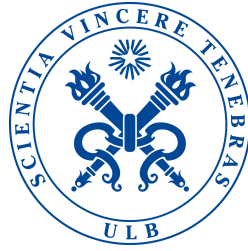


UNIVERSITÉ LIBRE DE BRUXELLES



INFO-F403 - INTRODUCTION TO LANGUAGE THEORY AND
COMPILING

Rapport 1ère Partie

Auteurs :

Yasin ARSLAN

Jacky TRINH

Titulaires :

Gilles GEERAERTS

Assistants :

Marie VAN DEN BOGAARD

Leo EXIBARD

Table des matières

1	Présentation	2
2	Implantation	2
2.1	Classes	2
2.1.1	LexicalUnit	2
2.1.2	Symbol	2
2.1.3	LexicalAnalyzer	2
2.1.4	Main	2
2.2	Expressions régulières	2
3	Exécution & Makefile	3
4	Tests	3
5	Conclusion	4
5.1	Bonus	4

1 Présentation

Dans le cadre du cours INFO-F403, il nous a été demandé d'implanter un compilateur pour le langage IMP. Ce projet est divisé en 3 parties. Cette première partie consiste en l'implantation d'un analyseur lexical reconnaissant la grammaire du langage IMP. Pour ce faire, nous avons utilisé l'outil JFLEX qui permet de générer l'analyseur pour Java.

2 Implantation

2.1 Classes

2.1.1 LexicalUnit

Le *LexicalUnit* est en réalité un *enum* plutôt qu'une *classe*. Il contient tous les mots clés nécessaires au langage IMP.

2.1.2 Symbol

Elle représente un *token* reconnu par l'analyseur lexical. Elle contient le type de *token* reconnu, la ligne ainsi que la colonne où se trouve le *token* et également sa valeur.

2.1.3 LexicalAnalyzer

Cette classe se chargera d'analyser un code et de reconnaître les *tokens* en les identifiant grâce aux mots clés contenus dans le *LexicalUnit* et créera les *Symbols* afin de pouvoir les utiliser dans une partie postérieure du projet. Pour l'instant, elle scanne toutes les lignes en affichant les *tokens* suivis de leur type. Pour finir, elle affiche, dans un ordre alphabétique, tous les *identifiers* ainsi que de la ligne dans laquelle ils ont été rencontrés pour la première fois.

2.1.4 Main

Une simple classe principale qui, pour le moment, appelle le *LexicalAnalyzer*.

2.2 Expressions régulières

Les expressions régulières suivant ont été utilisées afin de reconnaître les noms de variable et les nombres :

- $ALPHA = [a-zA-Z]$: représente n'importe quel caractère alphabétique.
- $NUM = [0-9]$: représente n'importe quel chiffre entre 0 et 9.
- $NUM_1_9 = [1-9]$: représente n'importe quel chiffre entre 1 et 9.

- $VARNAME = \{ALPHA\}(\{ALPHA\} / \{NUM\})^*$: une variable commence par une lettre suivi d'un nombre arbitraire de lettres et/ou de chiffres.
- $NUMBER = \{NUM_1_9\}\{NUM\}^* / 0$: un nombre commence par un chiffre entre 1 et 9 suivi d'un nombre arbitraire de chiffres entre 0 et 9 ou bien c'est tout simplement un 0.

Concernant les mots clés reconnus grâce au *LexicalUnit*, les expressions régulières appropriées sont simplement les *tokens* qui les correspondent.

Par exemple, une assignation correspondrait à `:=`, un IF correspondrait à un `if`, etc...

Afin de ne pas être redondant dans ce rapport, nous éviterons les détails.

3 Exécution & Makefile

Afin de pouvoir exécuter notre analyseur lexical, il faut se situer dans le dossier `/dist/` et rentrer la commande suivante dans un terminal : `java -jar LexicalAnalyzer.jar source-File` où le *sourceFile* est supposé être un fichier *.imp*.

Nous avons également créé un makefile afin de rendre les choses beaucoup plus simple. Pour l'utiliser, il faut se rendre dans le dossier `/more/`.

- `make` permet de compiler le projet.
- `make launch` permet de tester les fichiers *.class* avec le fichier d'input *Euclid.imp* provenant de l'Université Virtuelle.
- `make jar` permet de créer un fichier *jar* dans le dossier `/dist/` ainsi que de l'exécuter avec le fichier de base *Euclid.imp*

4 Tests

Ayant le même résultat que ce qui était attendu par le fichier *Euclid.out*, nous avons également voulu créer un nouveau fichier suivant la grammaire du langage **IMP** afin de pouvoir tester notre analyseur lexical. Ce fichier se nomme *Test.imp* et se trouve dans le dossier `/test/`. Chaque ligne est séparée par une ligne afin de mieux visualiser les informations.

```
[raina@raina-pc dist]$ java -jar LexicalAnalyzer.jar ../test/Test.imp

token: begin      lexical unit: BEGIN
token: x           lexical unit: VARNAME
token: :=          lexical unit: ASSIGN
token: 0           lexical unit: NUMBER
token: ;           lexical unit: SEMICOLON

token: for         lexical unit: FOR
token: i           lexical unit: VARNAME
token: from        lexical unit: FROM
token: 0           lexical unit: NUMBER
token: by          lexical unit: BY
token: 1           lexical unit: NUMBER
token: to          lexical unit: TO
token: 10          lexical unit: NUMBER
token: do          lexical unit: DO

token: x           lexical unit: VARNAME
token: :=          lexical unit: ASSIGN
token: x           lexical unit: VARNAME
token: +           lexical unit: PLUS
token: 2           lexical unit: NUMBER
token: ;           lexical unit: SEMICOLON

token: done        lexical unit: DONE
token: ;           lexical unit: SEMICOLON

token: print       lexical unit: PRINT
token: (           lexical unit: LPAREN
token: x           lexical unit: VARNAME
token: )           lexical unit: RPAREN

token: end         lexical unit: END

----- Identifiers -----
i         4
x         3
-----
```

FIGURE 1 – Affichage des tokens

5 Conclusion

Après cette première partie, nous sommes loin d'un compilateur révolutionnaire. Nous avons analysé et codé une première partie de ce compilateur. Celle-ci permet d'analyser un code donné et de retourner tout les *tokens* reconnus. La prochaine étape consistera en l'analyse de la cohérence de la syntaxe ainsi que de la grammaire. Nous n'avons pas rencontré d'énorme difficulté à coder cette partie, si ce n'est le premier pas avec JFLEX.

5.1 Bonus

La difficulté rencontrée dans les commentaires imbriqués est de savoir à quel moment devons-nous quitter l'*état* concernant les commentaires. En effet, nous entrons dans l'*état* des commentaires lorsque nous rencontrons un $(^*$ et nous en sortons lorsque nous rencon-

trons un $*$). Mais que se passerait-il si nous avions trois ($*$ suivi de trois autres $*$) ? Nous devrions donc faire en sorte que nous quittons l'*état* des commentaires après la troisième $*$) et pour ce faire, à l'aide de JFLEX, nous pouvons facilement créer une variable compteur qui s'incrémentera à chaque ($*$ rencontrée et qui décrémentera également à chaque $*$) rencontrée afin de savoir quand pourrons-nous quitter l'*état* des commentaires (à savoir lorsque ce compteur est à 0)