Haldenwang Thibault

Arzul Paul

Compilateur

Assembleur à Java

# But

Le code assembleur étant souvent un code peu explicite pour les programmeurs habitués aux langages haut niveau, l’idée est venue de compiler un code assembleur en code dit compréhensible tel que le Java.

Le compilateur interprétera les instruction basiques x86. Le langage assembleur étant peu complexe d’un point de vue lexical et syntaxique, le gros du travail sera sur l’interprétation sémantique. Ainsi la difficulté est principalement due à la compression des différentes instructions assembleur en une simple instruction Java.

# Instructions interprétées

Le jeu d’instruction x86 étant très large, seules les instructions basiques et communes sont prises en compte.

En se basant sur les mnémoniques présentes sur ce site :

<http://www.cs.virginia.edu/~evans/cs216/guides/x86.html>

Les instructions interprétées sont :

Affectation : MOV, PUSH, POP

Opérateurs : ADD, SUB, INC, DEC, IMUL, IDIV

Flux d’exécution : CMP, JE, JNE, JG, JGE, JL, JLE

La syntaxe utilisée est la syntaxe dite *Intel*.

L’utilisation de ces commandes peut se trouver sur le site mentionné ci-dessus.

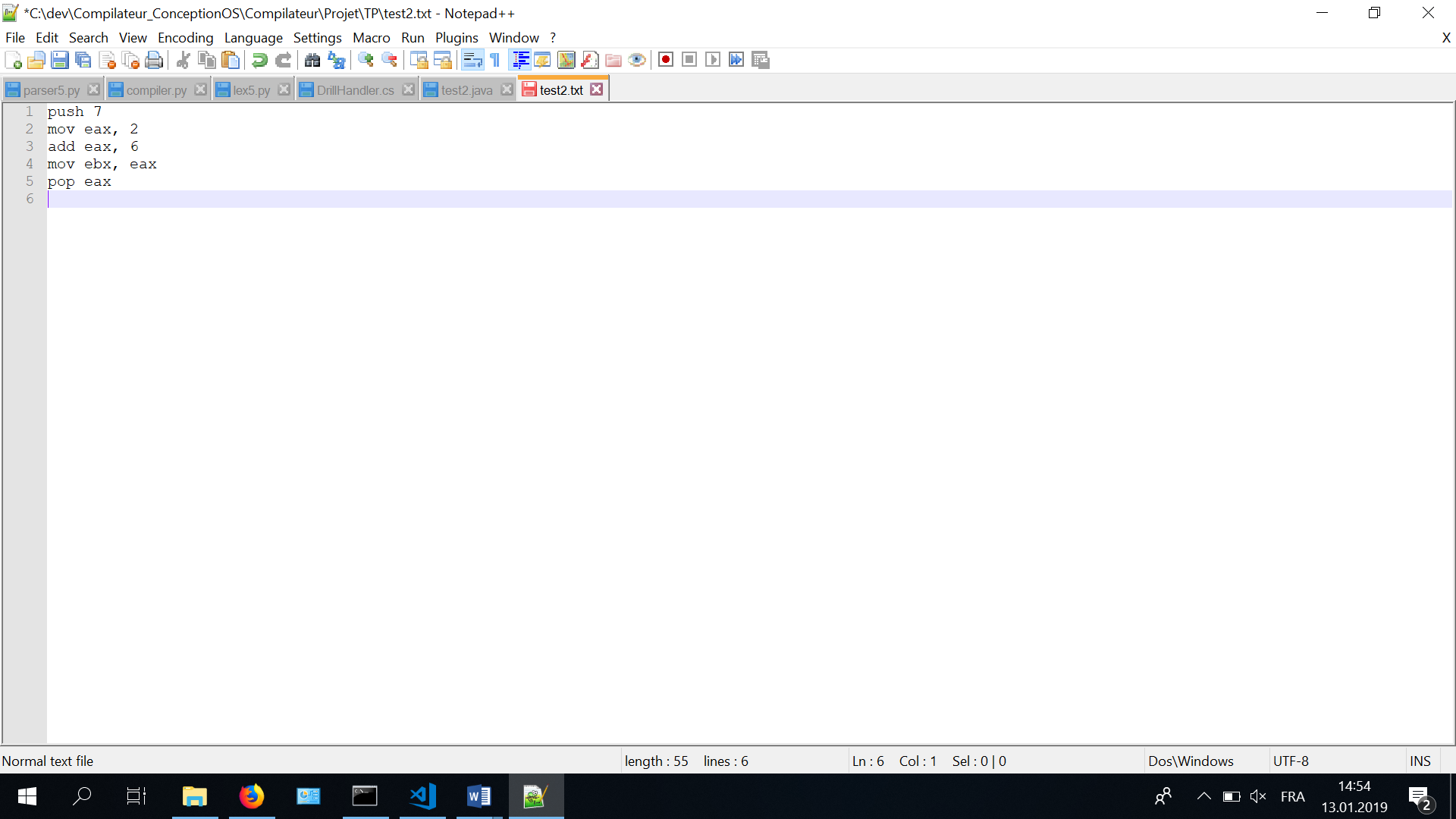
# Manuel d’utilisateur

1. Dézipper le .zip fournit dans le répertoire voulu.
2. Ajouter un fichier .txt à compiler dans le répertoire.
3. Lancer le programme python *compiler.py* dans une invite de commande par exemple avec le nom de fichier en argument, par exemple : *python compiler.py test1.txt*
4. Un fichier de sortie sera généré dans le répertoire possédant le même nom que le fichier d’entrée mais avec l’extension .java. Le contenu de ce fichier sera le résultat de la compilation. A remarquer que ce résultat sera déjà affiché par le programme python lors du traitement de la requête.

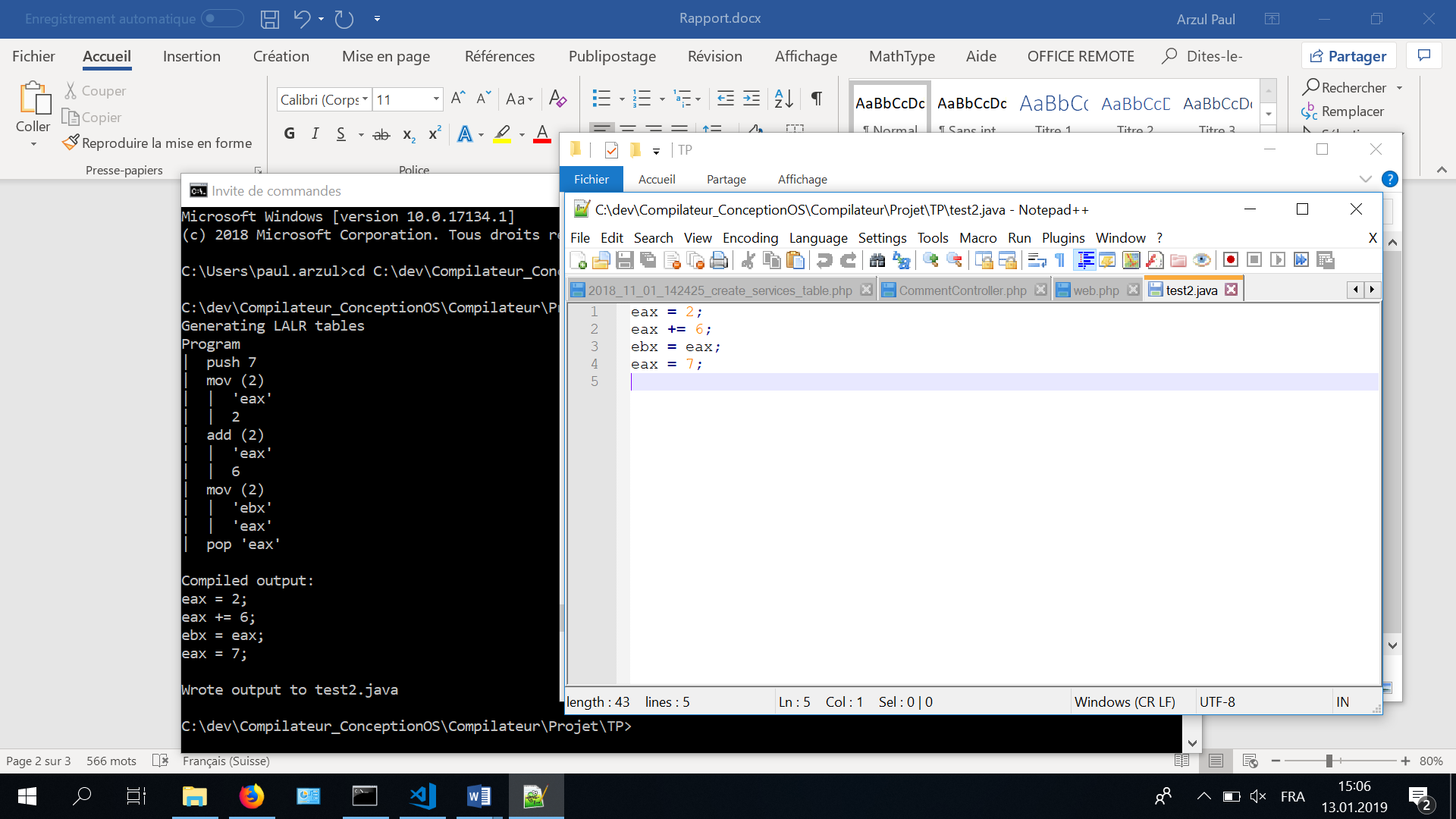
Il est également possible de tester l’analyseur lexical (*lex.py*) et le parseur (*parseur.py*) de la même manière, cependant aucun fichier de sortie ne sera généré.

