Compte rendu V.1.1 :

# Nouveautés :

* Mettre en place les expressions algébriques et arithmétiques
* Mettre en place les variables
* Faire les appels de fonctions prédéfinies
* Faire la génération de code et l’exécution du code
* Faire la traçabilité avec des affichages

Expressions logiques :

* ! -> NOT
* && -> AND
* || -> OR

Expressions comparaison :

* < : Strictement inférieur
* <= : inférieur ou égale
* > : Strictement Supérieur
* >= : Supérieur ou égale
* == : égale
* != : non égale

Séparateurs :

* ;
* {}

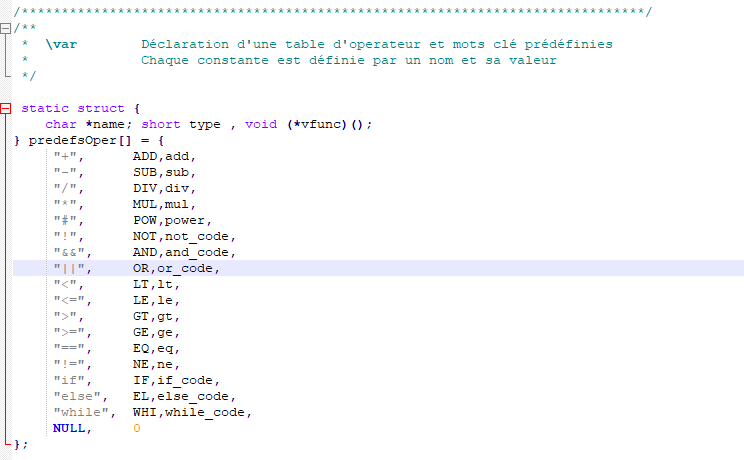
Mots-clés :

* if
* else
* while

# Etapes :

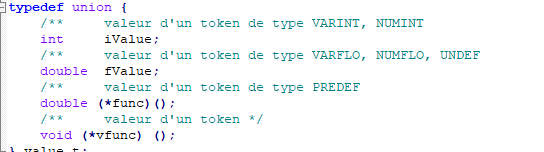
## Ajouts généraux :

**1er étape : Ajout d’une nouvelle structure dans defSymbols.c :**



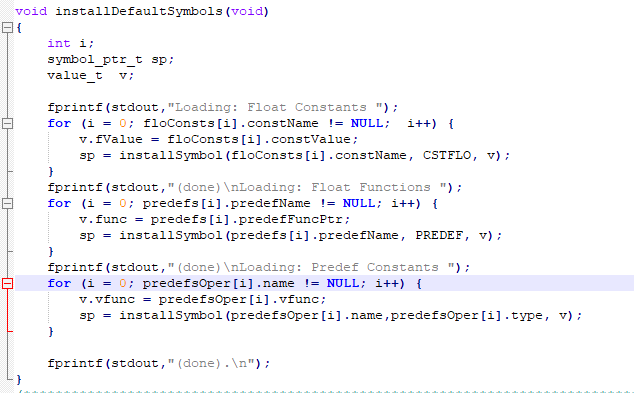
Déclaration d’une nouvelle structure pour reconnaitre tous les opérateurs (normaux, logique, comparaison) et les mots clés en tant que symbole. On associe pour chacun un type et une fonction .

**2éme étape : déclaration d’un nouveau type pour reconnaitre un symbol avec une fonction void ( \*vfunc) () ; fichier hoc.h :**



Ajout de la fonction void pour gérer les fonctions de type void appelées pour chaque symbole.

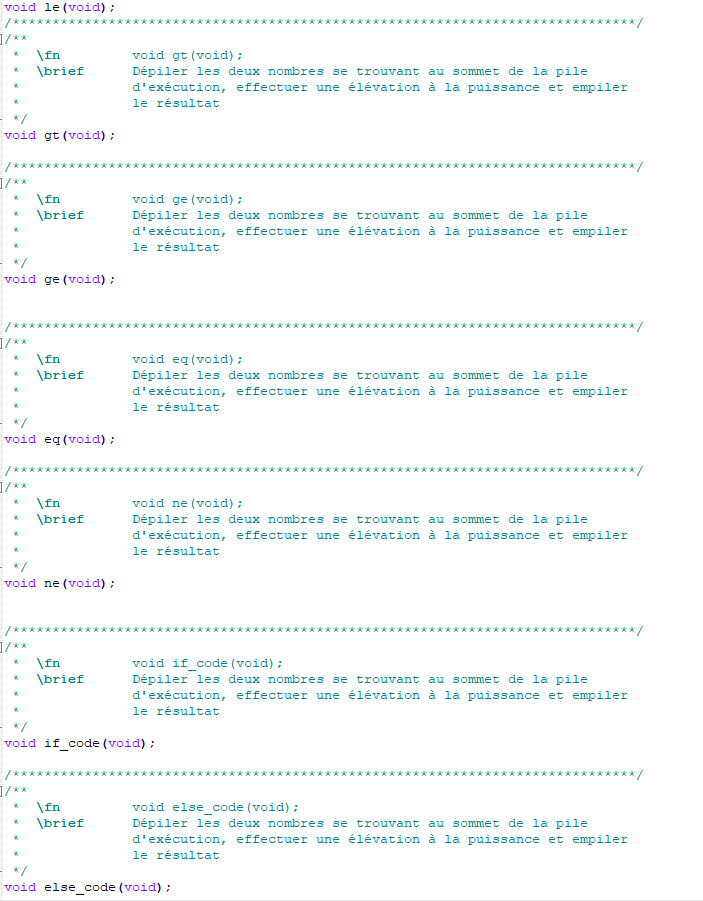
**3 éme étape : Installation de la nouvelle structure dans codeHoc.c :**



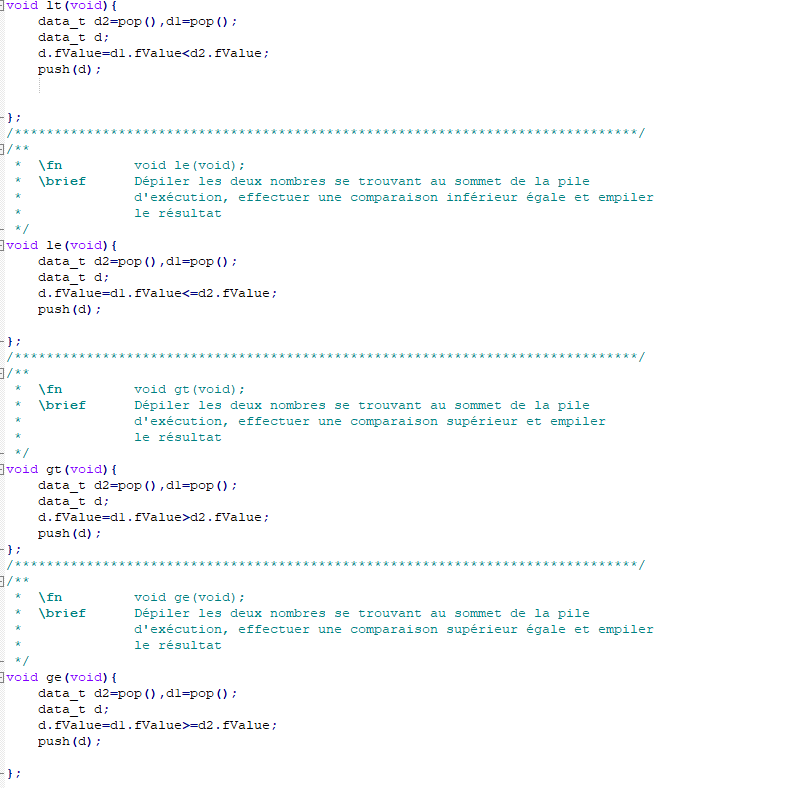
Ajout dans l’installation des symboles de la structure des opérandes.

