On définit ici le langage `K` pour lequel on veut faire un compilateur : c'est un sous-ensemble simplifié du langage `C`, par exemple pas de pointeurs/malloc/passage par adresse, ni de struct, etc

On souhaite reconnaitre un document source en langage `K`, et générer du code assembleur correspondant (p.ex. x86-64). Le travail à faire est de définir des fichiers de spécifications `lexeur.l` et `parseur.y` pour flex et bison (voir le document ENT) et un fichier `main.c` qui ne fait que dire si l'analyse est réussie.

Les actions sémantiques devraient être utilisées pour construire une structure de donnée arborescente en `C` représentant l'arbre syntaxique du document source afin d'en faire une représentation intermédiaire (AST Arbre Syntaxique Abstrait). Comme cela sera indiqué en cours cette étape ne sera pas faite mais on générera à la place du code assembleur : en effet les actions sémantiques peuvent être faite pendant l'analyse syntaxique comme un parcours de l'AST. On utilisera un formalisme adhoc qui permettra la génération de code.

Fragment 0

expressions arithmétiques et booléennes sur les entiers

opérateurs (*seulement arithmétiques* pour ceux qui n'ont jamais fait de compilation/théorie des langages)

```
arithmétiques : + - *
comparaisons :  == < <=
logiques :      ! && ||
```

Pas de variables, pas d'appels de fonctions, pas de booléens

Fragment 1

Fragment précédent complété avec :

opérateurs

```
priorités et associativités des opérateurs
```

identificateurs

```
variables
fonctions
```

expressions arithmétiques et booléennes

```
entiers, variables, appel de fonctions
```

Fragment 2

Fragment précédent complété avec :

types

```
simple : int
```

instructions

```
vide : absence d'instruction
declaration de variables (sans affectation : pas int x=2;)
affectation : d'une expression dans une variable
        (uniquement d'une expression : pas a=b=4)
bloc d'instructions : délimité par des accolades
conditionnelle : if ( -valeur- ) -bloc- else -bloc-
boucle : while ( -valeur- ) -bloc-
```

Fragment 3

Fragment précédent complété avec :

instructions

```
retour : d'une expression
```

programme / sous-programmes :

- tous les sous-programmes sont des fonctions à résultat entier dont les déclarations sont globales.
- Une fonction est déclarée en indiquant son nom, ses arguments et le type de son résultat (c'est toujours un entier).
- Les arguments sont simples (des entiers) et leur passage se fait par valeur.
- Une fonction peux avoir des variables locales qui sont déclarées dans le corps de la fonction.
- On peut ignorer le résultat rendu par une fonction (comme en `C`).
- Un programme consiste simplement en une possible déclaration de variables globales, puis de fonctions et d'une fonction principale, nommée main(), qui est le point d'entrée dans le programme.
- Pour faciliter la grammaire, on refuse de déclarer `int a,b;` mais on accepte `int a, int b;`

programme / sous-programmes : version légère remplaçant certaines des phrases précédentes (pour ceux qui ont des difficultés avec la version `C`)

- Une fonction est déclarée en indiquant son nom et ses arguments, mais pas le type de son résultat (c'est toujours un entier).
- Une fonction peux avoir des variables locales qui sont déclarées en une liste avant le corps de la fonction.

Fragment 4

Fragment précédent complété avec :

table des symboles : ajouter les fonctions de création de table (et agrandissement de table dynamique), d'ajout d'une entrée dans la table et de recherche d'un élément. Vous pouvez utiliser vos propres fonctions, celles données ne sont là que pour ne pas perdre trop de temps.

```
contexte et adresses : des variables de main.c
tsymb : table de symboles dynamique, crée et affichée
déclaration de variables globales et locales, de fonctions : ajouté à la table
```

Fragment 5

Fragment précédent complété avec :

assembleur Fragment 2 (hors expressions et appel de fonction)

```
save, jump, branch
```

en utilisant une machine virtuelle à pile comme pour du WebAssembly p.ex. (cf. doc génération de code) ou en générant du x86-64 si vous êtes à l'aise. Pour simplifier on ne génére pas opcode/opérande dans un fichier exécutable, mais dans un tableau `mem[TC] = ...` du `main.c` : cela permet au besoin de réparer du code.

Fragment 6

Fragment précédent complété avec :

assembleur Fragment 1

```
arithmetic, logic, set less than, load
```

optionnel : assembleur Fragment 3

```
appel de fonction, retour
```

Quelques consignes

On définit ici le parseur sans arbre syntaxique abstrait (AST) ni génération de code qui a été utilisé pour commencer le Fragment 0 (pour ceux qui n'ont pu être présents au TP1 faute de rattrapage).