# Exemple de code

Avec le code compilateur d’entrée :



Et en sortie, le code Java :



Il est important de noter que la première étape du code assembleur semble être ignorée, ceci est dû au fait que les fonctions *push* et *pop* de l’assembleur ne se traduisent pas directement en Java, une simplification se fait via une simple affectation, l’objectif est donc de traduire ces deux instructions de la manière la plus efficace possible. On peut donc voir que l’opération *pop* déclenche une sortie Java se traduisant par l’affectation de la valeur *push*.

Les autres étapes peuvent se traduire plus simplement. Ainsi un *mov* correspond à une affectation d’une case mémoire et les opérations se contentent de directement modifier la valeur d’une case mémoire.

# Problèmes et difficultés

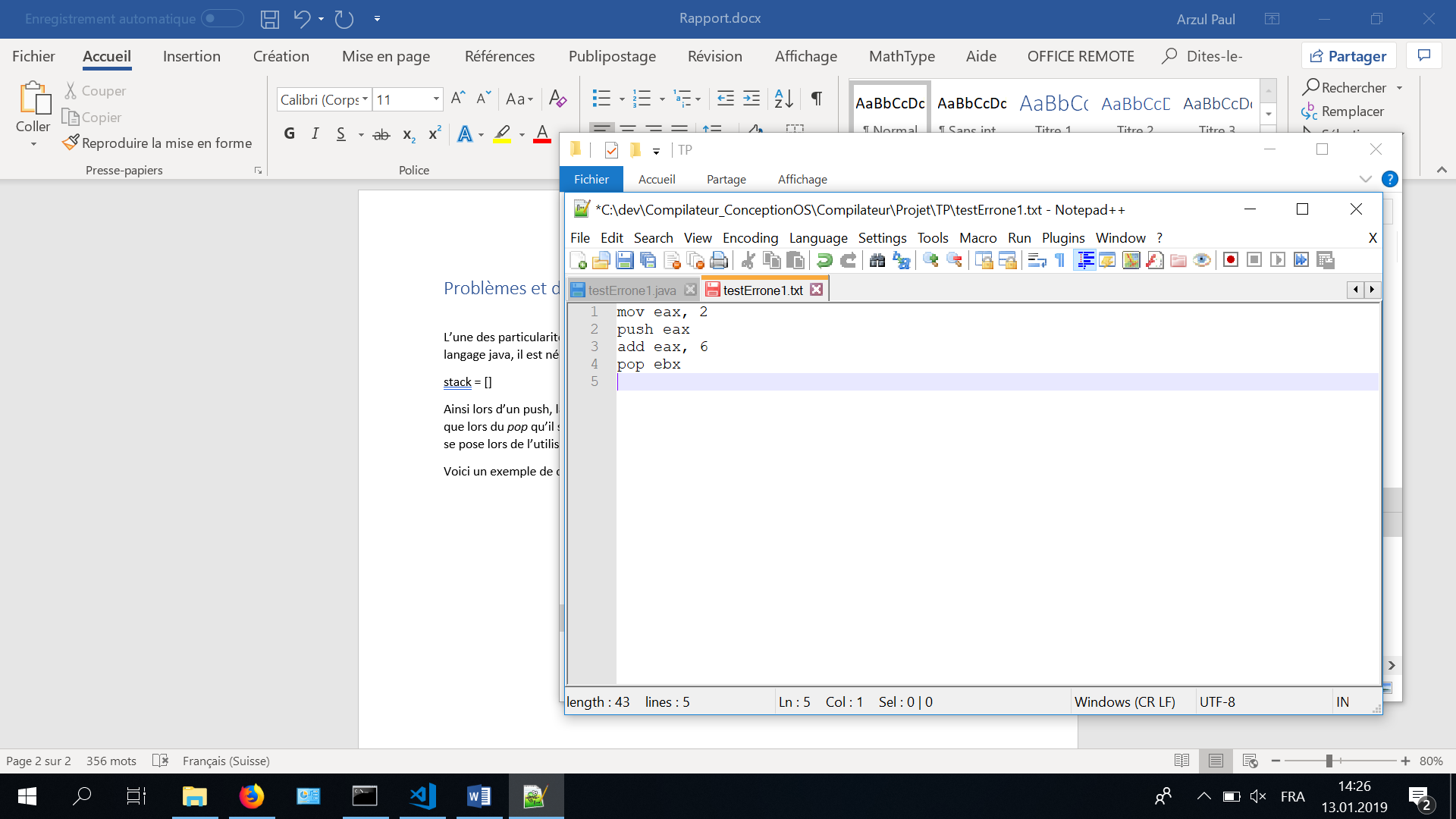
## Push / Pop

L’une des particularités du langage assembleur est la gestion du stack. Ainsi lors de la compilation en langage Java, il est nécessaire de créer dans le fichier *compiler.py* une variable stack :

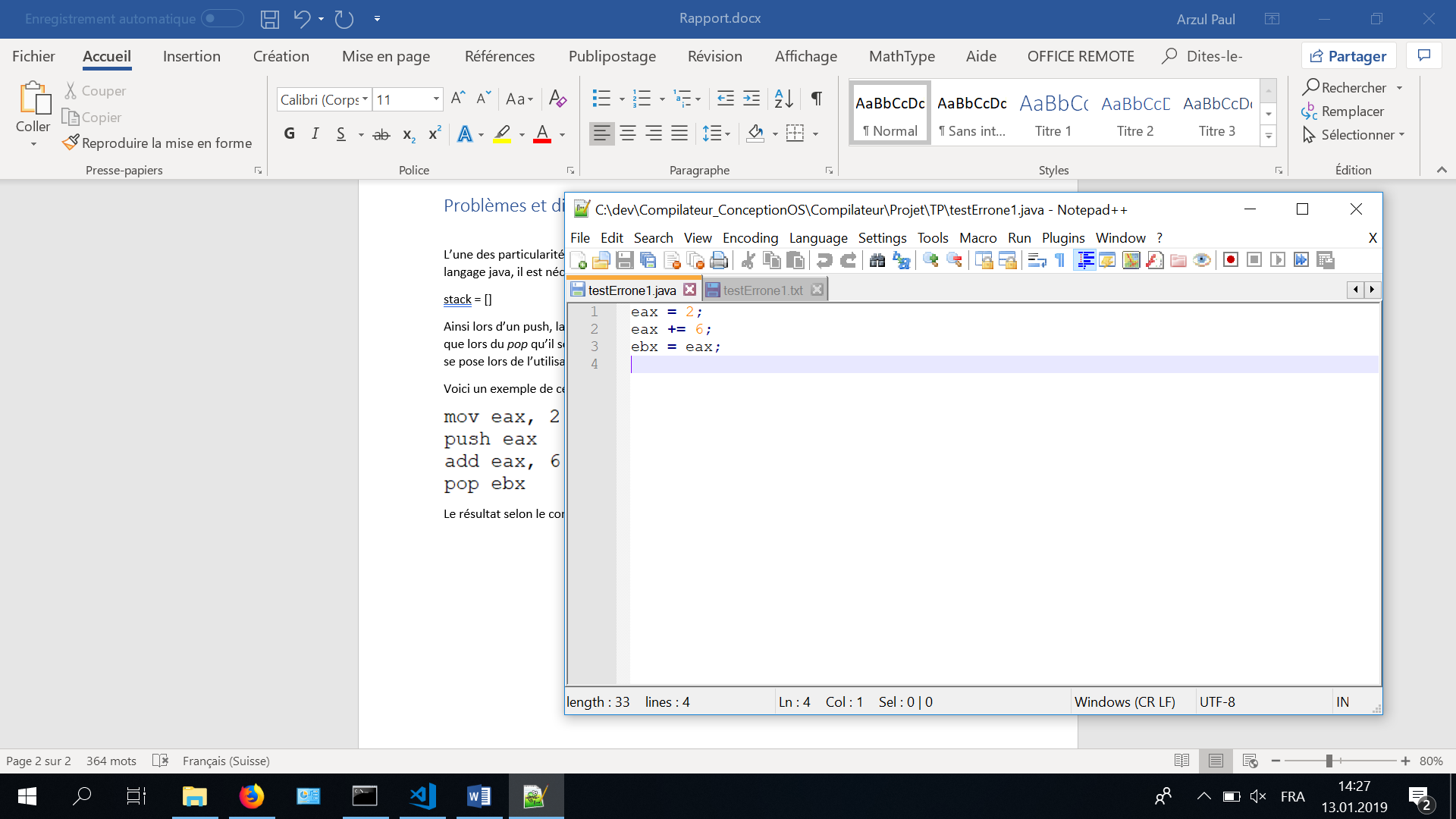
stack = []