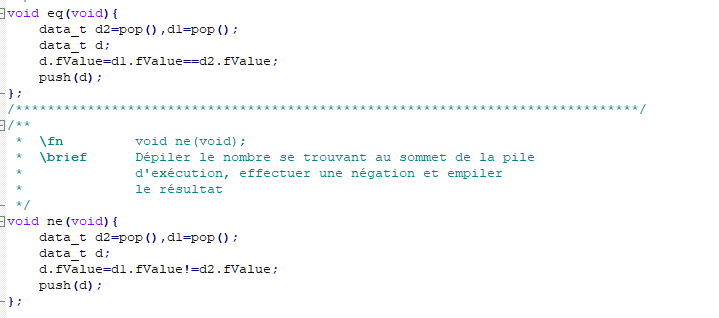
**4 éme étape : Déclaration des nouvelles fonctions dans hoc.h :**

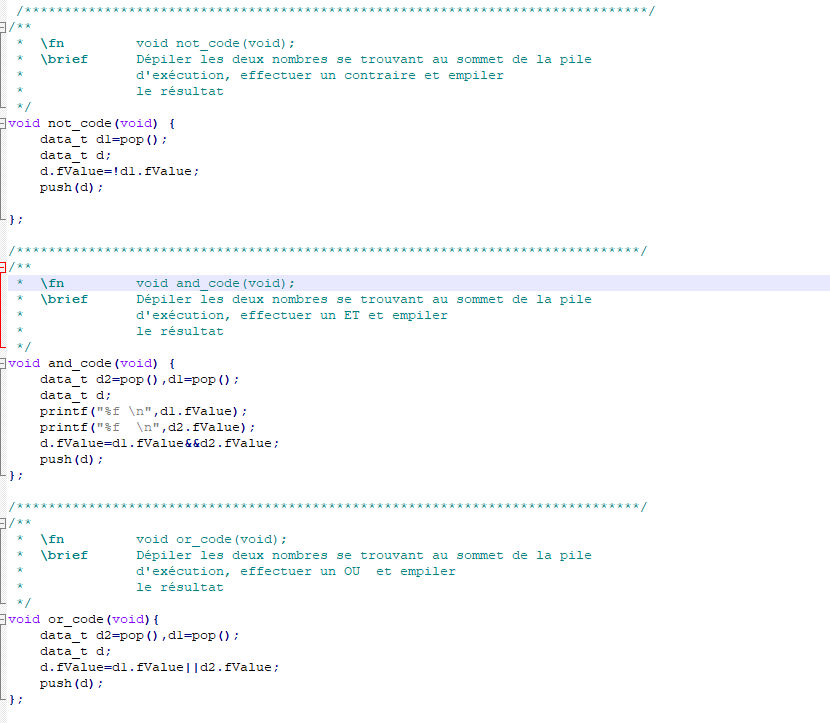
Aperçu des nouvelles fonctions

****

**5 éme étape : écriture du contenu des nouvelles fonctions dans codeHoc.c pour les opérateurs de comparaison et les booleen :**







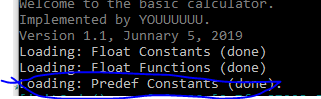
Dans chaque fonction on récupère les deux nombres ou un seul (NOT) et on effectue la comparaison avec les signes ou la comparaison booléenne. On push ensuite le résultat.

**6 éme étape : Ajout des nouveaux token dans hoc. :**

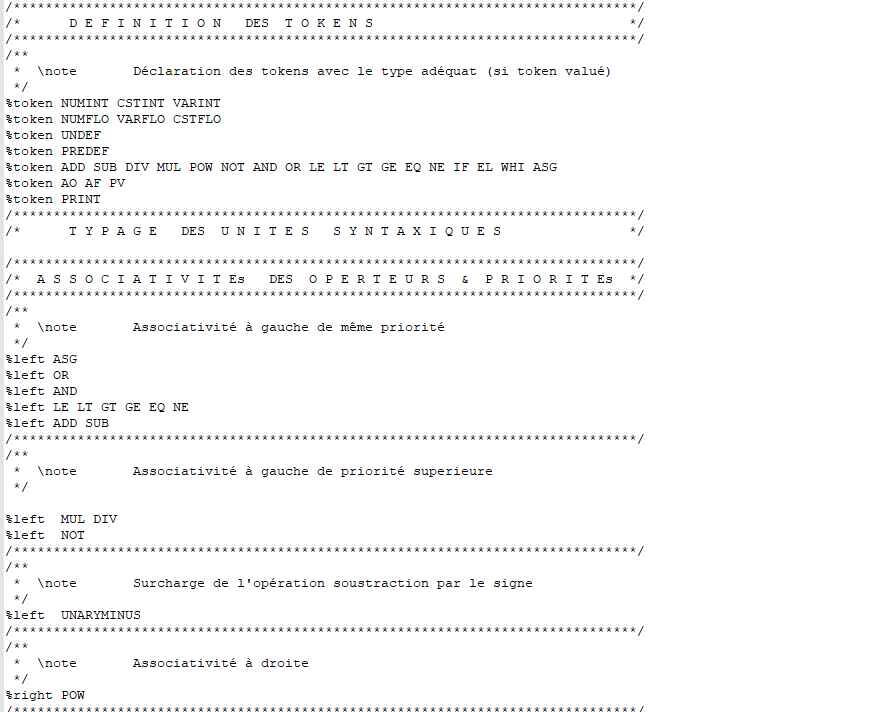


Token qui sont reconnues.

Test de l’installation des symboles :

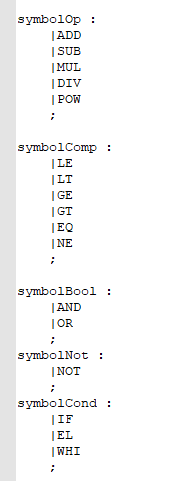


**7 éme étape : Ajout de l’ordre de priorité des opérandes dans hoc.y :**



On définit la priorité pour les nouveaux opérateurs par rapport aux autres.

