APS TD 1 - Analyses Lexicale et Syntaxique: `lex` et `yacc`

R. Demangeon, P. Manoury

Rappels: L'**analyse lexicale** est la transformation d'un texte (**source**) en une séquence de mots (**lexèmes**, jetons). L'**analyse syntaxique** est la transformation d'une séquence de mots en une structure organisée (**arbre syntaxique**). L'analyse lexicale se base (souvent) sur les langages réguliers, l'analyse syntaxique sur les grammaires (souvent) non-contextuelles.

La théorie des analyses lexicale et syntaxique n'est pas l'objectif de ce TD, pour des cours et des exercices à propose de l'analyse, des langages réguliers et des grammaires, consulter les documents de l'UE **3I018 Compilation**.

Ce TD donne une introduction aux générateurs d'analyseurs `lex` et `yacc`.

Exercice 1 – Généralités

Question 1

Décider si les opérations suivantes font partie de la partie analyse **lexicale** ou **syntaxique**. Justifier

1. Trouver les arguments d'un appel de fonction.
2. Ignorer les commentaires d'un programme.
3. Distinguer = et ==.
4. Regrouper des calculs au sein d'une parenthèse.
5. Reconnaître un mot-clef du langage.
6. Décider si une commande est dans l'alternant d'une conditionnelle.
7. Regrouper des chiffres en un nombre.
8. Trouver le verbe dans une phrase en allemand.
9. Transformer l'expression arithmétique $-(-n)$ en n.

Pour écrire des analyseurs lexical et syntaxique, on peut utiliser (et on utilise souvent) des **générateurs d'analyseurs**. Ces programmes prennent en entrée une description du langage à analyser et produisent en sortie un analyseur.

Question 2

Donner un schéma des flots arguments/résultats d'un système décrivant l'analyse d'un programme en APS vers un arbre syntaxique en Prolog utilisant des générateurs d'analyseurs en C.

Les générateurs d'analyseurs `lex` et `yacc`, d'une importance historique certaine, sont couramment utilisés dans un cadre pédagogique. Nous nous intéressons ici à leur implémentation GNU/Linux en C (`flex`/`Bison`).

La référence à garder à portée de main est la documentation en ligne: `https://www.gnu.org/software/bison/manual/bison.html`

Exercice 2 – Utilisation de `lex`

Un fichier `lex` est composé:

- d'une en-tête `%{ ...}%` décrivant les instructions à inclure en début de programme, typiquement des déclarations de variables (et des inclusions en fonction de la nature des actions).
- d'une partie **définitions** composée d'expressions régulières utilisant (entre autres) les symboles:

  - `[]`: classe de caractères,
  - `^`: absence d'un caractère dans une classe,
  - `*`: étoile de Kleene (une, plusieurs ou zéro occurences),
  - `+`: une ou plusieurs occurences,
  - `?`: une ou zéro occurence,
  - `|`: choix, ...

- d'une partie **règles** (apres un `%%`) liant des définitions à des actions (code C).
- d'une partie **utilisateur** facultative (apres un `%%`) contenant du code C (recopié tel quel dans l'analyseur généré). Ce code appelle `yylex()`, point d'entrée de l'analyseur.

Question 1
Générer un analyseur lexical qui compte le nombre de mots d'une phrase (en français).

La variable `yyin` stocke le fichier (`FILE`) a analyser.

Question 2
Générer un analyseur lexical qui compte le nombre de mots d'un fichier texte.

La variable `yyleng` stocke le nombre de caractere de la derniere paire lue, la variable `yytext` stocke la chaine en question.

Question 3
Générer un analyseur lexical qui recupere le mot le plus long d'un fichier texte.

Le rôle d'un analyseur lexical est la production des **lexèmes** utilisés par l'analyseur syntaxique. Ainsi, la plupart des règles de l'analyseur lexical ont pour action `return JETON`. Une valeur peut être associée à ce lexème en modifiant la variable `yylval`. La règle fréquemment utilisée:

```
. return yytext[0]
```

permet de retourner (à l'analyseur syntaxique) tout caractère isolé non analysé (et ainsi de ne pas créer de jeton pour les parenthèses et certains symboles simples).