Lors d’un *push*, la variable sera *push* dans le stack sans qu’il n’y ait encore de sortie Java. Ce n’est que lors du *pop* qu’il sera nécessaire d’aller prendre la dernière valeur *push.* Cependant un problème se pose lors de l’utilisation du *push* d’une valeur en mémoire.

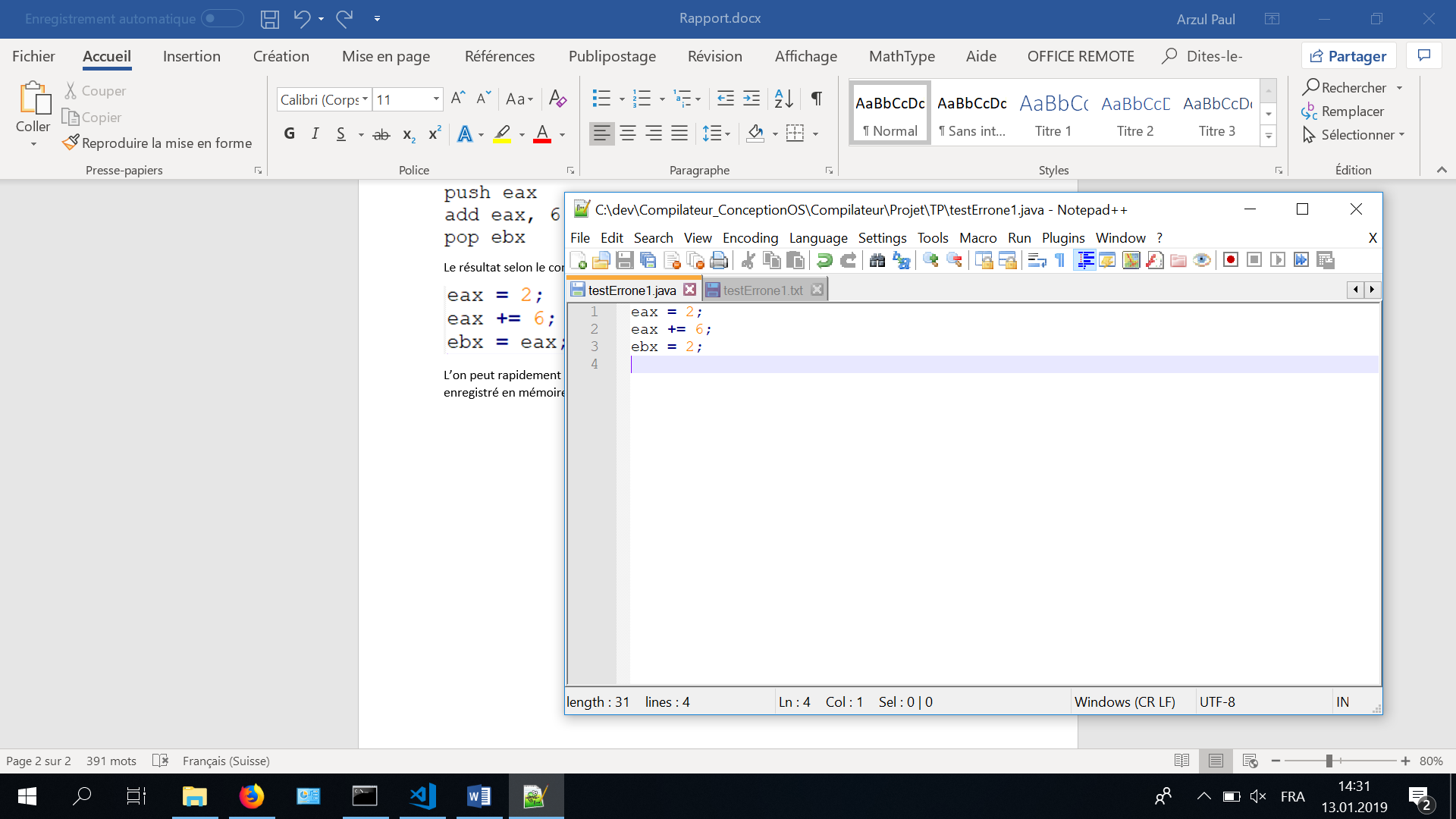
Voici un exemple de ce problème, avec un code assembleur tel que :