**8 éme étape : ranger les symboles dans une même catégorie dans hoc.y :**



Rangement en différentes parties pour mieux reconnaître les opérateurs d’une même catégorie

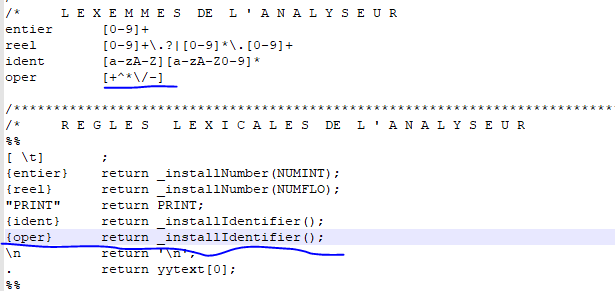
## Partie opérateur :

**1er étape : changer la reconnaissance pour les symbolOp  dans hoc.y:**



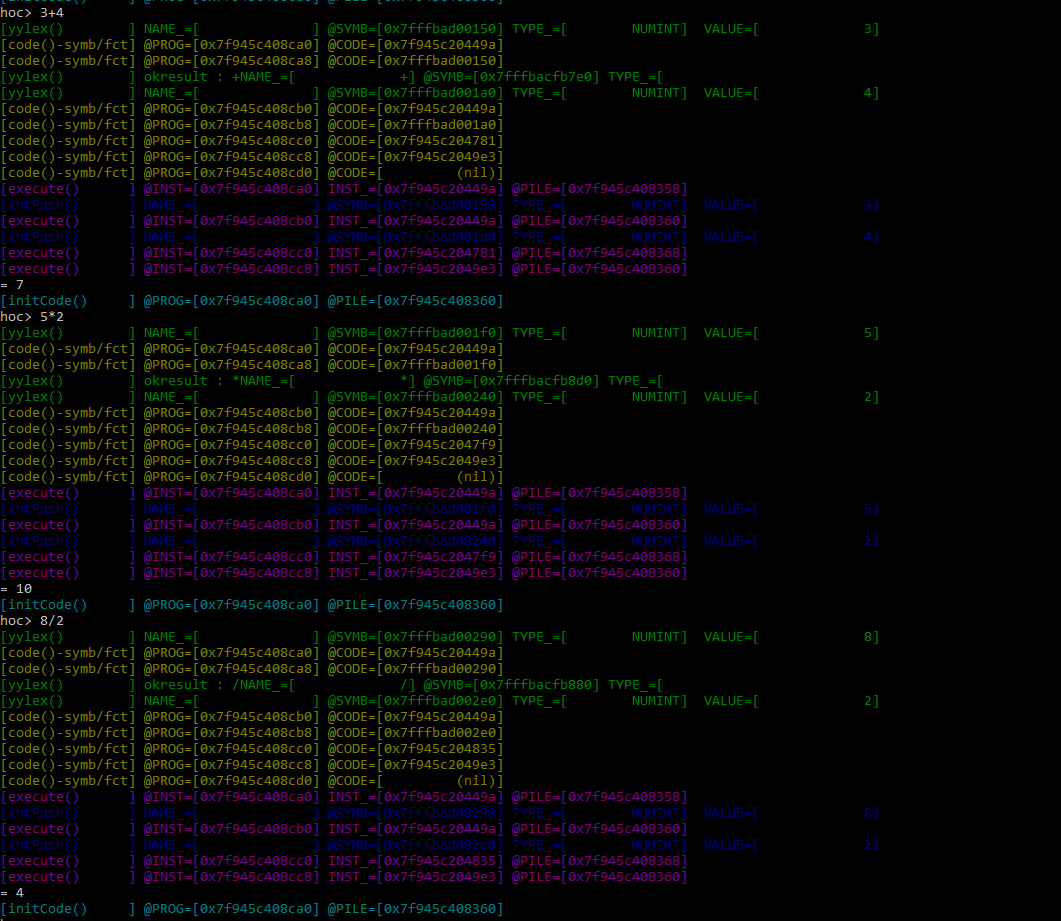
Une seule ligne pour tous les opérateurs à la place de 5.

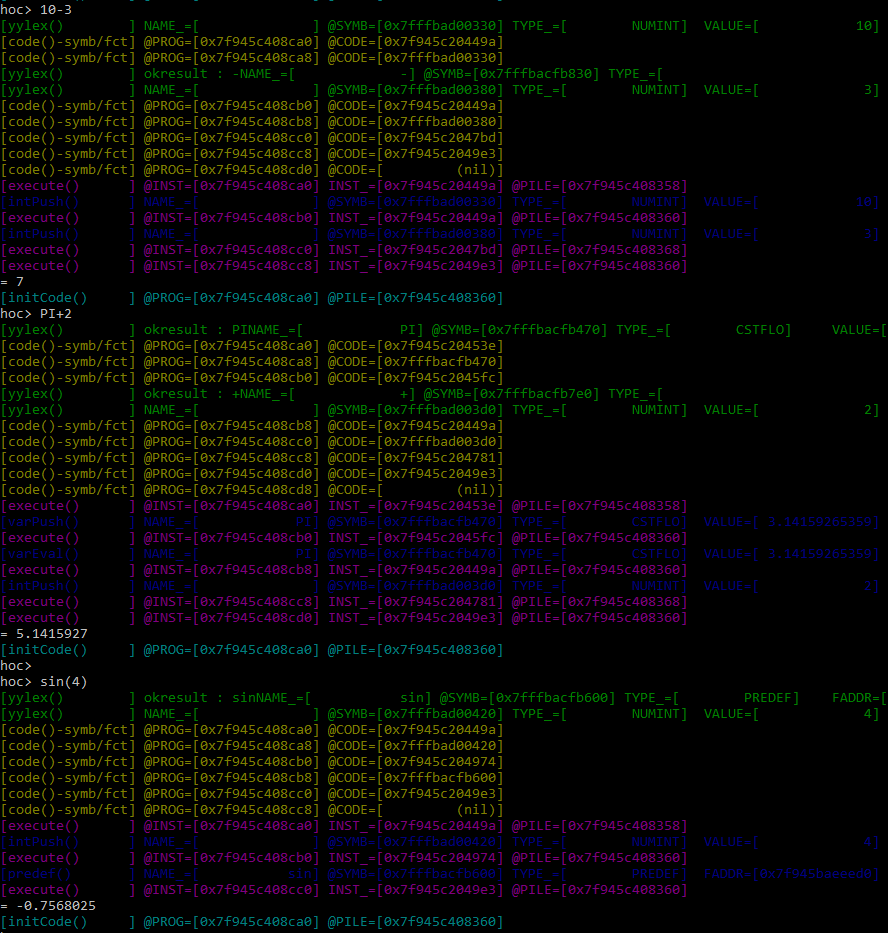
**2 éme étape : Ajout dans le lexème de l’analyseur : hoc.l et nouvelles règles lexical de l’analyseur :**

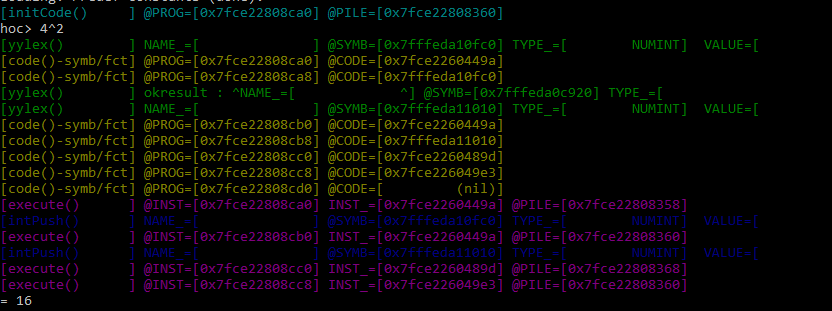


Reconnaissance des opérateurs + - / \* ^ et installation des symboles.

