

# Compilateur purscript : rendu 1

13 décembre 2023

## 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 :

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

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

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

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`)

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)

De plus, pour les patternes, on crée la fonction `ensuretyppattern` qui prend en argument un patterne et un type, vérifie que le patterne est bien cohérent avec le type et renvoie un environnement où on a ajouté les nouveaux ident apparus dans le patterne

## 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 `envtyps`) 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.

## 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

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

Les types d'arité  $>0$  ne marchent pas très bien, en particulier les fichiers `queens2.purs`, `pascal.purs` et `ral.purs` ne typent pas

Normalement, tout le reste fonctionne.

## 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 `purscript_main.exe`. Et Make s'occupe de lancer dune, de renommer l'exécutable, et de lancer les tests.