TD3 : Analyse lexicale – *Tokenization*

2023-2024

Ce TD contient plusieurs exercices qui sont distinct du compilateurs et sont regroupés dans le dossier `td_3`. Seul le lexeur concernant le langage du cours (exercice 3) est à réaliser dans le dossier `compiler/course_language` (puisque ce sera le lexeur de notre compilateur final).

Exercice 1: Lexeur à la main

Dossier `td_3/hand_lexer`.

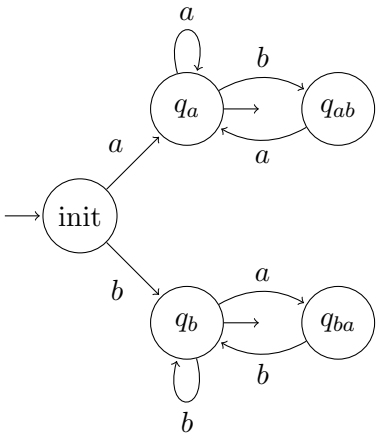
Dans ce premier exercice, on va regarder comment on peut faire un lexeur à la main, qui fonctionne de la même manière qu'un lexeur généré par un générateur. L'idée est d'avoir une structure représentant un automate et de l'exécuter sur une chaîne de caractères pour en déterminer le plus préfixe accepté par l'automate.

On se place dans un cadre simplifié (l'idée n'est pas de faire un lexeur équivalent à ce qu'on aura après, mais de voir que c'est faisable) :

- On considère directement des automates déterministes (en pratique, on peut faire des lexeur sur des automates non-déterministes)
- On ne considère pas de buffer
- On n'associe pas d'action aux états terminaux.

Le module `Automaton` contient un type représentant un automate déterministe, ainsi que des fonctions pour le manipuler, le module `Automata_examples` contient quelques exemples d'automates (que vous pouvez enrichir), le fichier `main.ml` génère l'exécutable `hand_lexer.out` qui vous permettra de voir l'effet de votre code, et enfin le module `Lexer` contiendra la fonction qui permet de réaliser l'analyse syntaxique d'une chaîne de caractères à partir d'un automate déterministe.

On considère l'automate suivant (`subject_example` dans le module `Automata_examples`):



1. Mise en jambe : donnez une expression régulière (simple) qui est acceptée par cet automate.

2. Donnez un exemple de mot accepté et de mot non-accepté (sur l'alphabet $\{a, b\}$).

3. On considère le mot $w = aabaabbabbabaaba$. Quel est le plus long préfixe de w qui est accepté ? Quels est le découpage qu'on obtient si on continue ce procéder (retirer le plus long préfixe du mot restant à chaque étape) ?

4. On va réaliser un programme qui exécute un automate déterministe sur une chaîne de caractères donnée en entrée, en tant que lexeur, c'est-à-dire qu'on ne veut pas savoir si une chaîne de caractères est acceptée, mais plutôt quel est le plus long préfixe qui l'est.

   Le module `Lexer` contient une fonction déterminant si un automate accepte une chaîne de caractère (`is_accepted`), qui vous est fournie (essentiellement pour vous faire observer que c'est pas dur). Vous devez implémenter la fonction `get_lexeme` qui prend en entrée un automate `automaton`, une chaîne de caractère `string` et un indice `start_pos` et qui doit renvoyer un couple `last_pos,state` qui vérifie que la sous-chaîne commençant en `start_pos` et terminant en `last_pos` est acceptée par `automaton` avec l'état `state` et qu'il n'existe pas de sous-chaîne plus longue commençant en `start_pos`.

   Attention `state` sera une option, car si le mot ne contient aucun préfixe accepté, il n'y a pas d'état final correspondant.

   L'idée de l'algorithme est d'exécuter l'automate comme si on cherchait à déterminer l'acceptation du mot, mais en retenant à chaque étape, le dernier état final vu et sa position, et de renvoyer cette information dès que soit on arrive à la fin du mot, soit qu'aucune transition n'est exécutable.

   Ce qui veut évidemment dire qu'on n'a aucun intérêt à avoir des automates complets !

5. Vérifiez votre implémentation en le testant sur quelques exemples. Vous pouvez changer l'automate utilisé par l'un de ceux présent dans `Automata_examples`. En particulier, l'exemple `arith_lexer` vous permet de découper des textes représentant des expressions arithmétiques (pas complètes), ce qui devrait vous montrer l'intérêt de ce genre d'approche dans un compilateur (de même que l'intérêt d'avoir une manière plus commode de donner nos automates).

Exercice 2: Introduction à Ocamllex

Dossier `td_3/simple_lexer`.

