TP Compilation - L3, S6
Grammaire VR et Langage de Requêtes sur Graphes à la SQL

L'objectif de ce TP c'est

1. reconnaître des graphes définis par une grammaire et de construire l'arbre de dérivation associé,
2. reconnaître des programmes écrits dans un langage de requêtes du style de SQL, de construire l'arbre de dérivation associé, et de générer du code à partir de l'arbre de dérivation,
3. interpréter le code sur l'arbre de dérivation associé au graphe.

Exercice 1.1 *Grammaire* VR

La grammaire VR est une grammaire qui permet de générer les graphes en donnant les couleurs aux sommets et en utilisant ces dernières pour générer les arêtes. Voici la grammaire que nous allons considérer qui va nous permettre d'affecter des termes à des variables et de les utiliser. Pour le moment, le langage est très simpliste et ne fait aucun saut à part le **where** qui présuppose que les variables utilisées sont déclarées. Les mots clé du langage, autres que les parenthèses et accolades, sont en gras. Les espaces ne sont pas importants. Ce n'est pas non plus important si vos programmes ne sont pas insensibles à la casse.

$$
\begin{aligned}
S &\rightarrow V \mid G \mid G \ \textbf{where} \ V \\
V &\rightarrow GID = G \ ; \ V \\
G &\rightarrow GID \mid (IN, ID) \mid IN \mid \textbf{SUM}(G,G) \mid \textbf{ADD\_}\{IN, IN\}(G) \mid \textbf{REN\_}\{IN, IN\}(G) \\
IN &\rightarrow digit+ \\
ID &\rightarrow letter\{letter + digit\}* \\
GID &\rightarrow [A - G]\{letter + digit\}*
\end{aligned}
$$

Vous avez des exemples de programmes valides dans les fichiers `graph1.vr` et `graphs.vr`. Vous avez les expressions régulières de **lex** dans le fichier `reg-lex.png`.

1. Ecrivez le code **lex** qui permet de faire l'analyse lexicale des programmes de notre grammaire VR.
2. Il ne suffit pas seulement de faire l'analyse lexicale, il faut aussi que l'on puisse faire l'analyse syntaxique et construire l'arbre de dérivation, tout en étant capable de faire remonter les petites erreurs. Des erreurs simples dans notre cas sont l'utilisation d'identificateurs non déclarés ou des oublis syntaxiques. L'objectif de cette question c'est d'utiliser **yacc** pour écrire l'analyseur syntaxique et en déduire l'arbre de dérivation du programme. L'utilisation de **yacc** est introduite en appendix.
3. Modifier le parser pour signaler les erreurs du type variable de graphes utilisé mais non déclaré.

4. On va appeler les mots générés par le non-terminal G des *termes*. On va supposer que le premier terme de nos programmes est le terme d'entrée. Ecrire un générateur de code qui prend en entrée l'arbre de dérivation de la question 2 et construit un arbre qui remplace chaque variable dans le terme principal par le terme qui le définit. Par exemple, pour le programme dans le fichier `graphs.vr`, le terme à générer est dans le fichier `graphs-sortie.vr`.

Exercice 1.2 *Langage de requêtes*
On va maintenant s'attaquer à la grammaire de requêtes sur les graphes et dont l'exécution utilise l'arbre de dérivation construire dans l'exercice précédent. La syntaxe de notre grammaire de requêtes est la suivante :

$$R \rightarrow \textbf{edge}(ID, ID) \mid RE(ID) \mid \backslash\textbf{Ex}\ ID.R \mid \backslash\textbf{All}\ ID.R \mid \backslash\textbf{neg}(R) \mid (R\ \textbf{or}\ R) \mid (R\ \textbf{and}\ R)$$
$$RE \rightarrow [A - Z]+$$

Pour éviter au départ toute difficulté, on a ajouté des parenthèses pour éviter les ambiguités. Vous les résoudrez dans un second temps en ajoutant des priorités dans votre parser en utilisant les directives %left et %right qui définit l'associativité gauche ou droite des terminaux créant des ambiguités. Ce langage est similaire aux requêtes **Xpath** de `XML`.