**Test de la génération de code et de l’exécution des symbolOp :**





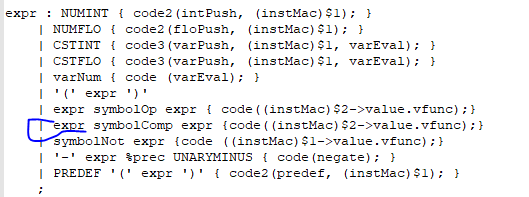
****

Test des différents symboles de types + ,-,/,\*,^ et des différentes constantes.

**Pour la suite des exemples les résultats de chiffre 1 correspond à TRUE et 0 à FALSE**

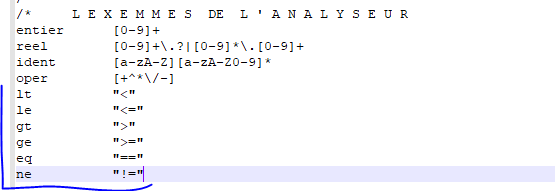
## Partie opérateur de comparaison :

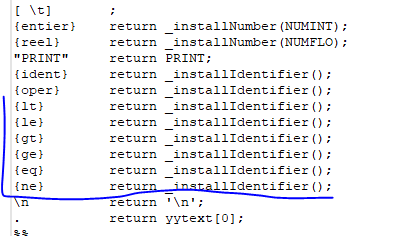
**1 er : créer la reconnaissance pour les symbolComp :**



Définition de la mise en place de la comparaison entre une expression et une autre .Génération du code avec la fonction associé à chaque symbole.

**2éme étape : Ajout dans le lexème de l’analyseur et nouvelles règles lexical de l’analyseur (hoc.l):**

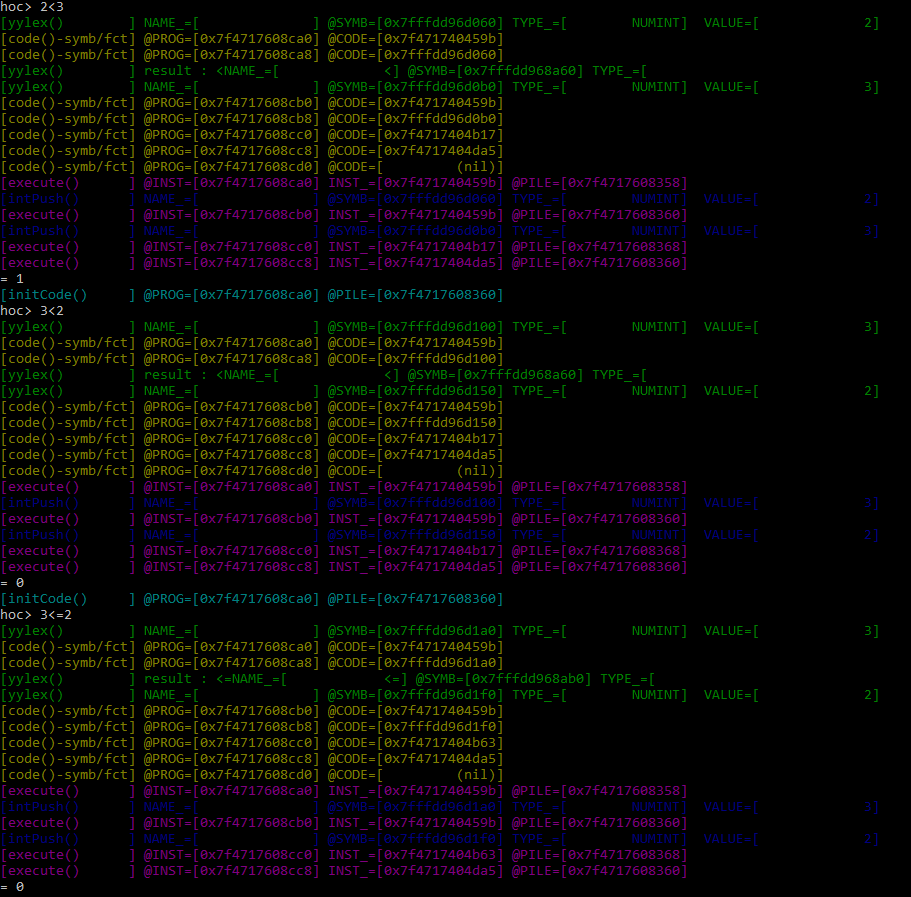




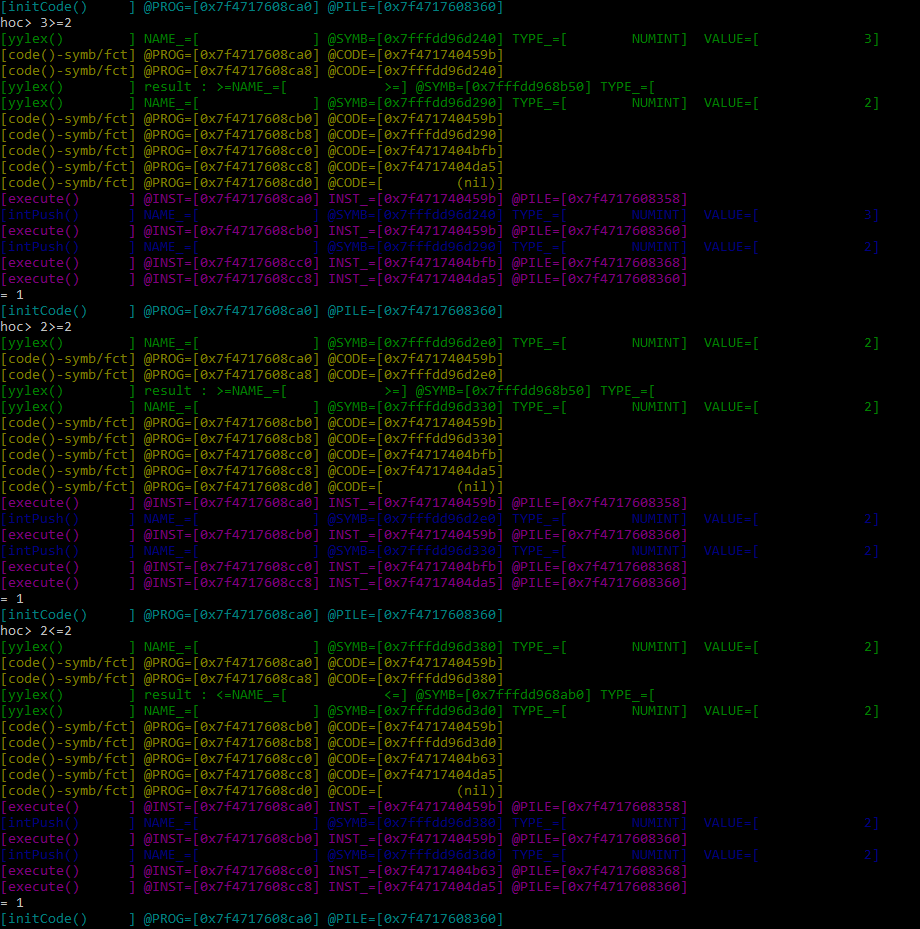
Définition des opérateurs de comparaison pour qu’ils puissent être reconnus en tant que symbole.

