

Projet de Compilation  
L3 Informatique  
2021-2022

Projet  
**Algo en LaTeX**

Dombry Baptiste  
Dourlen Maxime  
Groupe TP3

## Sommaire :

- I. Introduction
- II. Grammaire utilisée
  - II.1. Terminaux de la grammaire
  - II.2. Les non-terminaux de la grammaire
- III. Code Assembleur
- IV. Conclusion
- V. Jeu d'exemples

# I. Introduction

Lors de notre 3e année en licence d'informatique, durant les TP de compilation, nous avons créé un petit compilateur à l'aide de flex, bison et de l'assembleur ASIPRO. Nous allons donc utiliser les connaissances acquises lors de ces TP pour réaliser ce projet. Il consiste à écrire deux programmes :

- le premier : « `algo2asm fichier.tex` » génère un fichier `fichier.asm` contenant le code source ASIPRO correspondant au contenu de `fichier.tex`, dont on supposera qu'il ne contient que la description d'une fonction décrite en utilisant Algo.

- le deuxième : `run '\SIPRO{fichier}{arg}'` utilise SIPRO pour calculer la valeur de la fonction dont le nom est donné en premier argument de la commande `\SIPRO`, en utilisant les arguments donnés en second argument de la commande `\SIPRO`. Ainsi, `run '\SIPRO{puissance}{2,3}'` produit 8 sur sa sortie standard, si `puissance.tex` est un algorithme de puissance.

## II. Grammaire utilisée

Pour ce qui est de la grammaire, cela a été le principal axe de recherche pour commencer le projet, après beaucoup d'essais et de mise sur papier on obtient une grammaire correcte.

### II.1 Les terminaux de la grammaire

`\\begin{algo}`, `\\end{algo}` : représente la commande qui commence l'algorithme ainsi celle qui finit l'algorithme.  
Note, si l'on trouve « `\\begin{algo}` » dans un fichier correcte, on aura toujours un « `\\end{algo}` » à la fin, très utile à savoir pour la grammaire.

`\\SET` : représente la commande qui produit une affectation  
Exemple : `\\SET{i}{j}`: produit  $i \leftarrow j$ .

`\\DOFORI`, `\\OD` : représente la commande qui permet de faire une boucle for. « `\\OD` » se présente obligatoirement a la fin.

`\\DOWHILE` : représente la commande qui permet de faire une boucle while. Comme la boucle for, il faut que « `\\OD` » se trouve a la fin obligatoirement.

`\\RETURN` : représente la commande qui permet de retourner une valeur a la fin de l'algorithme

`\\IF`, `\\ELSE`, `\\FI` : représente la commande qui permet de faire des conditions, il faut que « `\\FI` » se trouve a la fin obligatoirement.

`\\SIPRO` : représente la commande qui permet de lancer une fonction.

`\\CALL` : représente la commande qui permet de faire l'appel à un autre algorithme.

`[[:digit:]]+` : représente une suite non vide de chiffres  
`[[:blank:]]|\n` : représente un espace ou un saut a la ligne  
(cela permet de les ignorer)

`&&`, `||`, `!` : représente les opérateurs logiques OU `||`, ET `&&`,  
NON `!`

`<`, `<=`, `>`, `>=`, `=`, `!=` : représente les opérateur relationnels

`true`, `false` : représente les deux valeurs booléennes  
possibles

`([a-zA-Z][a-zA-Z0-9_]*)|([_][a-zA-Z][a-zA-Z0-9_]*)` :  
représente une suite de caractères permettant notamment de  
déclarer le nom des variables

## II.2 Les non-terminaux de la grammaire

`starter: SIPRO '{' ID '}' '{' numbers '}'`  
`| function`

*Permet de lancer la fonction SIPRO si on l'utilise sinon on  
lit « function »*

`numbers: NUMBER ',' numbers`  
`| NUMBER`  
`| ε`

*Permet d'énumérer une liste de nombres séparés d'une virgule*

`function: function BEGIN_ALGO '{' ID '}' '{' ids '}'`  
`function END_ALGO`  
`| ε`

*Debut de l'algorithme encadré par BEGIN\_ALGO et END\_ALGO*

`ids: ids ',' ID`  
`| ID`

*Permet d'énumérer une liste de variables séparées d'une  
virgule*

`instrs: instr instrs`

|  $\epsilon$

*Permet de lire une suite d'instructions*

```
instr: SET '{' ID '}' '{' expr '}'  
      | DOFORI '{' fix ID '}' '{' expr '}' '{' expr '}'  
        instrs OD  
      | IF '{' expr fixif '}' instrs FI  
      | IF '{' expr fixif '}' instrs ELSE instrs FI  
      | RETURN '{' expr '}'  
      | DOWHILE '{' fix expr '}' instrs OD
```