L'exercice précédent montre qu'on peut très bien simuler un automate déterministe par un programme et l'utiliser pour découper des chaînes de caractères. Cependant, quand on va vouloir exprimer un motif à reconnaître, cela sera en général beaucoup plus simple de fournir une expression rationnelle. Et donc pour appliquer la technique précédente, il faudrait transformer l'expression en automate, le déterminiser, puis produire le code précédent qui est assez fastidieux.

Cette tâche est en réalité automatisable, et c'est le but des générateurs de lexeurs tels qu'Ocamllex, dont vous pouvez trouver une rapide explication ici[1] ou une discussion

[1] `https://v2.ocaml.org/manual/lexyacc.html`

plus détaillée là[2]. Attention cependant dans ces document, le lexeur n'est pas séparé du parseur (que nous aborderons pour notre part qu'au prochain TD), ignorez donc les parties se référant à OCamlyacc ou Menhir.

L'idée de ce format est qu'on associe à une expression régulière une `valeur` (qui peut évidemment être une action), et qu'on peut avoir plusieurs expressions régulière ayant différentes valeurs (toutes de même type). Lorsqu'il est utilisé avec un parseur, ces valeurs seront d'un type appelé *token* qui représentent le *lexème* (ou unité de base) et qui peuvent (ou pas) contenir des informations à son sujet (typiquement, pour un identifiant, on voudra en connaître le nom).

Dans `simple_lexer`, vous avez plusieurs fichiers `.mll` qui sont les lexeurs correspondant (plus ou moins) aux automates que vous aviez dans `hand_lexer`. Ils vous sont donnés entièrement pour avoir un aperçu de la syntaxe, et de différentes fonctionnalités. Vous pouvez (devriez) les essayer avec l'exécutable `simple_lexer.out` généré par make. Ils ne s'utilisent que sur des chaînes de caractères fournis en argument du programme (pensez à encadrer la chaîne de p̈our pouvoir y inclure des espaces).

Tous ces lexeurs génèrent des données de même types (`int * int * string * string`) pour pouvoir s'utiliser dans le même exécutable. Néanmoins, il est évidemment possible de renvoyer n'importe quel type dans un lexeur (il faut juste que tous les cas d'une règle soient de même type). En particulier, dans un compilateur, un lexeur renverra un type utilisé par le parseur (appelé token).

Ces lexeurs illustrent aussi un certain nombre de fonction du module `Lexing` qui permettent d'obtenir des informations sur le buffer utilisé (ses positions dans la chaîne de caractère), ainsi que de le manipuler (saut de ligne). Ils illustrent aussi plusieurs cas possibles (et assez standard) pour traiter les erreurs, ignorer les espaces, etc.

`Subject_lexer` est un exemple relativement minimal sans fioritures.

`Ab_star_lexer` de même, mais il ignore les espaces et donne un message d'erreur plus complet.

`Even_odd_lexer` utilise en plus des définitions d'ensembles de lettres.

`Arith_lexer` utilise des disjonctions de cas pour une même règle.

`Decode_lexer` utilise plusieurs règles de lexing pour en faire des analyses différentes. Bon, ce n'est pas méga intéressant, mais essentiellement, il recopie les mots en mode normal, et s'il rencontre un `<`, il va ne garder que la première lettres des mots jusqu'au prochain `>`. Cette dernière utilisation revient à expliciter un automate pour un traitement qui serait compliqué avec uniquement des expressions régulières, et peut être parfois utile (par exemple, si l'automate est plus aisé à trouver qu'une expression régulière). En compilation, on à intérêt à se servir de ça pour gérer les commentaires sur plusieurs ligne tout en comptant correctement les lignes du fichier à analyser.

Dans les fichiers `.mll`, l'ordre de ces règles est important : il définit un ordre de priorité : si vous déplacez la règle en premier, votre programme renverra une erreur sur toutes les chaînes de caractères (du coup, on la met tout le temps en dernier).

Vous pouvez aussi regarder le code OCaml généré par le Ocamllex : il est dans `_build/default/simple_lexer/Subject_lexer.ml`. Bon, n'y passez pas trop de temps, c'est illisible, mais c'est pas le but de l'être.

Attention, la syntaxe des expressions régulière n'est pas celle vue en MPC, mais une syntaxe particulière (proche de la POSIX, mais pas tout à fait – référez-vous au manuel d'Ocamllex).

