Purescript mais corrects au sens de Purescript.

2.4 Indications

Il est fortement conseillé de procéder construction par construction, en compilant et testant systématiquement son projet à chaque étape. De nombreux tests sont fournis sur la page du cours, avec un script pour lancer votre compilateur sur ces tests.

En cas de doute concernant un point de sémantique, vous pouvez utiliser un interprète ou un compilateur Purescript comme référence. Vous pouvez d'ailleurs vous inspirer de ses messages d'erreur pour votre compilateur (en les traduisant ou non en français).

Dans la phase suivante (production de code), certaines informations provenant du typage seront nécessaires. Il vous est conseillé d'anticiper ces besoins en programmant des fonctions de typage qui ne se contentent pas de parcourir les arbres de syntaxe abstraite issus de l'analyse syntaxique mais en renvoient de nouveaux, contenant notamment des types. Ces nouveaux arbres de syntaxe abstraite peuvent différer plus ou moins des arbres issus de l'analyse syntaxique. Ainsi, on suggère de profiter du typage pour transformer une fonction définie par plusieurs équations en une fonction définie par une seule équation et un `case` sur l'un de ses arguments.

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 répertoire, la commande `make` doit créer le compilateur, qui sera appelé `ppurs`. La commande `make clean` doit effacer tous les fichiers que `make` a produits 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

2. à Jean-Christophe.Filliatre@cnrs.fr
3. Le correcteur est susceptible d'utiliser cet éditeur.

```
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`).

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. (Voir les tests fournis dans le sous-répertoire `exec/`.) 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.