

# Compilateur purscript: compte rendu

20 janvier 2024

## Le lexer

Le lexer est codé dans le fichier `purscript_lexer.mll` et est compatible avec `ocamllex`.

La plupart des lexemes sont reconnus directement par la règle principale. Il y a quelques cas où l'on utilise une règle différente :

- les commentaires sur plusieurs lignes
- les chaînes de caractères
- les chaînes de caractères entre les symboles `"\"`

## Le parser

Il est codé dans le fichier `purscrip_parser.mly` et est compatible avec `menhir`.

Il reprend globalement la grammaire proposée pour le purscript. Mais doit gérer quelques cas délicats.

Le premier problème vient de la règle :

$$\langle \text{tdecl} \rangle ::= \langle \text{lident} \rangle :: (\langle \text{ntype} \rangle \Rightarrow)^* (\langle \text{type} \rangle - >)^* \langle \text{type} \rangle$$

Dans ce cas, il n'est pas possible de juste reconnaître grâce à la règle suivante :

$$\text{list}(\text{ntype}, \Rightarrow) \text{list}(\text{type}, - >) \text{type}$$

En effet, cela provoque un conflit car le parser ne sait pas quand passer d'une liste à l'autre.

La solution est donc de remplacer par 3 règles qui s'appellent en chaîne et qui permettent de placer nous-même les limites.

Le deuxième problème vient de deux dérivations possibles dans la grammaire. En effet, si l'on souhaite reconnaître un objet de 'type' et que l'on lit un 'uident', alors les deux dérivations suivantes sont possibles :

$$\left[ \begin{array}{l} \text{type} \rightarrow \text{atype} \rightarrow \text{uident} \\ \text{type} \rightarrow \text{ntype} \rightarrow \text{uident} \end{array} \right].$$

Mais pour la suite, ces deux dérivations sont équivalentes, donc nous avons forcé le parser à reconnaître l'un des deux (en l'occurrence le premier).

## Le typage : point de vue général

Pour chaque type de l'ast, on crée une fonction qui renvoie son type (par exemple `typexpr` renvoie le type d'une expression)

Pour vérifier qu'un fichier est bien typé, on appelle récursivement ces différentes fonctions en vérifiant systématiquement que les types sont bien cohérents (càd par exemple qu'on n'additionne pas 2 String)

## Le typage : les différents types

Un type peut être :

- Un des 4 types de base (Int, Boolean, String, Unit)
- Un type polymorphe (Tgeneral a)
- Un type créé par l'utilisateur et d'arité  $\geq 0$  (Tcustom("Pair",[Int,String]))
- Tany = un type qui peut être n'importe quoi, par exemple Nil dans `pascal.purs` est un Tcustom("List",[Tany])

## Le typage : les patternes

Pour les patternes, on crée la fonction `ensuretyppattern` qui prend en argument un pattern, un environnement et un type, vérifie que le pattern est bien cohérent avec le type et renvoie l'environnement initial où on a ajouté les nouveaux ident apparus dans le pattern

De plus, la fonction `checkexhaustivelist` prend en argument une liste de liste de patternes et vérifie récursivement que cette liste est bien exhaustive.

# Le typage : les environnements

On va stocker le type de tous les ident dans des environnements locaux passés en paramètre à à peu près toutes les fonctions

Il y a un environnement global pour les types (nommé `envtypes`) où sont stockés les types déclarés par `data`, auquel s'ajoute un environnement local qui contient les variables de type et passé en paramètre

Les fonctions et les classes sont quant à elles stockées dans un environnement global (nommé respectivement `globalenvfonctions` et `envclasses`)

De même que pour les types, Il y a un environnement global pour les instances (nommé `globalenvinstances`) où sont stockés les instances déclarées par `Instance`, auquel s'ajoute un environnement local qui contient les instances passées en paramètre aux fonctions et passé en paramètre (nommé `envinstances`)

# Le typage : gestion des instances

Quand une fonction `f` liée à une classe `C` est utilisée, on regarde une à une toutes les instances liées à `C` et on prend la première qui est cohérente avec les types des différents arguments de `f`

Si on n'en trouve aucune, alors on retourne une erreur.

Quand on définit une nouvelle instance, on l'ajoute à un environnement d'instance global.

De plus, pour les instances qui demande des instances en paramètre (càd des instances définies récursivement), on rajoute dans l'environnement d'instances local (initialisé à l'environnement d'instance global) les instances passées en paramètre

# Le typage : gestion des erreurs

L'`ast` est légèrement modifiée par rapport à la grammaire fournie de manière à associer à chaque type de l'`ast` une position dans le fichier

Quand une erreur est détectée, on retourne cette position et on affiche le bout du fichier qui y correspond

# L'arbre de typage

Le typage va renvoyer un arbre très proche de celui renvoyé par l'analyse syntaxique, mais avec quelques différences :

- Les expressions et atomes ont leur type indiqué
- Les fonctions avec plusieurs définitions n'en ont maintenant plus qu'une, on a juste rajouté un case au début
- Les instances ne sont plus renvoyées comme telles, à la place chaque fonction de chaque instance est considérée comme sa propre fonction, et quand on appelle une instance, on appelle la seule fonction de la classe qui convient

## Le typage : ce qui n'a pas été codé

Nous avons admis une limitation ambiguë de Petit Purescript (mais qui ne contredit aucun des tests fournis) : nous considérons évidemment que chaque fonction ne peut avoir du filtrage que sur un des paramètres, les autres devant être des ident. Mais en plus nous supposons que les noms d'un paramètre sont les mêmes pour toutes les déclarations de la fonction (cependant, il peut y avoir un `_` et un ident sur le même paramètre, comme dans `queens.purs` par exemple)

De plus, les instances récursives ne compilent pas très bien. En effet, nous avons choisi de considérer chaque instance comme sa propre fonction. Cependant, il est à noter que tous les tests qui utilisent les instances récursives fonctionnent, sauf `ral.purs`, mais pour une raison peu satisfaisante qui provient de la simplicité des tests (l'instance utilisée en paramètre est définie juste après).

## Makefile / dune / tests

Le projet contient également des scripts permettant d'automatiser la compilation et de tester rapidement sur de nombreux tests.

La plupart des tests sont fournis mais nous avons rajouté quelques tests afin de vérifier le comportement sur quelques cas particuliers.

Dune s'occupe de produire le fichier `purescript_main.exe`. Et Make s'occupe de lancer dune, de renommer l'exécutable, et de lancer les tests.