On insistera en particulier sur les points suivants :

[2] `https://dev.realworldocaml.org/parsing-with-ocamllex-and-menhir.html`

- On peut utiliser des caractères ou des strings
- $*$ désigne l'étoile de Kleene. + désigne l'itération non-vide.
- | est l'union de deux langage (au lieu du + habituel)
- ['a' 'b' 'c' 'e'] désigne l'un des caractère parmis ceux listés.
- ['a' - 'z'] désigne n'importe quel caractère entre 'a' et 'z' (dans l'ordre ASCII). Il peut être mélangé avec la notation précédente.
- [^ 'a' - 'z'] désigne n'importe quel caractère qui n'est pas entre 'a' et 'z'.
- _ désigne n'importe quel caractère.
- Vous pouvez nommer la chaîne de caractères reconnue par une expression (ou toute sous expression) grâce au mot-clef `as`. Par exemple si avec l'expression `a ((b)* as s)c` on analyse le mot abbbbc, s représentera la chaîne bbbb.
- Si une expression correspond à un cas que l'on peut ignorer, on rappelle la fonction de lexing sur la suite du buffer. Cela se fait simplement avec l'action `token lexbuf` (si la règle s'appelle `token`).

Ici, essentiellement, testez les lexeurs. Vous pouvez si vous le souhaitez vous amuser à les modifier. Le lexeur `Custom_lexer` est là pour tester ce que vous souhaitez. Il ne serait pas très compliqué d'en ajouter d'autres (mais il faut dans ce cas modifier `main.ml` et le fichier `dune`).

Exercice 3: Lexeur du langage du cours

Dossier `compiler/course_language`.

Dans cet exercice, on va produire un lexeur qui analyse des fichiers de code représentant des langage du cours.

Le code à compléter se trouve dans `compiler/course_language`, et l'exécutable permettant de l'appeler sur les exemples est `program_lexer.out`, (cet exécutable n'est pas à compléter).

On fournit un module appelé `Parser` qui contient une définition des tokens et une fonction d'affichage (et uniquement cela pour l'instant), cela sera à terme remplacé par un vrai parseur (avec les mêmes tokens : l'idée est que le fichier que vous produirez ici est le lexeur de notre langage).

Vous devez compléter `compiler/course_language/Lexer.mll` pour générer un lexeur qui reconnaît les éléments lexicaux du langage de programmation du cours.

Vous avez des exemples de programme dans `programs`. Dans `programs/examples_lexing` se trouvent des fichiers illustrant tous les cas possibles et pour lesquels l'exécutable comparera votre résultat avec ce qui est attendu (comportement que vous pouvez désactiver avec l'option `-no-test`). Tous les autres programmes présents sont bien sûr utilisables (mais non testables). Vous pouvez rajouter des fichiers, mais ne modifiez pas ceux de `programs/example_lexing` (ou les tests ne voudront plus rien dire).

On rappelle ici ces éléments lexicaux :

- Des mots clefs : `if`, `then`, `else`, `while`, `int`, `float`, `bool`, `var`, `print`, `null`, `size`
- Des opérateurs : $+ - * / \% \&\& || ! = <> <= >= < >$
- Des symboles de parenthésage : [ ] { } ( )

- Des symboles de structure : ; := ::= , .
- Un entier est une suite de chiffres ('0' - '9')
- Un flottant est une suite de chiffre ('0' - '9') suivi d'un point '.' suivi d'une suite de chiffres (qui peuvent être vides).
- Un booléen est soit `true` soit `false`.
- Un identifiant est une suite de caractères alphanumériques (lettres, chiffres, \ et _) commençant par une lettre ; et qui n'est aucun des mots-clefs précédents (ce qui se gère simplement avec l'ordre des règles).
- Une string est une chaîne de caractères entourée de ".
- Le caractère de fin de fichier eof.
- Les caractères d'espaces et de retour à la ligne sont ignorés.
- Les commentaires simples commencent par // et s'arrêtent à la fin de la ligne. Ils sont ignorés.
- Les commentaires sur plusieurs lignes commencent par /* et se terminent au premier */. Si vous voulez que le lexeur compte correctement les lignes dans le fichier, il est nécessaire d'avoir une règle supplémentaire qui gère les commentaire en ignorent tous les caractères, mais en n'oubliant pas d'incrémenter le compteur de ligne du buffer à chaque retour à la ligne (se référer aux exemples de l'exercice précédent).
- Les caractères d'espacements sont ignorés (pas de token) et les retours à la ligne doivent de plus incrémenter le compteur de ligne du buffer (cf point précédent).

N'oubliez pas de tester ce que vous obtiendrez avec le programme généré. C'est le point crucial de ce TD, ne passez à la suite qu'après avoir suffisamment avancé.

Exercice 4: Lexeur comme interpréteur

Dossier `td_3/calculette_lexeur`

Cet exercice vise à illustrer que dans certains cas (assez précis) d'utilisation, on peut utiliser un générateur de lexeur pour faire plus que simplement convertir une chaîne de caractère en séquence de token, c'est-à-dire directement traiter des données à la volée.

Le but de l'exercice est de réaliser un interpréteur de termes arithmétiques (sans variables) en notation préfixe (ou polonaise).