Le résultat selon le compilateur est ainsi :



L’on peut rapidement constater ici que le code Java va affecter la valeur de *eax* actuelle et non celle enregistrée en mémoire. Le résultat correct dans ce cas-là serait donc :



Une solution pour régler ce problème serait parfaitement envisageable, il faudrait simplement interpréter la valeur du token et non prendre le token lui-même.

Cependant dans une optique future, il serait théoriquement possible de demander des valeurs à l’utilisateur. Dans un cas comme celui-là, simplement exécuter le token ne serait pas possible, la valeur d’entrée de l’utilisateur n’étant pas encore déterminée lors de la compilation.

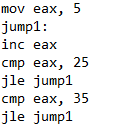
L’optique de faire réapparaitre des variables plus loin dans le code lors de la compilation est un problème plus compliqué qu’envisagé, la solution implémentée semble tout de même convenir dans un premier temps.

## Jump

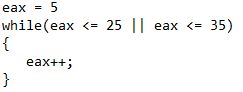
L’instruction jump permet d’altérer le flux d’exécution du programme. Cette instruction est gérée via les saut conditionnels courants : on fait suivre l’instruction *cmp* d’une instruction de branchement conditionnel, p.ex *jne* (jump when not equal).

Actuellement, seul un label est géré, c-à-d une seule boucle, mais il est possible d’intégrer plusieurs sauts conditionnels.