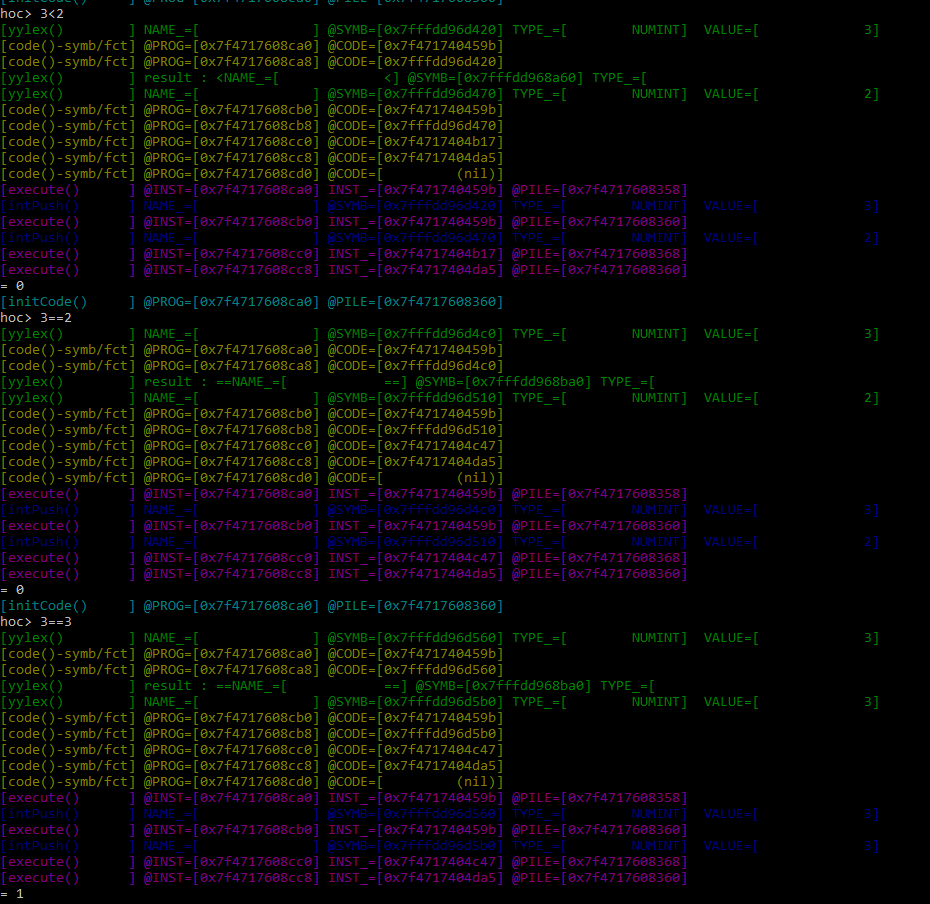
**Test de la génération de code et de l’exécution des symbolComp :**

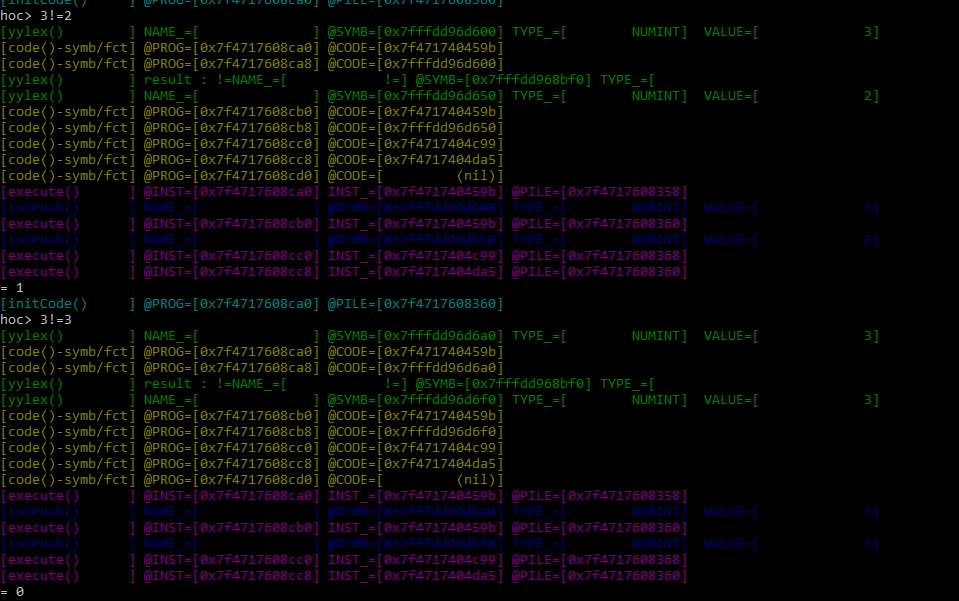
Test des différents symboles de comparaison <,>, ==,=, <=,>=.



## 

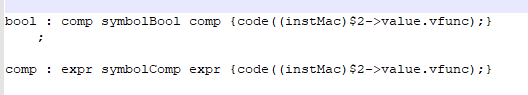


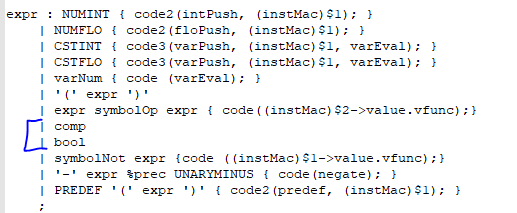




## Partie opérateur booléen :

**1 er : créer la reconnaissance pour les symbolBool :**





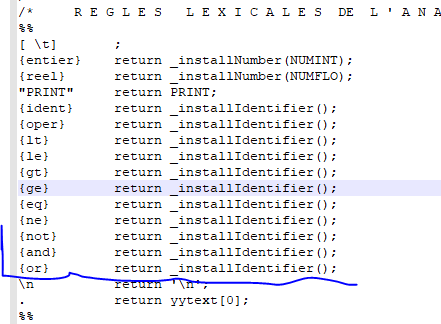
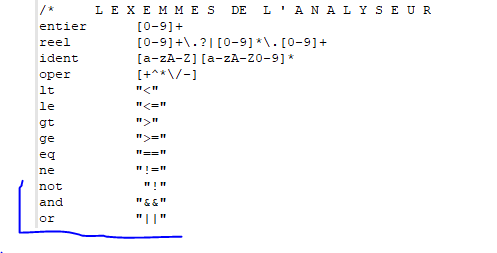
1er image : Reconnaissance des symboles ET et AND entre deux « comp » que l’on appelle  « bool »

Reconnaissance de « comp » qui correspond à une comparaison entre deux expressions

2éme image : Ajout d’en « expr » de la reconnaissance d’une comparaison et d’un booléen entre deux comparaison

Ajout de symbolNot devant une « expr » pour reconnaitre un NOT suivit d’une expression

