Module EISE4 : Compilation

TPs 3 : Premiers pas avec Lex/Yacc, Expressions arithmétiques et Langage EISE

Ce troisième TP est consacré à l'étude et l'utilisation des générateurs d'analyseurs lexicaux (**Lex**) et syntaxiques (**Yacc**). La progression est aussi douce que possible et se fait en trois étapes, étape 0 à étape2. étape0 est la prise en main des outils **lex** et **yacc** au travers d'une première version de la grammaire EISE à réaliser. étape1 va consister à intégrer des règles grammaticales supplémentaires dans la grammaire fournie pour pouvoir exécuter une suite d'instructions **write** avec une constante réelle simple comme argument (exemple : write 1;). étape2 consiste à réaliser l'exercice classique de compilation avec des write prenant comme argument des expressions arithmétiques**,** pour afficher la valeur d'une expression à l'écran.

L'idée force de ce TP est de faire en sorte que votre compilateur EISE puisse générer le code assembleur correspondant au programme source EISE compilé par vos soins. Voyons ce que serait un résultat valide pour les deux dernières étapes :

Etape 1 :

ex1.eise

```
program ex1 ;
begin
        write 1 ;
end.
```

Le programme **ex1.eise** est composé d'une instruction **program** donnant le nom du programme, et d'un bloc principal contenant une unique instruction **write 1**...qui affiche 1 et qui rend le bloc **begin**...**end** grammaticalement correct. La compilation devrait donner comme code assembleur :

```
        pushr  1.000000
        ouput
        halt
        end
```

Etape 2 :

ex2.eise

```
program ex2 ;
begin
        write 2 * (3 + 1) ;
end.
```

Cette fois, le bloc principal contient une instruction **write** ayant une expression arithmétique comme paramètre, l'expression ayant elle-même une valeur (valeur=8). Votre compilateur devrait générer le code assembleur suivant :

```
pushr   2.000000
pushr   3.000000
pushr   1.000000
add
mult
output
halt
end
```

Il faut bien comprendre qu'utiliser les outils Lex et Yacc implique de nombreux allers et retours dans les différentes parties/fichiers du code, et que les erreurs produites par ces outils restent pour l'instant sybillines. Pour cette raison, il faut vraiment procéder avec méthode et suivre explicitement les consignes du texte de tp.

ETAPE 0 : Etat des lieux, Grammaire simple, mécanique générale

Dans le sous-répertoire **etape0**, vous trouverez la version initiale et opérationnelle du tp.

Vous reconnaîtrez :

evm.c, la machine virtuelle.
asm.c, l'assembleur.
vm_codops.h définissant la valeur des instructions.

eise.l, le fichier lex définissant les entités lexicales à reconnaître dans la grammaire EISE. Une fois passé par la commande flex, **eise.l** servira à créer le fichier **lex.yy.c**.

eise.y, le fichier yacc définissant la grammaire et les actions sémantiques associées à cette première version du langage EISE. Une fois passé par bison, **eise.y** servira à créer **y.tab.c** et **y.tab.h**.

ast.h et **ast.c**, fichiers C contenant des fonctions de création, de parcours de l'AST construit par les actions sémantiques contenues dans le fichier **yacc**.

ex0.eise, un fichier d'exemple, compilable en l'état.

res.asm, le fichier assembleur résultat généré par cette version du compilateur.
res.dot, fichier graphviz
res.png, fichier png

build, pour tout reconstruire.

Q1) dans le répertoire etape0, exécuter la commande ./build, pour tout reconstruire, et vérifier qu'il n'y a aucune erreur de compilation (du compilateur EISE écrit en c).

Q2) dans le répertoire etape0, exécuter la commande ./eise < ex0.eise, pour compiler votre tout premier programme EISE. Normalement, les fichiers res.asm, res.dot et res.png devraient être recréés. Les messages affichés prendront sens plus tard.