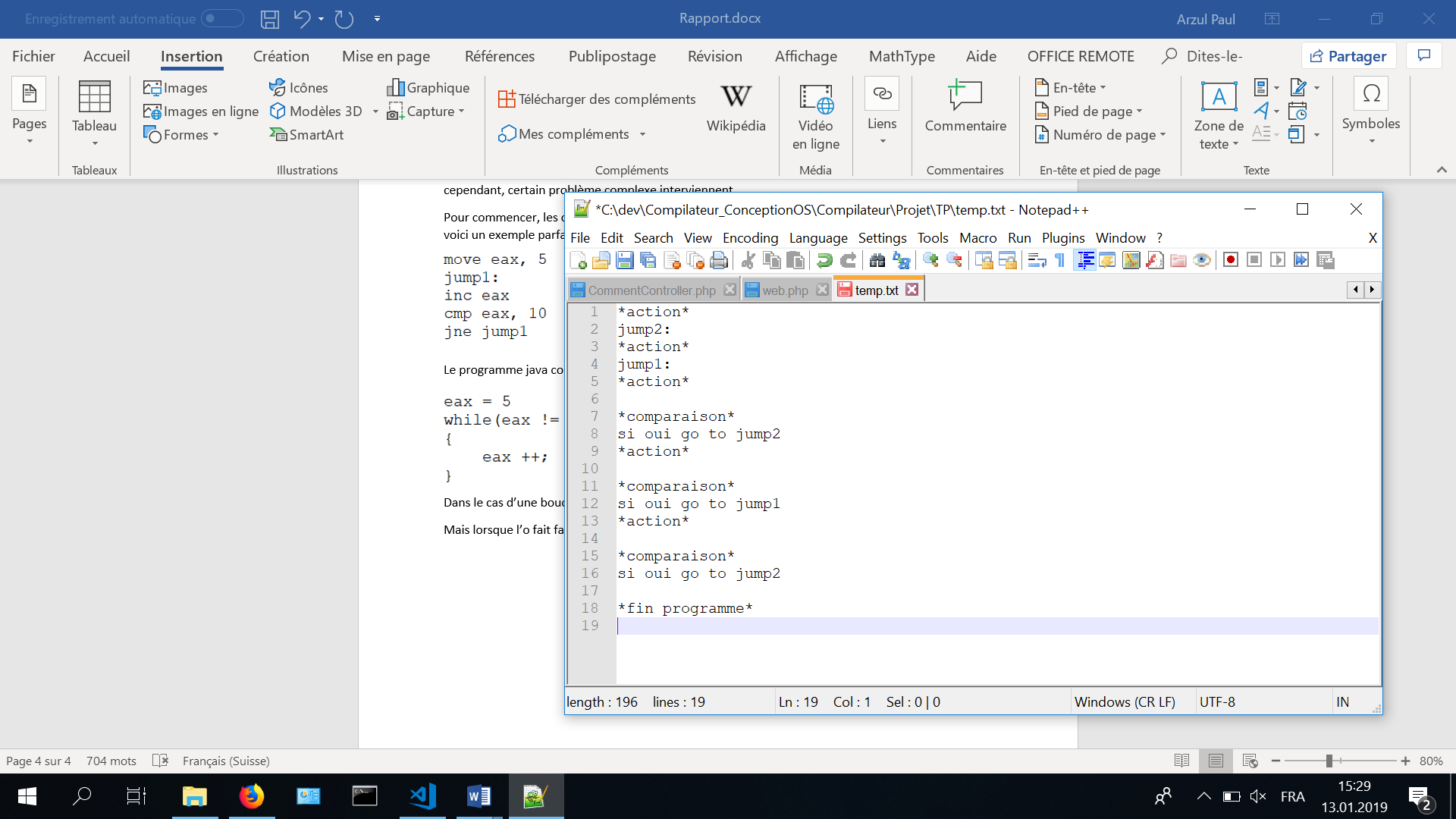
Le code suivant est géré



et correspond au code Java suivant :



Le code suivant, qui comporte plusieurs labels, et qui correspondrait à deux boucles imbriquées n’est cependant pas géré.



Tout en sachant que le concept de saut tel que le langage compilateur possède n’existe pas vraiment en Java, l’une des solutions possibles serait la création d’une fonction pour chaque action après label. Ces fonctions se faisant ensuite appeler lors des comparaisons. Une telle implémentation ajouterait de nouvelles complications, en effet comment déterminer quelle partie de code correspondrait au corps de la fonction du label. Sachant que plusieurs labels peuvent être disposés l’un à la suite des autres bien que le code de chacun s’exécute tout de même de manière séquentiel.

En résumé le problème comporte de multiple couche de complexité qui, dans un travail tel que celui-ci, s’avère trop compliqué.

## Conclusion

En l’état actuel, le compilateur satisfait les objectifs fixés. Son domaine de fonctionnement reste tout de même très restreint. L’ajout des sauts auraient été idéal pour palier à cela, malheureusement la problématique posée par les sauts s’est avérée plus grand qu’attendu.

Par manque de temps et de compréhension, les sauts n’ont donc pas été implémentés.

Il est important de noter que le compilateur n’interprète pas les commande possédant un nombre d’argument supérieur à deux, en effet pour les opérateurs il est possible de passer un argument supplémentaire. Cependant, dans chacun de ces cas, cet argument supplémentaire est out à fait traitable de la même manière grâce à une instruction supplémentaire.

Il faut aussi savoir que la gestion de l’emplacement mémoire des variables n’est pas géré, ainsi, il n’est pas possible de décaler ou modifier les valeurs bit par bit en java. A moins bien évidement via l’utilisation d’une classe spécifique au problème, ceci allant donc à l’encontre de l’exercice.