1. Ecrivez l'analyseur lexical des requêtes de notre langage.
2. Ecrivez l'analyseur syntaxique des requêtes de notre langage. Il faudra essayer de récupérer des erreurs simples du genre oubli de parenthèse ou virgule ou des points. Utilisez l'arbre de dérivation pour signaler des erreurs du genre un identificateur rencontré mal orthographié.
3. Modifiez la grammaire pour ajouter des règles du type $requete(x, y, z) = R$ où dans la requête R seules x, y et z ne sont pas quantifiées, et pour toutes les autres variables t on pourra trouver en remontant dans l'arbre de dérivation un ancêtre où la règle $\backslash\textbf{Ex}t.$ ou $\backslash\textbf{All}t.$ a été utilisée.
4. Pour la génération de code, il faut comprendre la sémantique associée à la grammaire VR et ensuite on définira la sémantique et le code associé aux requêtes. Le code de chaque requête est un automate. La génération de code se fait en construisant un automate inductivement. Il y a un automate pour chaque entier k. Si vous arrivez jusqu'ici, je vous donne rendez-vous dans mon bureau pour vous expliquer la sémantique. Avant, essayez de feuilleter les chapitres 1 et 2 du livre TATA `https://gforge.inria.fr/frs/?group_id=426`.
5. Vous pouvez maintenant écrire un programme d'interprétation qui prend en entrée l'automate de la question précédente et l'arbre de dérivation du graphe et exécute l'automate sur ce dernier.
6. Modifiez votre langage pour y ajouter des expressions conditionnelles, des affectations, des appels d'automates ou des boucles.
7. Vous savez maintenant écrire un langage de programmation car nous avons substitué les automates aux langages machines, et avons écrit un petit interprétateur qui interpréte l'automate sur les termes.

A Comment générer un analyseur syntaxique avec yacc

yacc est l'analyseur syntaxique qui va de pair avec **lex** et ce qui est décrit ici est son utilisation couplée à **lex**. Le gros du travail se fait dans la fonction yyparse qui utilise les lexèmes produits par l'analyseur lexical (dans notre cas la fonction principale de **lex** `yylex`), et il y a la possibilité d'utiliser la fonction yyerror(const char*) pour gérer les erreurs (soit essayer de les corriger, soit renvoie d'un message explicite à l'utilisateur pour lui demander de corriger son code). Vous comprendrez que ce qui est donné à **yacc** est la grammaire non-contextuelle et que les terminaux de cette grammaire vont correspondre aux lexèmes produits par yylex. Un fichier **yacc** se termine par **.y** et est composé, tout comme **lex**, de 3 parties :

```
declarations
%%
productions
%%
code additionel
```

Voici ce qu'il faut retenir pour écrire l'analyseur syntaxique :

1. Dans `declarations` : on y met toutes les directives `C` dont on a besoin dans les fonctions de traitement (macros, variables, etc.). Ces directives du `C` commencent par %{ et se termine par %}. Ensuite, on peut déclarer les terminaux (qui sont ceux produits par **lex**) et chacun précédé de `%token`. Par exemple `%token INTEGER`. On y déclare le token spécial `%start entree-programme` où entree-programme est notre non-terminal de départ. Dans notre grammaire, c'est le non-terminal S.
2. Dans `productions` : on y met les règles de notre grammaire. La syntaxe est la même que d'habitude, sauf que `:` est utilisée à la place de $\rightarrow$, et pour plus de lisibilité allez à la ligne pour chaque règle d'un non-terminal. Vous avez la possibilité de faire exécuter du code lorsque une règle d'un non-terminal est appliquée. Par exemple :

   ```
   E : nombre {code}
      | E P E {code}
   ```

   La partie {code} est la sémantique associée à chaque règle. La déclaration des règles d'un non-terminal est terminée par `;`.
3. Dans `code additionnel` : on y met les codes des fonctions que l'on utilise. Par exemple, la fonction de gestion des erreurs yyerror et la fonction main, cette dernière n'appelant par exemple que yyparse. Vous comprendrez que c'est ici que vous écrivez le code qui construit l'arbre de dérivation associé au programme.

Pour pouvoir écrire le code lors de l'application d'une règle il faut pouvoir accéder aux valeurs que l'on a donné aux différents terminaux et non-terminaux de la règle appliquée. Ainsi, $$ correspond à la variable pour la valeur du non-terminal se trouvant à gauche (celui avant le :), $1 pour le premier symbole à droite, $2 pour le deuxième, etc. Si pas de code, le code suivant est généré par défaut `\{$$=$1\}`. N'oubliez pas de déclarer la variable **yylval** comme variable externe, par exemple vous pouvez créer un fichier .h contenant les déclarations globales. Vous pouvez y mettre par exemple la déclaration de **yylval** en la déclarant comme variable externe. On peut par exemple écrire

```
#define YYSTYPE  mon_arbre
extern YYSTYPE yylval
```

On peut avoir des types différents par token.

Vous pouvez aussi récupérer des erreurs et essayer de les corriger. Donc, il faut ajouter des règles qui génèrent le terminal `error` et le code associé on traite l'erreur. Par exemple `G : règle` error code ; yyerrorok ;| Lorsque l'analyseur syntaxique exécute l'instruction `yyerrorok` il retourne à l'état juste avant l'erreur et recommence.

Je renvoie vers `https://www.gnu.org/software/bison/manual/bison.pdf` pour la documentation sur la version libre de **yacc**. La version libre de **lex**

`http://www-inf.it-sudparis.eu/COURS/CSC4508/Current/Documents/WebCoursTrad/Manual/flex-2.5.39.pdf` ou `https://westes.github.io/flex/manual/`.