*Permet de lire les différentes instructions possibles des algorithmes passés en paramètre*

fix:  $\epsilon$

fixif:  $\epsilon$

*Évite les bugs*

```
exprs: expr ',' exprs  
      | expr  
      |  $\epsilon$ 
```

*Permet d'énumérer une liste d'expressions séparées d'une virgule*

```
expr: NUMBER  
      | '(' expr ')'  
      | expr '+' expr  
      | expr '-' expr  
      | expr '*' expr  
      | expr '/' expr  
      | VRAI  
      | FAUX  
      | ID  
      | expr OU expr  
      | expr ET expr  
      | expr EQ expr  
      | expr NEQ expr  
      | CALL '{' ID '}' '{' exprs '}'
```

*Permet de lire une expression*

### *III. Code Assembleur*

Pour la partie assembleur, nous avons fait le choix de nous inspirer de ce que nous avons fait lors des séances de TP. Lors de nos séances de TP nous avons pu mettre en place de nombreuses vérifications sur les types, booléen ou entier. Lors de ce travail afin de ne pas trop compliquer le code assembleur, nous avons préféré enlever certaines vérifications sur les booléen. Ainsi le code assembleur produit ne peut gérer que des entiers en entrée, car lors de la déclaration de la fonction dans le fichier .tex il n'est pas spécifié de quel type la variable sera.

Lors de la déclaration d'une fonction nous avons fait le choix d'ajouter une condition permettant de sauter à la fin d'une déclaration. Les instructions :

« `const dx,func:%s:fin` » suivi de « `jmp dx` » permettent de ne pas tomber accidentellement dans la fonction déclarée. Ainsi le seul moyen d'accéder à cette fonction est grâce à l'appel « `call` » ou avec un appel à « `jmp` ». Pour appeler les fonctions lors de l'exécution nous avons préféré utiliser l'appel « `call` » plutôt qu'un appel à « `jmp` » qui est plus restreignant pour un retour à l'instruction suivant cet appel.

Les appels à « `call` », nous ont posé certains soucis, car l'adresse de la prochaine instruction est stockée en sommet de pile. Les valeurs des variables sont donc en dessous de l'adresse de la prochaine instruction. Il nous a donc fallu, pour récupérer les valeurs faire des échanges en sommet de pile à chaque variable lue. Il est possible de voir ces échanges dans le fichier à la ligne 121-123 du fichier .y

Nous avons fait le choix pour encore une fois simplifier le code produit, de demander à l'utilisateur d'écrire dans le fichier .tex toutes les fonctions annexes appelées par l'algorithme principal. Ce qui a ensuite été un gain par rapport à un programme qui aurait effectué les appels « `call` » avec des fonctions déclarées dans un autre fichier car il aurait été nécessaire de regarder si le fichier contenant la fonction existe. S'il n'existe pas il faut essayer de le compiler en imaginant qu'il se situe dans le

dossier de compilation. Une fait trouvé de le compiler en code assembleur et ensuite de copier le contenu du fichier dans le fichier en cours d'écriture.

L'un des problèmes rencontré a été les valeurs de retour des fonctions avec l'appel « ret ». Ce dernier doit avoir en sommet de pile l'adresse de l'instruction suivant l'appel « call ». Or nous avons la valeur de retour en sommet de pile, nous avons du faire un échange entre ses deux valeurs.



## IV. Conclusion

Ce projet, nous à permet de découvrir, compléter et assimiler le bison et le yacc. Nous avons du recourir à des méthodes de réflexion différentes de celles dont nous avons l'habitude. En effet, nous devions imaginer la grammaire avant d'écrire le code assembleur qui correspond, en plaçant bien le code assembleur avant ou après certaines évaluations de non terminal. Ce projet nous a permit d'apprendre à gérer la pile et imaginer l'état de la pile au cours de l'exécution ce qui permet d'éviter certains problèmes comme les valeurs en sommet de pile devant les instructions « ret » ou bien encore les valeurs en dessous de la valeur placée en sommet de pile par l'appel « call ». Nous n'avions pas effectué de code assembleur depuis la première année de licence Informatique, qui était du code assembleur en 8086. Ce projet nous a donc permis de replonger dans l'assembleur, de redécouvrir ses subtilités et de nous entraîner à décoder des programmes compilés.

Ce projet, nous permet d'imaginer comment les langages de programmation comme le « Brainfuck » ont été écrits et développés, ce qui peut nous ouvrir également les portes de la création de nouveau langage.

