ENSICAEN
ÉCOLE NATIONALE SUPÉRIEURE D'INGÉNIEURS DE CAEN
& CENTRE DE RECHERCHE

Compilateur et théorie des langages

2I2AB2
Regis.Clouard@ensicaen.fr
Xavier.Desquesnes@unicaen.fr

Travaux pratiques
Interpréteur d'expressions ensemblistes

1. Présentation générale du projet

On désire réaliser un **interpréteur** d'expressions ensemblistes en utilisant les outils Lex et Yacc. Les opérations ensemblistes prises en compte sont l'union, l'intersection et le complémentaire. Voici un exemple d'exécution de l'interpréteur :

```
A := { }                  met l'ensemble vide dans A
A := {1, 22}              met l'ensemble { 1, 22 } dans A
B := A union {3}          met l'union de A et du singleton { 3 } dans B
B                         affiche l'ensemble B : {1, 3, 22}
C := {1, 3, 5}            met l'ensemble {1, 3, 5} dans C
A := comp C               met le complémentaire de C dans A
A                         affiche l'ensemble A : {2,4,6,7,8,9,...}
|B|                       affiche le nombre cardinal de B : 3
{1,2} union {3}           affiche {1,2,3}
{1} union B inter {3}     affiche {1,3}({1} = union (B inter {3})
A := B inter {3}          affiche {3}
A                         affiche {3}
```

Le travail se fera en considérant deux cycles itératifs dans le but de construire deux versions successives (contenue dans deux dossiers différents) : une version minimale de l'interpréteur, puis une version intégrant des fonctionnalités plus sophistiquées présentées en section 5.

La conception de la première version se fera en trois étapes :

1. Programmation d'un analyseur lexical d'expressions ensemblistes (uniquement le *scanner*). Il devra **afficher** chacune des **unités lexicales** reconnues lorsque l'on tape une expression ensembliste.
2. Programmation d'un analyseur syntaxique d'expressions ensemblistes (uniquement le *parser*). L'analyseur devra **afficher** la liste des **symboles** reconnus lorsque l'on tape une expression ensembliste valide ou non.
3. Programmation de l'interpréteur complet à partir de l'analyseur sémantique qui lit une expression et l'exécute.

2. Étape 1 : Analyseur lexical

Pour des raisons pédagogiques, la réalisation de l'analyseur lexical est soumise aux deux contraintes :

- La compilation du fichier Lex résultant devra se faire en C (donc avec gcc).
- Le fichier contenant l'analyseur lexical devra être suffixé avec « .lex ». Puisque ce suffixe n'est pas un standard reconnu par la commande `make` de Linux, il faut donc ajouter cette extension dans la liste des suffixes du makefile si vous voulez utiliser les règles implicites de compilation.

L'analyseur lexical de l'interpréteur prend en compte les restrictions suivantes :

- Les identificateurs correspondent aux noms des ensembles. Les ensembles sont uniquement nommés à partir d'une lettre et il n'y a pas de différence entre les lettres majuscules et minuscules.
- Les ensembles sont décrits par les délimiteurs '**{ }**' et les éléments à l'intérieur sont des entiers entre 1 et 63 séparés par des virgules.
- Le nombre cardinal d'un ensemble A est noté **|**A**|**.
- Les opérateurs ensemblistes s'écrivent '**union**', '**inter**' et '**comp**' et sont insensibles à la casse. Union et inter sont des opérateurs binaires tandis que comp est un opérateur unaire.
- L'opérateur d'affectation s'écrit '**:=**'.
- Une expression se termine par une fin de ligne '**\n**'.
- Toutes les erreurs détectées au niveau lexical devront faire appel à la fonction C `printError()` que vous devez écrire comme affichant le message d'erreur. La signature de la fonction est caractérisée un numéro d'erreur (arbitraire) et une liste de chaînes de caractères dont la première est forcément non nulle et la dernière est forcément NULL :

  ```
  void printError(int erreurNumber, char *string1, …)
  ```

  Voici un exemple d'appel (avec la variable globale `char tmp[1024]`) :

- ```
  sprintf(tmp, "Error: '%c' (%d): illegal character", yytext[0], yytext[0]); printError(1, tmp, NULL);
  ```

- Il est nécessaire de faire vos tests en ajoutant un `main()` dans le fichier de l'analyseur qui lit une expression booléenne sur l'entrée standard et affiche les unités lexicales reconnues ou un message d'erreur à chaque erreur constatée.

