I63- Compilation et théorie des langages
`Flex` et `Bison`

Licence 3 - 2018/2019

Consignes

Le rendu de TP devra se faire selon un des deux modes suivant:

1. un dossier par exercice, chaque exercices pouvant se compiler et s'exécuter indépendamment des autres.
2. l'interprète complet et fonctionnel (sans forcément la totalité des règles de grammaires telles que le `PRINT` ou la définition de fonction)

Le rendu se fera via moodle.

1 Utilisation conjointe de `Flex` et `Bison`

Il est possible d'écrire de puissant analyseurs syntaxiques avec `Bison` tout en sous-traitant la construction de l'analyseur lexical à `Flex`. Pour cela on commence par écrire l'analyseur syntaxique puis l'analyseur lexical chargé de reconnaitre les unités lexicales (tokens) définies par `Bison`.

Les fichiers `calc.y` et `analex.lex` fournissent un exemple élémentaire d'utilisation conjointe de `Flex` et `Bison`. La calculatrice est décrite dans le fichier `calc.y` et le programme `analex.lex` fournit à bison l'analyseur lexical dont il a besoin pour analyser une entrée. Le programme `Flex` récupère la définition des unités lexicales grâce au fichier `calc.h` qui sera produit par `Bison` lors de la compilation du fichier `calc.y` avec l'option `-d`. La compilation de la calculatrice se fait alors avec la séquence de commandes suivantes :

```
$ bison -o calc.c -d calc.y
$ flex -o analex.c analex.lex
$ gcc -Wall -o calc calc.c analex.c -lfl
```

1. Compléter les fichiers `calc.y` et `analex.lex` afin d'obtenir une calculatrice gérant:
   - les nombres entiers;
   - les identificateurs définis par une lettre majuscule;
   - les opérations `+,-,*,/,**,(,)` ainsi que le `-` unaire sur les entiers et l'affectation `=`;
   - une table de symboles de 26 cases.
2. Étendre la grammaire et le lexique pour gérer les opérateurs de comparaisons entre entiers: `==, !=, <, >, <=, >=`. On attribura les valeurs entières 0 et 1 aux valeurs de vérité `Vrai` et `Faux`.

2 Table de symboles

On souhaite étendre la gestion des identificateurs en acceptant toutes les chaînes alphanumériques d'au plus 256 caractères ne commençant pas par un chiffre. La table ne sera plus un simple tableau de 26 entiers mais une liste chainées basée sur la structure suivante

```
typedef struct table_symb{
  char *id;
  int val;
  struct table_symb *next;
    } table_symb;
```

1. Compléter les fichiers `ts.h` et `ts.c` chargés de la gestion d'une table de symbole dynamique.
2. Ajouter les lignes suivantes dans la zone de déclarations du programme `Bison`.

   ```
   % union {
     int nb;
     char id[256];
   }
   ```

   Cela signifie que désormais `yylval` est une union de deux types de données.

   Les instructions `%token <type> TOK` et `%type <type> non_term` permettent d'assigner respectivement le type `type` au token `TOK` ou au non-terminal `non_term`. Affecter le type correspondant aux token `NB, ID` et au non terminal `exp`.
3. Modifier l'analyseur lexical pour tenir compte de cette modification.
4. Modifier l'analyseur syntaxique pour gérer les identificateurs de plus d'un caractère.

3 Arbre de syntaxe abstrait

Afin de gérer des opérations plus complexes telles que les différents types de données, les structures de contrôle de flots ou les appels de fonction il est nécessaire de construire un arbre de syntaxe abstrait au lieu de simplement effectuer des calculs. L'évaluation d'un programme ce fait alors par l'évaluation de l'arbre.

1. Compléter les fichiers `asa.c` et `asa.h` afin d'évaluer les expressions arithmétiques simples.
2. Modifier le fichier `asa_test.c` pour que celui-ci construise l'arbre abstrait à partir d'une expression arithmétique postfixée rentrée par l'utililisateur à l'exécution.
3. Rajouter la gestion de l'affectation, des identificateurs d'une lettre et des différentes opérations de comparaison.

4 Programme complet

Le but de cette section est d'écrire un interprète de code en pseudo-C. Pour cela nous allons dans un premier temps modifier la grammaire à analyser. D'une part, le symbole
`n` va être ajouter aux blancs. De plus, un programme sera désormais une suite d'instructions qui elles-mêmes pourront prendre différences formes. Nous allons reprendre ici une partie de la grammaire du C: le caractère `;` servira de séparateur (token `SEP`) et les accolades de délimiteur de blocs (tokens `BO` et `BF`). Nous allons ensuite introduire au fur et à mesure la gestion de différentes structures: `if/else`, `while`, `print` ainsi que la gestion de fonctions.

```
    //Fragment de la grammaire
    //du Pseudo-C
PROG :
| PROG INST
;

INST : SEP
| EXP SEP
| PRINT EXP SEP
| BO INSTS BF
;

INSTS : INST
| INST INSTS
;
```

```
    //Fragment de l'analyseur
    //lexical correspondant

";"         { return ';'; }
"print"     { return PRINT;}
"{"         { return BO; }
"}"         { return BF; }

[ \t\n]     { /* ignorer les blancs */ }
```

Au lieu d'évaluer une expression arithmétique, le role des règles sémantiques consiste à construire l'arbre abstrait associé au programme, seules certaines productions conduiront à une évaluation de l'arbre.

Ainsi la règle

```
NB        { $$ = $1; }
```

sera remplacée par

```
NB        { $$ = create_nodeNb($1); }
```

De même la règle

```
EXP ADD EXP        { $$ = $1 + $3; }
```

sera remplacée par

```
EXP ADD EXP        { $$ = create_nodeOp('+', 2, $1, $3); }
```

1. Modifier les règles sémantiques et la grammaire du programme pour gérer la création de l'arbre abstrait.
2. Ajouter l'instruction `print` qui sera seule chargée de l'affichage de données.
3. Ajouter les instructions `if/else` sur le modèle du C.
4. Ajouter l'instruction `while` sur le modèle du C.
5. (optionnel) Ajouter la gestion de fonctions prédéfinies. Pour cela on définira un nouveau non-terminal `LIST_EXP` chargé de reconnaitre les liste d'expressions séparées par des virgules.
6. (optionnel) Ajouter la gestion de définition de fonction. On utilisera pour cela le mot clé `def` pour différencier la définition de l'appel d'un fonction.