Q3) en examinant le contenu du fichier ./build, déterminer les commandes qui permettent de générer l'analyseur syntaxique du langage EISE et l'analyseur lexical. Lancer ces commandes à la main, dans l'ordre bison puis flex, déterminer quels sont les fichiers générés et vérifier que ceux-ci ont bien pour date de création la date courante (ils viennent d'être régénérés).

Etude de eise.y

Q4) la grammaire du langage est la pièce maitresse de notre compilateur. Editer le fichier eise.y, et déterminer l'étendue (# ligne de départ, # ligne de fin) des trois parties de ce fichier.

Q5) la partie 2 d'un fichier yacc contient les règles de la grammaire EISE, les productions. La première règle, program_eise, est la règle principale. Après examen de la grammaire, écrire un fichier ex01.eise qui ne donnera pas d'erreur à la compilation, puis un qui en produira une en tout état de cause.

Q6) Que fait exactement la règle skip_instruction ? Quelle est la fonction du code entre parenthèses ?

Q7) Que fait exactement la ligne 45 ?

Q8) Que fait exactement la ligne 46 ?

Q9) Est-ce que la ligne 42 est syntaxiquement logique ? Sémantiquement logique ?

Q10) A quoi sert l'union à la ligne 15 ?

Q11) A quoi servent les directives bison à partir de la ligne 25 ?

Q12) Que fait la ligne 36 ?

Q13) En résumé, quelles informations sont contenues dans la partie 1 du fichier eise.y ?

Q14) Que contient la partie 3 du fichier eise.y ?

Q15) Modifier les fichiers d'exemple **.eise**, et visualiser les **res.dot** pour vous convaincre de la réelle puissance de **bison**.

Etude de eise.l

Q16) Même exercice, quelles lignes définissent les parties 1, 2 et 3 du fichier lex ?

Q17) A quoi servent les lignes 10 à 13 du fichier **eise.l** ?

Q18) que représente **[a-zA-Z][a-zA-Z0-9_]*** ? A quoi sert le code entre **{}** ?

Q19) Même question pour tout le reste de la partie 2 du fichier **eise.l** ?

Q20) Que contient la partie 3 d'un fichier lex ?

ETAPE 2 : Gestion des expressions arithmétiques

Dans cette dernière étape, dans le répertoire **etape2**, il s'agit d'ajouter à la grammaire du langage EISE les règles permettant de manipuler les expressions arithmétiques. Pour cela, on va introduire une nouvelle instruction du langage EISE, l'instruction **write** qui prendra comme paramètre une expression arithmétique uniquement constituée de **numeric** et d'opérateurs mathématiques (pas de variables). Le fichier **ex2.eise** donne un exemple de programme EISE grammaticalement valide.

Q21) On souhaite que la grammaire des expressions arithmétiques soit la plus lisible possible, tout en sachant que cette grammaire est ambigüe (plusieurs AST possibles pour une même expression). La grammaire lisible suivante vous est donnée, mais **bison** ne peut en l'état la manipuler car tous les opérateurs ont la même précédence.

```
expr: T_NUMERIC
      | expr '+' expr
      | expr '-' expr
      | expr '*' expr
      | expr '/' expr
      | '(' expr ')'
      ;
```

Au même niveau que les directives%token ou%type, ajouter les lignes suivantes

```
%left '+' '-'
%left '*' '/'
```

va permettre de résoudre le problème. Ces lignes disent que l'associativité des opérateurs est « à gauche » et que les + et – sont moins prioritaires que * et /.

Q22) Ajouter tout ce qui est nécessaire à la grammaire pour pouvoir correctement construire les parties d'AST représentant les expressions arithmétiques. Cela implique d'ajouter les règles grammaticales nécessaires, les actions sémantiques nécessaires, les **%type** nécessaires et les **%token** nécessaires. On ajoutera également dans l'**enum** de **ast.h** tous les nouveaux types d'operateur, et on modifiera les fonctions de parcours de **ast.c** pour générer correctement l'assembleur et le fichier **.dot**. Au Texas Holdem poker, cette question ressemblerait à un « All in ».