Le correcteur attachera une attention particulière à la qualité du fichier makefile et notamment au nombre de règles utilisées.
  ```

3. Étape 2 : Analyseur syntaxique

La compilation de l'analyseur syntaxique se fera avec un compilateur C++ pour profiter de la représentation par classes des éléments de l'interpréteur, mais la partie analyseur lexical devra toujours être compilée en C.

Écrivez la grammaire pour reconnaître toutes les formes correctes d'expressions ensemblistes. Ce programme devra simplement déterminer si une expression donnée est correcte ou incorrecte.

Toute erreur détectée devra faire appel à la fonction `printError()` écrite à l'étape 1.

4. Étape 3 : Analyseur sémantique et exécution

La construction de l'analyseur sémantique est régie par les contraintes suivantes :

- La table des symboles est représentée simplement par un tableau de 26 entiers longs non signés. La case 0 correspond à la valeur de l'ensemble A (les 64 valeurs possibles), la case 1 à celle B, …
- Dans cet interpréteur, on ne considère que les ensembles d'entiers compris en 1 et 63. Ceci permet de représenter un ensemble par un entier très long (`unsigned long long` : 64 bits – Attention ce type n'est pas encore portable sur tous les compilateurs). Chaque bit mis à 1 dans l'entier signifie que l'élément correspondant est présent dans l'ensemble. Par exemple, la valeur 0 correspond à l'ensemble vide {} et la valeur 5 (qui est codé par le nombre binaire 0000 0000 0000 0000 0000 0000 0000 0101) correspond à l'ensemble {1, 3}.
- Compte-tenu du choix de représentation des ensembles, les opérations ensemblistes devront être directement implémentées par les opérations bit à bit du C.

Cette fois l'analyse sémantique d'une expression doit permettre d'exécuter cette expression et d'afficher le résultat si elle est correcte.

Toute erreur détectée devra faire appel à la fonction `printError()` écrite à l'étape 1.

5. Fonctionnalités complémentaires

La conception de cette nouvelle version doit se faire dans un nouveau dossier distinct du précédent.

Voici une liste des fonctionnalités à ajouter :

1. Compléter la grammaire précédente pour prendre en compte les divers types d'erreur possibles dans l'écriture des expressions et afficher en conséquence un message d'erreur informant. Par exemple, l'expression A:=|A| pourrait afficher le message d'erreur :

`"Impossible d'affecter une valeur numérique à un ensemble."`

2. Utiliser des identificateurs d'ensemble de plus d'une lettre et qui soit toujours insensibles à la casse.
3. Utiliser des ensembles de plus de 64 éléments.
4. Ajouter le test d'égalité entre deux ensembles, avec le symbole '=' : A=B.
5. Ajouter le test d'inclusion entre deux ensembles, noté '**in**' : A in B.

6. Livrables

- un dossier contenant la **distribution source complète** de l'interpréteur divisé en deux sous-dossiers, un pour chacune des deux versions. Le dossier doit être construit de telle manière qu'il suffit d'effectuer les deux commandes suivantes pour utiliser le logiciel :

```
$ make
$ ./set_interpreter < fichier_test.txt
```

  où `fichier_test.txt` contient un exemple d'instructions ensemblistes.
- un document texte (README.odt ou README.html) décrivant les fonctionnalités implémentées et la modélisation UML des classes utilisées.