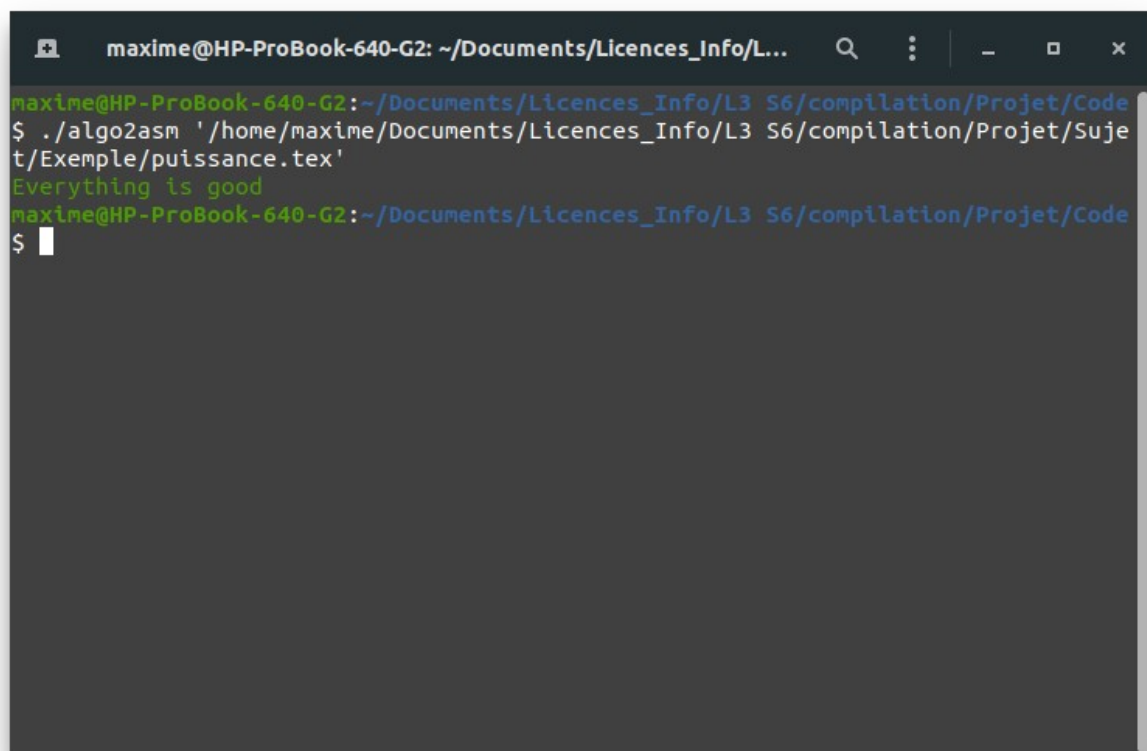
## V. Jeu d'exemples

Nous souhaitons compiler le fichier puissance.tex qui se trouve dans le dossier Exemple qui se trouve dans le dossier Sujet, le fichier puissance.tex contient l'algorithme permettant de calculer la puissance d'un nombre 'a'.

```
Projet > Sujet > Exemple > TeX puissance.tex
1  \begin{algo}{puissance}{a,b}
2  |  \IF{b=0}
3  |  |  \RETURN{1}
4  |  \FI
5  |  \RETURN{\CALL{puissance}{a,(b-1)} * a}
6  \end{algo}
```

Ici nous pouvons voir que l'algorithme est récursif.

Pour compiler ce fichier on fait la commande :

A terminal window with a dark background. The title bar shows 'maxime@HP-ProBook-640-G2: ~/Documents/Licences\_Info/L...'. The prompt is 'maxime@HP-ProBook-640-G2:~/Documents/Licences\_Info/L3 S6/compilation/Projet/Code'. The user enters '\$ ./algo2asm '/home/maxime/Documents/Licences\_Info/L3 S6/compilation/Projet/Sujet/Exemple/puissance.tex''. The output is 'Everything is good'. The prompt returns to '\$ ' with a cursor.

```
maxime@HP-ProBook-640-G2:~/Documents/Licences_Info/L3 S6/compilation/Projet/Code
$ ./algo2asm '/home/maxime/Documents/Licences_Info/L3 S6/compilation/Projet/Su
t/Exemple/puissance.tex'
Everything is good
maxime@HP-ProBook-640-G2:~/Documents/Licences_Info/L3 S6/compilation/Projet/Code
$
```

Nous pouvons voir que ici que tout s'est bien passé car le message « Everything is good » s'affiche. Ensuite pour utiliser la fonction compilée nous effectuons la commande :

```
maxime@HP-ProBook-640-G2: ~/Documents/Licences_Info/L...  
$ ./algo2asm '/home/maxime/Documents/Licences_Info/L3 S6/compilation/Projet/Code  
t/Exemple/puissance.tex'  
Everything is good  
maxime@HP-ProBook-640-G2:~/Documents/Licences_Info/L3 S6/compilation/Projet/Code  
$ ./run '\SIPRO{puissance}{2,3}'  
8  
Everything is good  
maxime@HP-ProBook-640-G2:~/Documents/Licences_Info/L3 S6/compilation/Projet/Code  
$
```

Nous pouvons ici voir que tout s'est bien passé car le message « Everything is good » s'affiche. Nous pouvons également voir au dessus le résultat de l'algorithme utilisé, ici  $2^3$  qui donne 8.