TD 04 – Regex, Mise en place du projet, Analyse lexicale

1 Regex

Vous trouverez un rappel sur les Regex à la page suivante.

Exercice 1. *Écrire des Regex*

Sur la page Moodle du cours, dans un dossier appelé Ressources TD4 vous trouverez un fichier appelé **dico.txt** contenant environ 300000 mots de la langue française, un par ligne.
Pour tester vos réponses à cette exercice, sur linux ou macosx vous pouvez utiliser la commande grep suivante :

```
grep -E '^[aeiouy]+$' dico.txt
```

grep est un outil très pratique pour chercher des information dans un fichier (notamment grâce à des regex). L'option -E permet d'utiliser la même syntaxe de regex qu'en python ou avec lex. On met la regex entre deux quotes ′ (pour que le terminal n'essaye pas d'interpréter certains caractères). L'accent circonflexe et le dollar marque respectivement le début et la fin d'une ligne (ça permet de ne pas récupérer un "faux mot" à cheval sur deux lignes)
Alternativement, vous trouverez dans le même dossier un fichier **lire_dico.py** que vous pouvez placer dans le même dossier que **dico.txt**. Vous pouvez remplacer la regex indiqué à cette ligne :

```
ma_regex = r"^[aeiouy]+$"
```

par la regex de votre choix et tester avec

```
python3 lire_dico.py
```

1. Écrire une regex qui reconnait les mots suivants :
   - **(a)** Les mots composés uniquement de voyelles (sans accent).
   - **(b)** Les mots qui commencent par un z et finissent par un x.
   - **(c)** Les mots entre 23 et 25 lettres.
   - **(d)** Les mots avec au moins 3 v
   - **(e)** Les mots de taille multiple de 4 et qui alternent 2 consonnes, 2 voyelles, 2 consonnes, 2 voyelles,... Exemple : bleu, braillai,...
   - **(f)** Les mots avec 5 consonnes à le suite.
   - **(g)** Les mots qui commencent par un y et dont l'avant avant dernière lettre et un a.

Syntaxe Regex (python, lex,...)

Tableau avec la syntaxe la plus utile des Regex python. Pour la liste complète voir la documentation Python.

syntaxe Regex	Signification
rs	Concaténation des regex r et s
$r\|s$	Disjonction r et s (comme le $r + s$ en expressions régulières)
$r*$	Étoile de Kleen (comme r^* pour les expressions régulières)
$r\{n\}$	n occurrences de r (par exemple $r\{3\}$ est comme rrr)
$r\{n, m\}$	Entre n et m occurrences de r.
$r\{, m\}$	Au plus m occurrences de r (comme $r\{0, m\}$)
$r\{n, \}$	Au moins n occurrences de r
$r?$	Optionnalité : 0 ou 1 fois (comme $r\{0, 1\}$)
$r+$	Au moins 1 occurrences (comme $r\{1, \}$)
(r)	Parenthèses de priorité (par exemple les parenthèses dans $a(b\|c)$ indiquent qu'ici l'opération | est prioritaire sur la concaténation)
.	N'importe quel caractère sauf un retour à la ligne
$[abc]$	N'importe quel caractère entre les crochet (comme $(a\|b\|c)$)
$[a - z]$	N'importe quel caractère dans $\{a, b, c, \ldots, z\}$.
$[A - Z]$	N'importe quel caractère dans $\{A, B, C, \ldots, Z\}$.
$[0 - 9]$	N'importe quel caractère dans $\{0, 1, \ldots, 9\}$.
[^abc]	N'importe quel caractère sauf $\{a, b, c\}$.
\	Caractère d'échappement (si vous voulez utiliser un caractère spécial).
^	Début de ligne (ou seulement début de string selon l'outil)
$	Fin de ligne (ou seulement fin de string selon l'outil)

2 Mise en place du projet

Le projet est à réaliser en binôme, vous êtes donc prié de constituer des groupes de deux pour commencer à travailler sur le projet.

Le projet consiste à réaliser un compilateur du langage FLO (spécialement inventé pour ce cours) vers le langage assembleur. Sur la page du projet vous pouvez trouver un fichier `Sujet projet de compilation` qui fait une présentation du langage. Ce fichier sera probablement modifié selon les besoins ou vos demande de précision.

Toujours sur la page du projet, la pourrez trouver des instructions d'installation pour les différents logiciels dont vous aurez besoin pour la réalisation du programme. Il s'agit d'installer une implémentation de `lex` et `yacc` (la librairie `SLY` pour le python, `flex` et `bison` pour le `c`) pour réaliser l'analyse lexicale et syntaxique et un logiciel pour pouvoir lire les fichiers `.nasm` que votre compilateur va produire.
Une fois les logiciels installés vous pourrez récupérer l'archive `Projet python` si vous voulez travailler en `python` et `Projet c` si vous voulez travailler en `c`.
Décompressez cette archive qui contient une version rudimentaire du compilateur que vous allez réaliser. Si vous avez choisit le python votre archive doit contenir 2 dossiers et 6 fichiers.