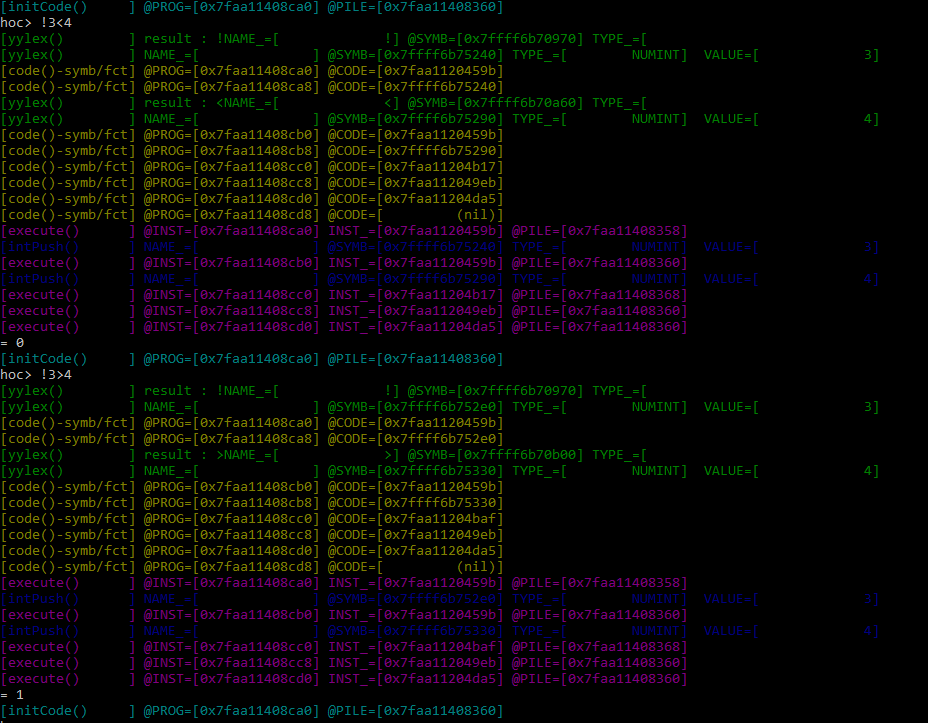
**2éme étape : Ajout dans le lexème de l’analyseur et nouvelles règles lexical de l’analyseur (hoc.l):**



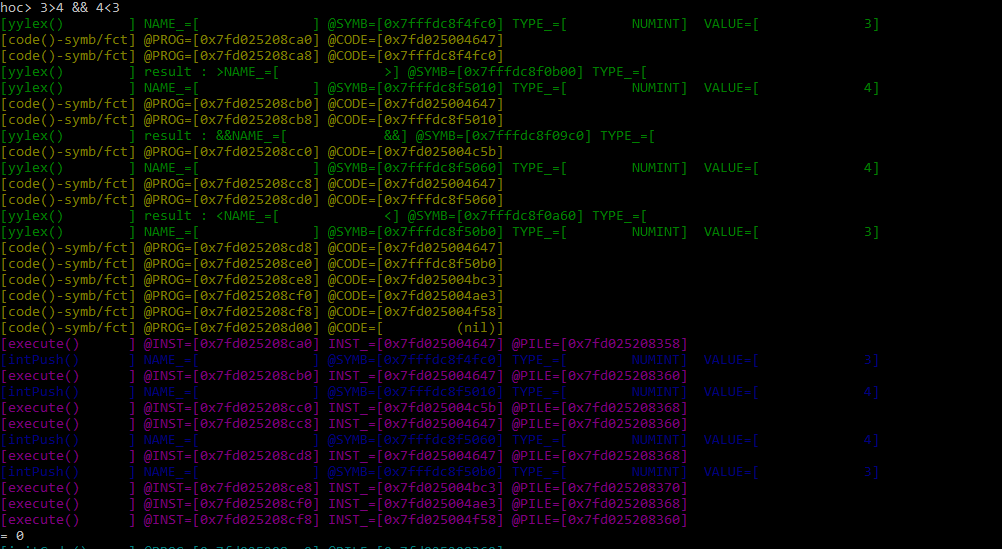
Définition des opérateurs booléen pour qu’ils puissent être reconnus en tant que symbole.

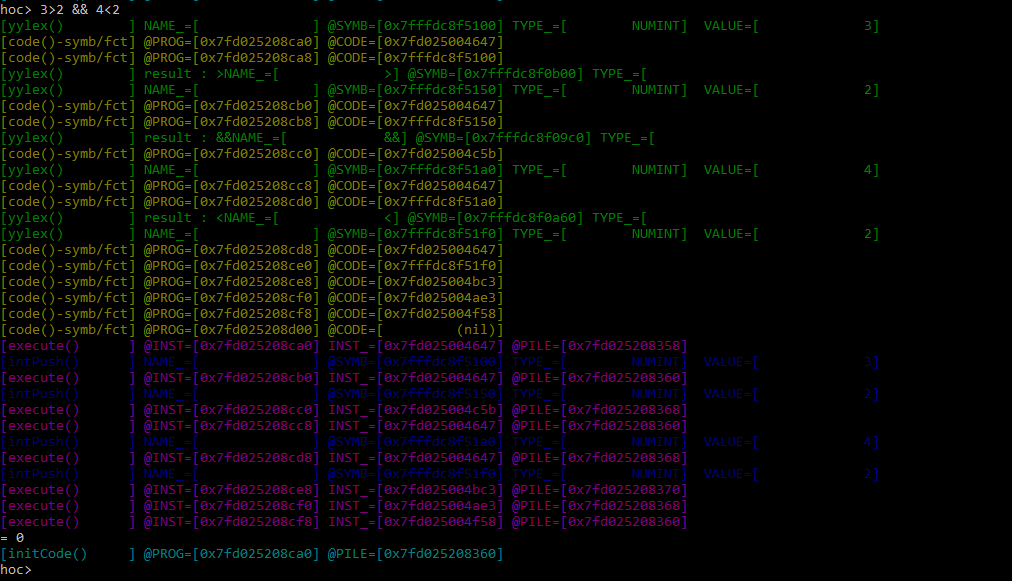
**Test de la génération de code et de l’exécution des symbolBool :**

