# TP01 - ANALYSE LEXICALE ET SYNTAXIQUE AVEC SABLECC

## 1. Objectif

L'objectif de ce TP est de programmer un analyseur lexical et un analyseur syntaxique pour le langage $L$.
Les deux analyseurs sont produits automatiquement avec le logiciel `sablecc`, à partir d'un fichier de spécification. Ce fichier décrit les unités lexicales du langage, ainsi que ses règles de grammaire.
Vous trouverez un exemple de fichier de spécification dans les transparents du cours et une description complète du format des fichiers de spécification dans la documentation de Sablecc, que l'on peut trouver à l'adresse suivante :
`http://sablecc.sourceforge.net/thesis/thesis.html`.

## 2. Le langage $L$

Le langage $L$ est un langage de programmation minimaliste, inspiré du langage C. Voici ses principales caractéristiques :

**Types:** Le langage $L$ connaît deux types de variables :
- Un type simple : le type `entier`.
- Un type dérivé : les tableaux d'entiers, déclarés avec des crochets `[ ... ]`.

**Opérateurs:** Le langage $L$ connaît les opérateurs suivants :
- arithmétiques : `+`, `-`, `*`, `/`
- comparaison : `<`, `=`
- logiques : `&` (et), `|` (ou), `!` (non)

**Instructions:** Le langage $L$ connaît les instructions suivantes :
- Bloc d'instructions, délimité par des accolades `{ ... }`
- Affectation `a = b + 1;`
- Instruction `si` *expression* `alors { ... }` et
  `si` *expression* `alors { ... } sinon { ... }`
- Instruction `tantque` *expression* `faire { ... }`
- Instruction `retour` *expression* `;`
- Instruction d'appel à fonction `fonction(` *liste d'expressions* `);`

**Fonctions:** un programme $L$ est une suite de fonctions, parmi lesquelles la fonction principale, appelée, comme en C : `main`
- Ce sont des fonctions à résultat entier.
- Le passage des arguments se fait par valeur.
- Les fonctions possèdent des variables locales.
- Une fonction ne peut pas être déclarée à l'intérieur d'une autre.
- On peut ignorer le résultat rendu par une fonction.

**Fonctions prédéfinies:** Les entrées-sorties de valeurs entières se font à l'aide de deux fonctions prédéfinies, `a = lire();` et `ecrire(a);`.

Voici un exemple d'un programme en $L$ :

```
f(entier a, entier b)    # déclaration d'une fonction à deux arguments
entier c, entier k;      # déclaration de deux variables locales
{                        # début d'un bloc d'instruction
  k = a + b;             # affectation et expression arithmétique
  retour k;              # valeur de retour de la fonction
}                        # fin du bloc d'instruction
main()                   # point d'entrée dans le programme
entier d;
{
  d = f(d, 2);           # affectation et appel de fonction
  ecrire(d + 1);         # appel de la fonction prédéfinie ecrire
}
```

## 3. Analyse lexicale

3.1. **Unités lexicales.** On distingue 5 types d'unités lexicales :
- les symboles simples (`+`, `;`, `(`, `{`, ...),
- les mots-clefs, dont :
  - les instructions de contrôle (`si`, `alors`, `retour`, ...),
  - les types (`entier`),
  - les fonctions spéciales `lire` et `ecrire`,
- les noms de variables et de fonctions, et
- les nombres, uniquement entiers.

Les unités lexicales de type symboles simples et mots-clefs constituent des **classes fermées**, on peut en faire l'inventaire exhaustif.
Les unités lexicales de type noms de variables et de fonction ainsi que les nombres constituent des **classes ouvertes**, on ne peut pas en faire l'inventaire. Cependant, on peut les décrire à l'aide de contraintes de forme (par exemple : un nombre entier est une suite de chiffres).

3.2. **Identificateurs et nombres.** Un identificateur est une suite de caractères qui n'est pas un nombre ni un mot-clef. Il s'agit d'une suite de caractères contenant des lettres non accentuées majuscules ou minuscules, des chiffres, et les symboles dollar ($) ou underscore (_). Un identificateur commence toujours par un caractère différent d'un chiffre (c'est-à-dire une lettre majuscule, minuscule, dollar ou souligné). Un identificateur de fonction et de variable peut avoir un seul caractère (y compris seulement un dollar ou souligné). De plus, un identificateur ne peut pas être identique à un mot-clef du langage $L$. Par exemple, on ne peut pas déclarer une fonction qui s'appelle `tantque` ou `ecrire`. Le langage $L$ est sensible à la casse.