Question 4
Pour chacun des langages suivants, décider d'un jeu de lexèmes pertinent et générer un analyseur lexical.

- entiers unaires: $N ::= 0 \mid S(N)$.
- expressions booléennes: $b ::= \top \mid \bot \mid (b \wedge b) \mid (b \vee b) \mid (b \Rightarrow b)$.
- expressions arithmétiques: $e ::= n \mid (e + e) \mid (e - e) \mid -e \mid (e * e) \mid (e/e)$ avec n entier.
- lambda-termes de De Bruijn: $M, N ::= n \mid \lambda.M \mid M\ N$ avec n entier.

Exercice 3 – Utilisation de `yacc`

Un fichier `yacc` est composé:

- d'une partie **définitions** contenant, entre autres, la liste des lexèmes précédée de `%token`.

- d'une partie **règles** (apres un `%%`) décrivant la grammaire à l'aide de lexèmes et de caractères isolés (+, =, parenthèses). Chaque production est composée d'un membre gauche, suivi de `:` suivi d'un membre droit composé de plusieurs options, séparées par des `|`. Une option décrit une suite de lexèmes. On peut associer à chaque option de chaque production une action, cette dernière peut faire référence aux **valeurs** associés aux lexèmes avec les variables `$$, $1, $2, ...`

Question 1

Générer des analyseurs lexical et syntaxique permettant de calculer une **expression arithmétique** (cf. plus haut).

On peut bien sûr ouvrir un fichier depuis `yacc`, qui est le programme de haut-niveau: il appelle `lex`. Comme c'est ce dernier qui lit le fichier source, il convient de déclarer `yyin` comme une variable externe lors de l'écriture du code `yacc`.

Question 2

Générer des analyseurs lexical et syntaxique permettant de calculer la valeur d'une série d'instructions d'une **calculatrice à mémoire unique**.

Par exemple, l'analyse du fichier:

```
MP ((3 + 4) * 2)
MC
MP (4 * 2)
MP ((M + 1) * M)
AF (M - 1)
```

devra afficher la valeur `79`.

Parfois (mais pas toujours), lorsqu'un langage source permet l'utilisation de variables, l'analyse syntaxique crée **une table de symbole** contenant des informations sur les variables déclarées, mise à jour au fur et à mesure des déclarations. **Remarque:** L'analyse syntaxique n'a pas pour rôle la gestion du contenu de la mémoire, qui fait, bien sûr, partie de l'évaluateur.

Dans une analyse, il est possible que **le type** de valeur associé à un lexème change en fonction du lexème (valeur entière pour un nombre, chaine). Dans ce cas, il faut déclarer, dans l'analyseur syntaxique, la variable `yylval` comme un type union avec `%union`.

Question 3

Générer des analyseurs lexical et syntaxique permettant de calculer la valeur d'une série d'instructions d'une **calculatrice à variables**. Il faudra, exceptionnellement, passer outre la remarque précédente, et faire gérer les contenus des variables à l'analyseur syntaxique.

Par exemple, l'analyse du fichier:

```
MS saucisse 8
MS brouette (M saucisse + 2)
MS saucisse 2
MS brouette (M brouette * M saucisse)
AF M brouette
```

devra afficher la valeur `20`.

Exercice 4 – Pour s'entraîner

Question 1
Ecrire un analyseur syntaxique qui transforme un entier unaire en un entier décimal.

Question 2
Ecrire un analyseur syntaxique pour le lambda-calcul avec indice de De Bruijn.

Question 3
Ecrire un analyseur syntaxique pour les polynômes. En déduire un évaluateur simple de polynômes. Faire de même pour les polynômes multivalués.

Question 4
Ecrire un analyseur syntaxique pour les phrases de français, qui reconnait les groupes nominaux et donne la structure d'une phrase (sujet / verbe / complement). On pourra ensuite distinguer la transivité des verbes, et reconnaître les subordonnées.