- Le dossier `input/` qui contient des exemples de fichier `.flo` pour tester votre compilateur sur différents aspects. Vous êtes bien sûr invités à créer vos propres fichiers `.flo` pour vérifier que votre compilateur fait bien ce qu'on attend de lui (ce qui inclut de créer des fichiers avec des erreurs pour vérifier qu'il détecte bien l'erreur).
- Le dossier `output/` qui contiendra les codes exécutables correspondant à vos fichiers `.flo`. Pour le moment, il est vide.
- Le fichier `analyse_lexicale.py` qui permet de réaliser l'analyse lexicale d'un fichier source. En tapant (sur un terminal, en vous plaçant dans le bon dossier) :

  ```
  python3 analyse_lexicale.py input/exemple1.flo
  ```

  vous devez obtenir la liste des lexèmes sur le fichier `input/exemple1.flo`.

  ```
  Token(type='ECRIRE', value='ecrire', lineno=1, index=0, end=6)
  Token(type='(', value='(', lineno=1, index=6, end=7)
  Token(type='ENTIER', value=42, lineno=1, index=7, end=9)
  Token(type=')', value=')', lineno=1, index=9, end=10)
  Token(type=';', value=';', lineno=1, index=10, end=11)
  ```

  Cette commande vous permettra justement de tester votre analyse lexicale. Si ça ne fonctionne pas, retour à la partie installation.
- Le fichier `analyse_semantique.py` qui permet de réaliser l'analyse syntaxique (en faisant appel lui même appel à `analyse_lexicale.py`).

  ```
  python3 analyse_syntaxique.py input/exemple1.flo
  ```

  vous devez obtenir

  ```
  WARNING: Token(s) {INFERIEUR_OU_EGAL,IDENTIFIANT} defined, but not used
  WARNING: There are 2 unused tokens
  WARNING: 4 shift/reduce conflicts
  <programme>
   <listeInstructions>
    <ecrire>
  ```

```
    [Entier:42]
   </ecrire>
  </listeInstructions>
</programme>
```

La commande fait des avertissements car certains types de lexèmes définis dans l'analyse lexicale n'ont pas encore d'équivalent dans l'analyse syntaxique et aussi car la grammaire n'est pas correcte mais nous corrigerons ça pendant les prochaines séances. Cette commande vous permettra justement de tester votre analyse syntaxique.

— Le fichier `arbre_abstrait.py` qui fait la liaison entre l'analyse syntaxique et l'analyse sémantique/la génération de code.

— Le fichier `generation_code.py` qui réalise l'analyse sémantique/la génération de code `.nasm` en faisant appel aux fichiers précédemment définies.
La commande

```
python3 generation_code.py -nasm input/exemple1.flo
```

permet d'afficher le code -nasm correspondant au fichier `input/exemple1.flo` (précédé d'éventuels warning). Elle donne ceci.

```
WARNING: Token(s) {IDENTIFIANT,INFERIEUR_OU_EGAL} defined, but not used
WARNING: There are 2 unused tokens
WARNING: 4 shift/reduce conflicts
%include        "io.asm"
section .bss
sinput: resb    255     ;reserve a 255 byte space ...
v$a:    resd    1
section .text
global _start
_start:
        push    42
        pop     eax
        call    iprintLF
        mov     eax,    1       ; 1 est le code de SYS_EXIT
        int     0x80    ; exit
```

La commande

```
python3 generation_code.py -nasm input/exemple1.flo > output/exemple1.nasm
```

permet de mettre ce code dans un fichier `.nasm` (et éventuellement affiche des warning qui ne vont pas dans le fichier `.nasm`).

— Le fichier `io.asm` qui est une sorte de librairie pour le langage assembleur pour gérer les entrées/sorties. Elle est indispensable pour compiler vos fichiers `.nasm`.
Si vous tapez les commandes suivantes :

```
nasm -f elf -g -F dwarf output/exemple1.nasm;
ld -m elf_i386 -o output/exemple1 output/exemple1.o;
rm output/exemple1.o;
./output/exemple1
```

vous allez transformer votre fichier .nasm en fichier executable et l'executer.
Le programme `exemple1.flo` est

```
ecrire(42);
```

et votre programme affiche donc 42.

— Le fichier `Makefile` qui compile vos programmes à l'aide des commandes précédentes. Quand vous tapez

make

vous produisez un fichier exécutable `output/monProgramme` pour chaque fichier `input/monProgramme.flo` tel que `monProgramme` est placé dans `INPUT` à la ligne

INPUT = exemple1 exemple2

du `Makefile`.

3 Analyse Lexicale

Notre tâche du jour sera de compléter la partie "analyse lexicale" du code. Lisez attentivement le fichier `Sujet projet de compilation` sur la page du projet. Vous y trouverez une descriptions des différents éléments du langage.
Si vous travaillez en `Python`, vous n'aurais qu'à modifier le fichier `analyse_lexicale.py`. Si vous travaillez en `C`, vous devez modifier le fichier `analyse_lexicale.l` (flex) et analyse_syntaxique.y (bison). Dans le fichier `analyse_syntaxique.y`, vous devez ajouter une ligne

```
%token NOM_DE_MON_TOKEN
```

pour chaque nouveau type de lexème que vous allez créer que vous allez créer.
Dans le fichier `analyse_lexicale.l`, vous devez ajouter une ligne :

```
EXPRESSION_REGULIERE        { return NOM_DE_MON_TOKEN; }
```

pour chaque lexème que vous allez créer. Vous devez également modifier la fonction `nom_token` pour qu'elle affiche correctement vos lexèmes.
Vous pouvez tester votre analyse lexicale sur le fichier d'exemple `input/eval_lexicale.flo` comme expliqué dans la dernière section. Vérifiez que votre analyse repère bien les lexèmes composés de plusieurs symboles (comme $<=$ par exemple) et ne les interprète pas comme une série de lexèmes plus petit ($<$ puis $=$). Vérifiez aussi que les mots clés ne sont pas identifié comme des identifiants (noms de variables ou de fonctions).
Pour tester votre analyse lexicale en c :

```
make && ./main -l input/eval_lexicale.flo
```