Remarque : La séparation lexeur/parseur est toujours un choix que vous devez faire. Ici on fait le maximum dans le parseur en utilisant par exemple directement les tokens (symboles terminaux) qui sont un simple caractère ascii (p.ex. `'+'`), et le lexeur n'est en apparence utilisé que pour les tokens non triviaux ; cela n'empêche pas que l'on passe toujours par le lexeur.

1. Créez trois fichiers :
   - `main.c` qui est le programme principal,
   - `lexeur.l` qui génère le lexeur,
   - `parseur.y` qui génère le parseur.
2. Dans `main.c`, on essaie de parser le document source et d'afficher si l'analyse syntaxique a réussie ou échouée :

```c
/* file main.c :: limited to yyparse() call and printed result */
/* compilation: gcc -o main main.c parseur.tab.c lex.yy.c */
/* result: main = syntaxic analysis */
/* usage: ./main < input.txt */

#include <stdio.h>   /* printf */
#include <stdlib.h>  /* exit */

int main(void)
{
  if (yyparse()==0) {  /* yyparse calls yylex */
    printf("\nParsing:: syntax OK\n");  /* reached if parsing folllows the grammar */
  }
  exit(EXIT_SUCCESS);
}
```

3. Dans `parseur.y`, on définit le parseur en utilisant des `char` comme token.

```c
/* file parseur.y */
/* compilation: bison -d parseur.y */
/* result: parseur.tab.c = C code for syntaxic analyser */
/* result: parseur.tab.h = def. of lexical units aka lexems */

%{
  int yylex(void); /* -Wall : avoid implicit call */
  int yyerror(const char*); /* same for bison */
%}

%token NOMBRE
%token O /* operator */
%start E /* axiom */

%%

E : E O E        /* binary operator */
  | '(' E ')'    /* well-parenthesized */
  | NOMBRE       /* recursive base case */
  ;

%%

#include <stdio.h>
int yyerror(const char *msg)
  { printf("Parsing:: syntax error\n"); return 1;}
int yywrap(void){ return 1; } /* stop reading flux yyin */
```

Dans l'ordre,

- on définit le prototype du lexeur et de la fonction d'erreur de bison
- on y décrit les tokens non triviaux (càd qui n'est pas un simple `char`) : `NOMBRE` et `O`
- on y décrit une grammaire avec expression comme symbole non terminal et avec `'('`, `')'` et les tokens `NOMBRE` et `O` comme symboles terminaux.
- on inclut des bibliothèques `C` pour pouvoir écrire le programme renvoyé en cas d'erreur,
- on y décrit la fonction `yyerror` qui est appelée en cas d'erreur et la fonction `yywarp` afin d'arrêter le parseur lorsque

4. Dans `lexeur.l`, on définit l'unique token non trivial :

```c
/* file lexeur.l */
/* compilation: flex lexeur.l */
/* result: lex.yy.c = lexical analyser in C */
```

```
%{
  #include <stdio.h>            /* printf */
  #include "parseur.tab.h"      /* token constants def. in parseur.y via #define */
%}

%%

0|[1-9][0-9]*    { printf("lex::NOMBRE %s\n",yytext); return NOMBRE; }
"+"|"-"|"*"      { printf("lex::O %s\n",yytext); return O; } /* double-quote ! */
[ \t\n]          { ; } /* ignore space, tab, and line return */
.                { printf("lex::char %s\n",yytext); return yytext[0]; } /* other one-char are

%%

/* nothing here */
```

Dans l'ordre :

- on inclut `parseur.tab.h` qui est généré à partir de `parseur.y` et qui définit le token `NOMBRE`,
- selon l'expression régulière, on retourne le token `NOMBRE`,
- la ligne `[ \t\n] { ; }` permet d'ignorer les séparateurs,
- la ligne `. return yytext[0];` indique que si on lit autre chose que les motifs précédents, on renvoit au parseur un token trivial avec ce caractère lu,

Vous pouvez faire un `makefile` des trois commandes suivantes dans le terminal :

```
$ bison -d parseur.y
$ flex lexeur.l
$ gcc -o analyse main.c parseur.tab.c lex.yy.c
```

Cela génère deux fichiers intermédiaires : votre parseur `parseur.tab.c`, votre lexeur `lex.yy.c`, puis votre exécutable `analyse`.

Vous pouvez exécuter `analyse` dans un terminal en lui redirigant dessus un fichier texte :

```
$ ./analyse < tmp.txt
```

Si ce contenu est correct vous aurez un message l'indiquant sinon vous aurez un message d'erreur.