3.3. **Commentaires et blancs.** Les unités lexicales peuvent être séparées les unes des autres dans le programme source par un nombre quelconque de blancs, caractères de tabulation (`\t`) et retours chariot (`\n`). Tous ces caractères doivent être ignorés. C'est l'analyseur

lexical qui doit s'occuper de supprimer les espaces. Cependant, l'analyseur doit s'en servir pour séparer les unités lexicales. Par exemple, il est obligatoire de mettre un (ou plusieurs) espaces dans la déclaration d'une variable, entre le mot-clef et le nom, par exemple, `entier max`. Cependant, dans une expression comme `max = 1;` les espaces sont optionnels car le caractère `=` ne peut pas faire partie de l'identificateur.
Il est possible de rajouter des commentaires d'une ligne. Ces commentaires commencent par un caractère dièse (`#`) et se terminent à la fin de la ligne. Le caractère dièse et tout ce qui suit jusqu'à la fin de la ligne (`\n`) doit être ignoré par le compilateur. Il ne doit rester aucune trace des commentaires au niveau de l'analyseur syntaxique. Notamment, les commentaires ne sont mentionnés nulle part dans la grammaire.

## 4. Analyse syntaxique

L'objectif de l'analyseur syntaxique est de vérifier d'une part que le programme analysé est correct et, s'il l'est, de produire un arbre de dérivation du programme. Nous utiliserons comme point de départ la grammaire hors contexte construite en TD. Vous écrirez les règles de la grammaire au format Sablecc, puis vous utiliserez le logiciel `sablecc` pour générer un analyseur syntaxique $LR$ automatiquement.

## 5. Intégration

Les unités lexicales reconnues par l'analyseur lexical sont fournies à l'analyseur syntaxique. L'interfaçage entre les deux est géré automatiquement par le code généré par `sablecc`. L'analyseur syntaxique est ensuite appelé par votre compilateur à travers la méthode `parse`, comme dans l'exemple suivant :

```
public class Compiler{
 public static void main(String[] arguments){
  try{
   // Création des analyseurs lexicaux et syntaxiques
   Parser p =
    new Parser(
    new Lexer(
    new PushbackReader(
    new InputStreamReader(System.in), 1024)));

   // analyse du programme lu sur l'entrée standard
   Start tree = p.parse();
  }
  catch(Exception e){
   System.out.println(e.getMessage());
  }
 }
}
```

## 6. GIT

Le squelette du projet se trouve dans un dépôt `etulab` à l'adresse suivante :
`https://etulab.univ-amu.fr/nasr/2022_compilation`
Vous devez créer un "fork" du dépôt, à partir de l'interface web de `etulab`.
Vous DEVEZ donner à votre fork le nom suivant" :
`2022_compilation_X_Y`
où `X` est le nom du premier membre du binôme et Y le nom du second membre du binôme, par exemple
`2022_compilation_macron_castex`
Votre fork doit être privé et vous devez ajouter comme membre en qualité de Maintainer votre responsable de TP.
Ensuite vous pourrez faire un clone du "nouveau" dépôt créé :
`git clone https://etulab.univ-amu.fr/monId/2022_compilation_X_Y.git`
où `monId` est votre identifiant `etulab`
Pour donner le nom `squelette`, par exemple, au dépôt d'origine afin de garder le lien avec lui, on fait :
`git remote add squelette https://etulab.univ-amu.fr/nasr/2022_compilation`
Ensuite, lors des TP suivants, vous pourrez mettre à jour votre dépôt en faisant :
`git pull squelette master`
Vous pourrez utiliser le dépôt que vous avez créé de la même façon que d'habitude.

## 7. Ce qu'il faut faire

(1) Ecrire l'analyseur lexical et l'analyseur syntaxique sous la forme d'un fichier de spécification Sablecc.
(2) Générer l'analyseur à l'aide du programme `sablecc`.
(3) Faire tourner l'analyseur sur les exemples de `2022_compilation/test/input`.
(4) Afficher l'arbre de dérivation à l'aide du visiteur `Sc2xml.java`.