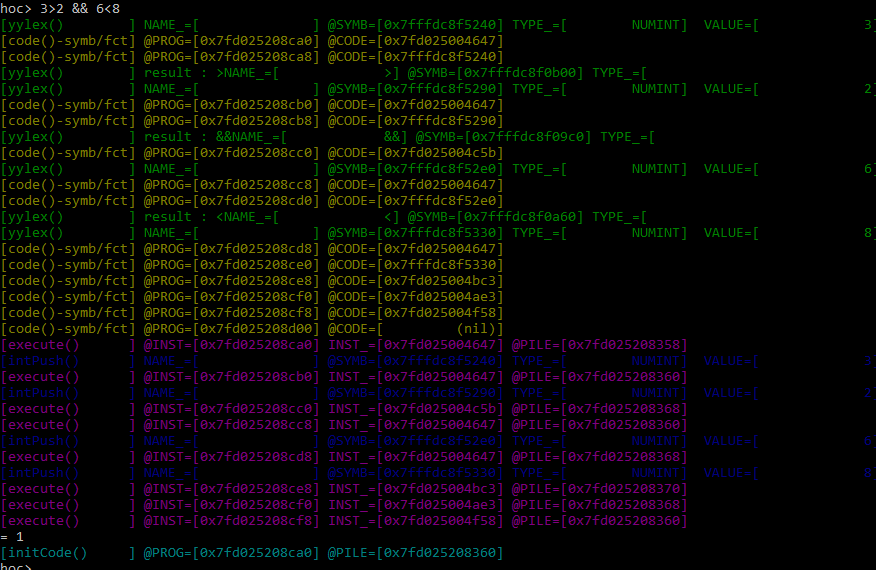
Test du NOT



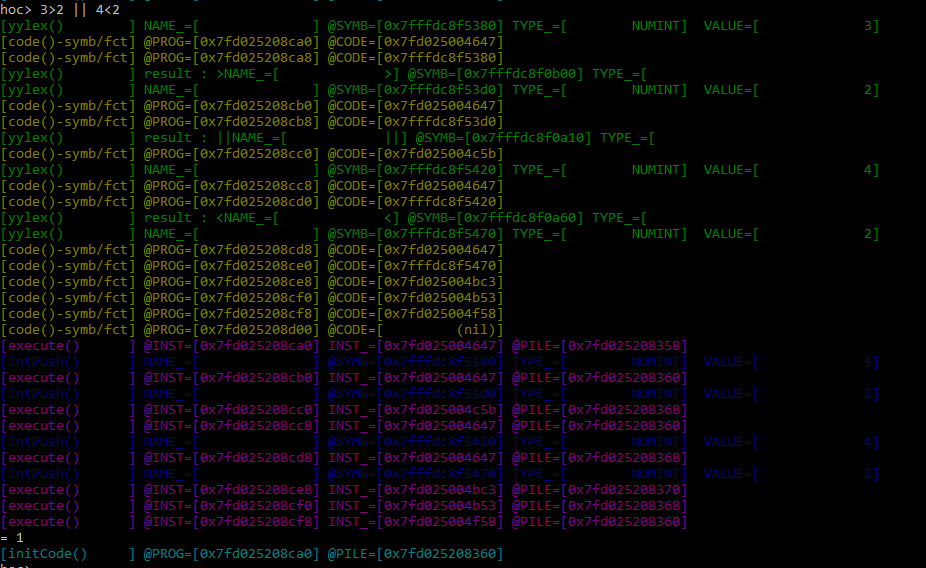
Test du ET

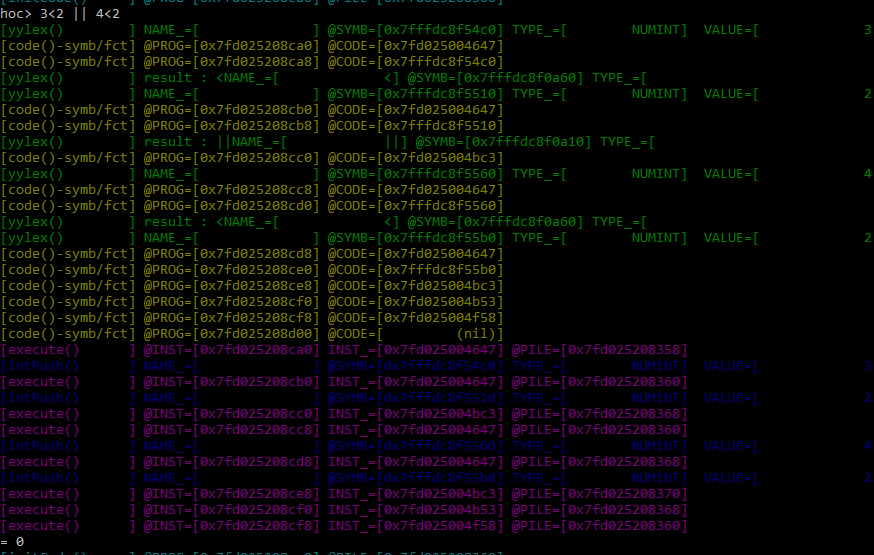


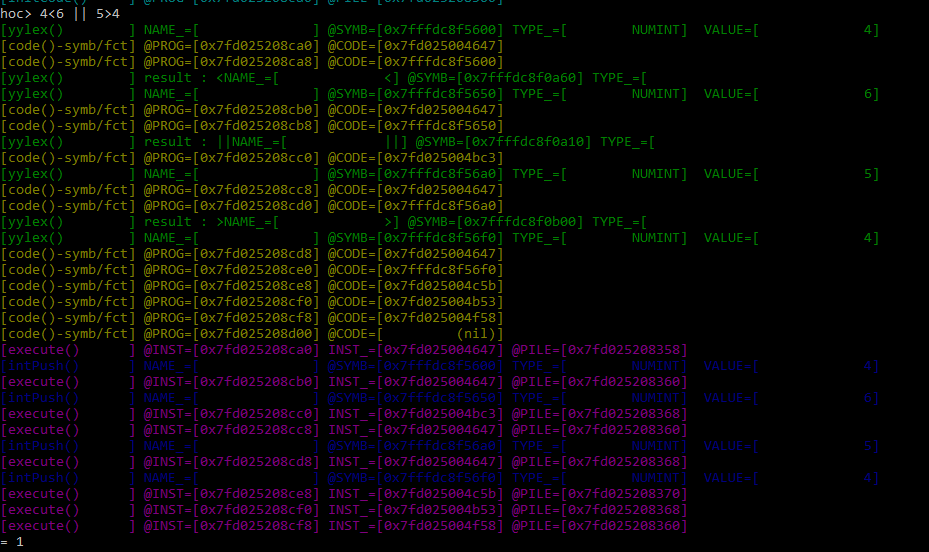




Test du OU :

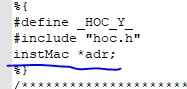


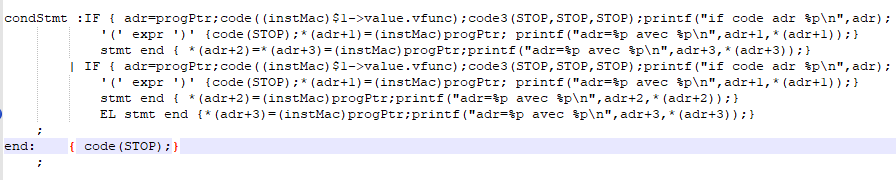


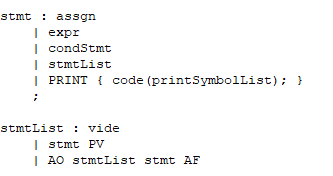


## Partie opérateur de condition :

**1 er : créer la reconnaissance pour les symbolCond :**





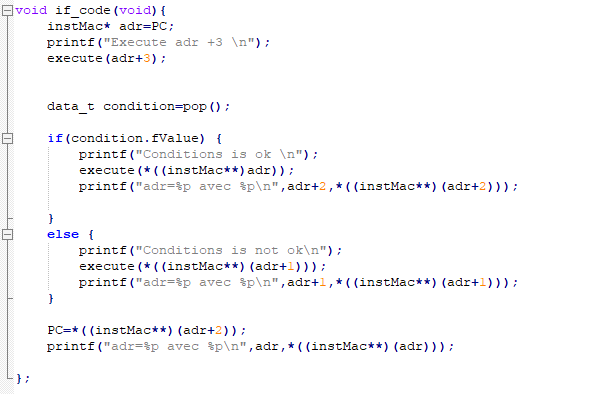


1er image : Création de la reconnaissance complète de la condition IF avec ou sans ElSE.