Vous pouvez voir que le main se contente de passer l'argument au lexeur et de récupérer et d'afficher le résultat (qui est un entier). Votre but est de compléter le lexeur de manière à ce qu'il accepte (et calcule le résultat) une expression arithmétique en forme préfixe :

- Il ignore les espaces.
- Si il reconnaît un entier (une séquence de chiffre), il renvoie l'entier correspondant.
- Si il reconnaît une opération (+, -, *, / ou %), il se rappelle deux fois de suite sur le buffer, et calcule ensuite le résultat correspondant qu'il reçoit.
- S'il atteint la fin du fichier, il renvoie une erreur
- S'il rencontre tout autre caractère, il renvoie une erreur

Exemple : + 1 * 7 2 donnera 15 (c'est-à-dire 1 plus (7 fois 2)).

Extension plus difficile :

Modifiez maintenant le lexeur obtenu de manière à ce qu'une chaine de caractère encadrée de < et > soit interprétée comme un nombre écrit en français textuel (sans les accents). Par exemple, si on a + <un deux zero> <trois quatre cinq>, cela devra être interprété comme si on avait la chaîne de caractère + 120 345 (et donc donner 465 comme résultat).

Pour cela, vous aurez besoin d'une seconde règle de lexing qui est appelée si le caractère est < et traite tous les chiffres en français pour reconstruire le nombre représenté jusqu'au > suivant (le plus simple est de construire la chaîne de caractères en chiffres puis de la traduire avec `int_of_string`).

Exercice 5: Détection de plagiat et sérialisation

Dossier `td_3/serialiser`.

L'utilisation principale d'un lexeur générant des tokens est d'être interfacé avec un parseur (ce qu'on fera au prochain TD). Mais on peut faire autre chose que cela avec la séquence de tokens.

Par exemple, la sérialisation consiste à le traduire en séquence d'octets la liste des tokens afin par exemple de compresser uniquement la partie signifiante du code (en éliminant les commentaires et normalisant les identifiants). Cela peut également permettre de détecter des codes qui sont strictement équivalent malgré des commentaires et des noms d'identifiants différents (et qui sont donc potentiellement plagiés l'un sur l'autre).

Il serait évidemment possible d'inverser le processus et de dé-sérialiser un fichier. Ce qu'on ne fera pas ici.

Si comparer les tokens suffira pour tous les mots clef, les opérateurs et les symboles de structure, ce n'est plus vrais pour les identifiants dont le nom est sans signification et la seule chose qui compte est qu'un même identifiant revienne aux mêmes positions. Pour cela, il faudra donc remplacer les noms des identifiants par leur numéro de première apparition dans le programme.

De même, il faudra encoder les constantes comme des séquences d'octets (ce point vous est fourni).

1. Écrivez la fonction `serialise_to_int_list` dans un premier temps. Cette fonction est celle qui sera appelée par l'option par défaut du programme (display) et sera utile pour les deux suivantes.

   Cette fonction doit d'abord ouvrir le fichier fourni en argument (grâce à `MenhirLib.LexerUtil.read`), et ensuite récupérer les tokens un à un grâce au lexeur et les convertir en séquence d'entiers entre 0 et 255 (pour qu'ils n'occupent qu'un octet).

   Pour les mots-clefs, vous devrez déterminer un code (arbitraire) associant à chaque mot-clef un entier. Si le token contient une donnée, cette donnée sera traduite en suivant (et aura un code de fin pour les données de taille variables) – exceptés les booléens, qui ne contenant que deux valeurs peuvent voir ces valeurs encodés directement comme si c'étaient des mots-clefs différents. Des fonctions vous sont fournies pour traiter les entiers, les flottants et les chaînes de caractères.

   Vous devrez codez vous-même le cas des identifiants, cas pour lequel une table de hachage et un compteur vous sera sans doute nécessaire. On pourra supposer pour simplifier (ce qui sera suffisant ici) que le programme dispose au maximum de 255 identifiants différents (et que donc un octet suffira à les encoder) – mais on pourrait évidemment en encoder un nombre arbitraire.

2. Implémentez ensuite la fonction `detect_plagiarism` (appelée par la tâche detect).

   Deux programmes seront strictement équivalents si les listes d'entiers ainsi retournées sont les mêmes pour ces deux programmes.

   Il devra permettre de déterminer que les programmes `plagiat_1.p` et `plagiat_2.p` sont strictement équivalents.

3. La sérialisation consiste à écrire les entiers ainsi produit comme des caractères dans un fichier texte. C'est le rôle de la fonction `serialise_to_channel` que vous pouvez finalement implémenter.

L'analyse de la ligne de commande pour les différentes étape de l'exercice vous est fournie.