* Après le IF on initialise l’instruction machine « adr » avec le prochain emplacement libre, on lance la fonction if\_code et on crée 3 emplacements libre (code STOP \*3)
* Après « expr » on lance un STOP et on stock le prochain emplacement libre dans adr+1. (partie THEN)
* Après stmt end vu qu’il n’y a pas de « ELSE » on stocke dans adr+2 et adr+3 l’adresse du prochain emplacement (partie ELSE ET FIN)
* Avec un ELSE on sépare juste avec adr+2 et adr+3 indépendant. adr+2 pour le ELSE et adr+3 pour la FIN.

A chaque STOP on modifie l’adresse pour avoir la prochaine et on se sert de ces changements et avancement d’adresses pour la fonction if \_code.

**2 éme étape : Remplir la fonction if\_code :**

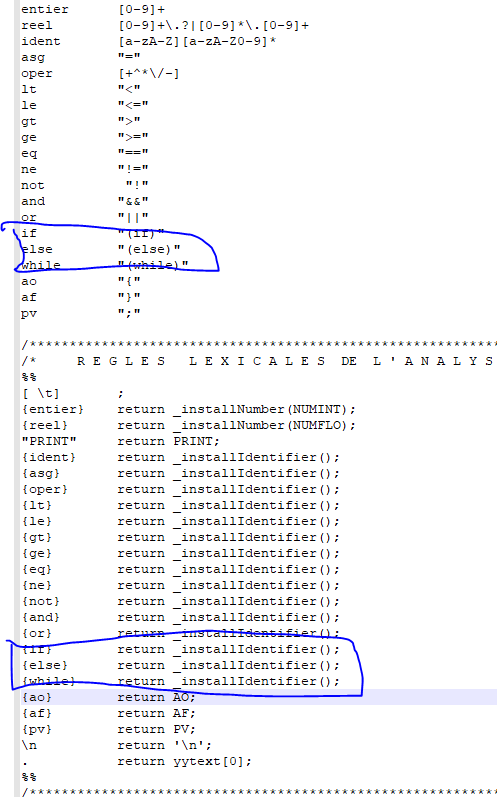


Fonctions if\_code :

* On récupère l’instruction machine du compteur d’exécution courant ( adr)
* On exécute(ou avance) l’instruction machine à l’adresse adr +3 pour pouvoir se positionner sur l’adresse de la condition (la partie IF)
* On récupère la condition du IF (fonction pop)
* On vérifie si la condition est vrai -> si vrai on exécute (ou avance) à l’instruction machine situé à « adr » pour exécuter la partie « THEN » (adr) de la condition
  + Si faux on exécute l’instruction machine de la partie « ELSE » (adr+ 1de la condition)
* A la fin on redéfinie (ou avance) l’instruction machine du compteur d’exécution avec adr+2 qui correspond à la « FIN » de la condition.

En général donc adr+3 correspond au IF, adr le THEN, adr+1 le ELSE et adr+2 la FIN.

**3éme étape : Ajout dans le lexème de l’analyseur et nouvelles règles lexical de l’analyseur (hoc.l):**

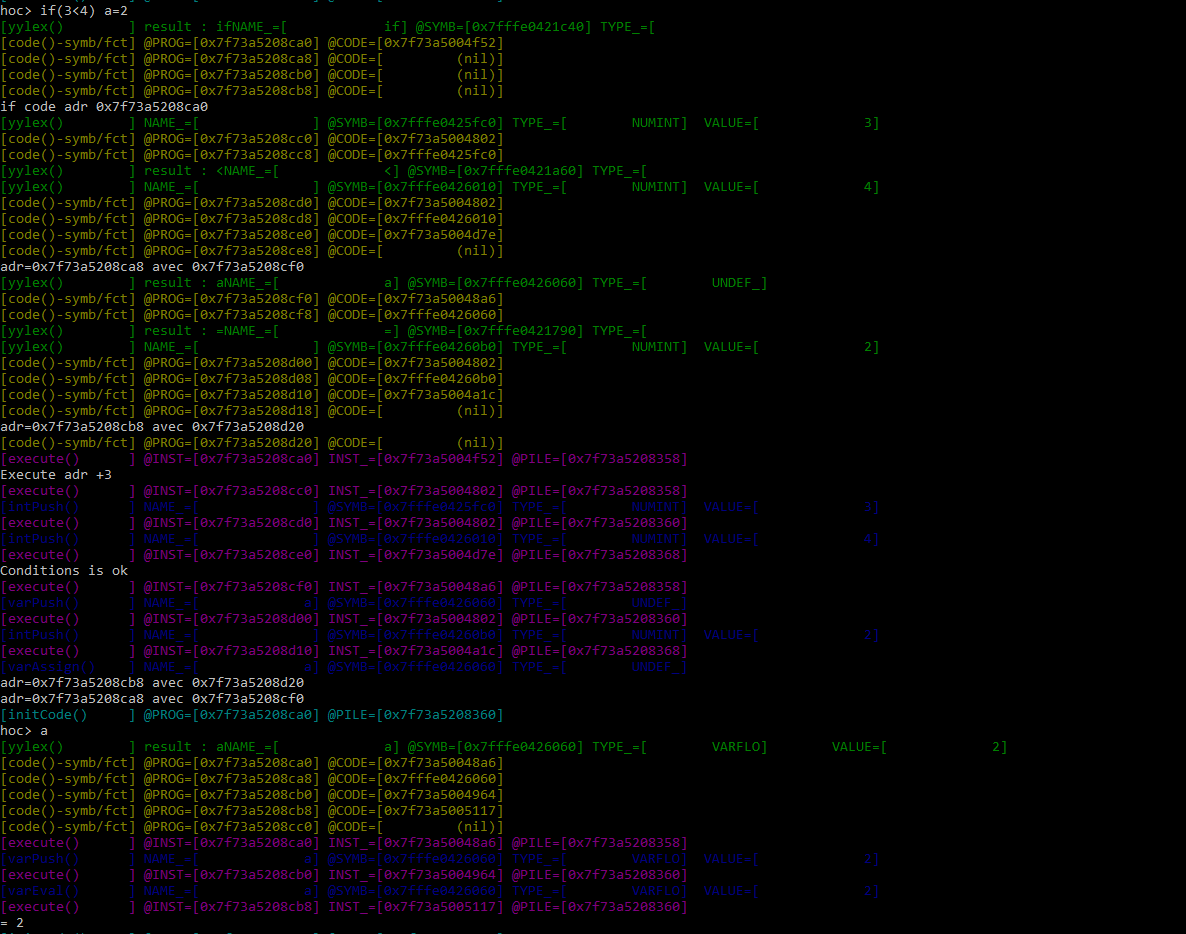


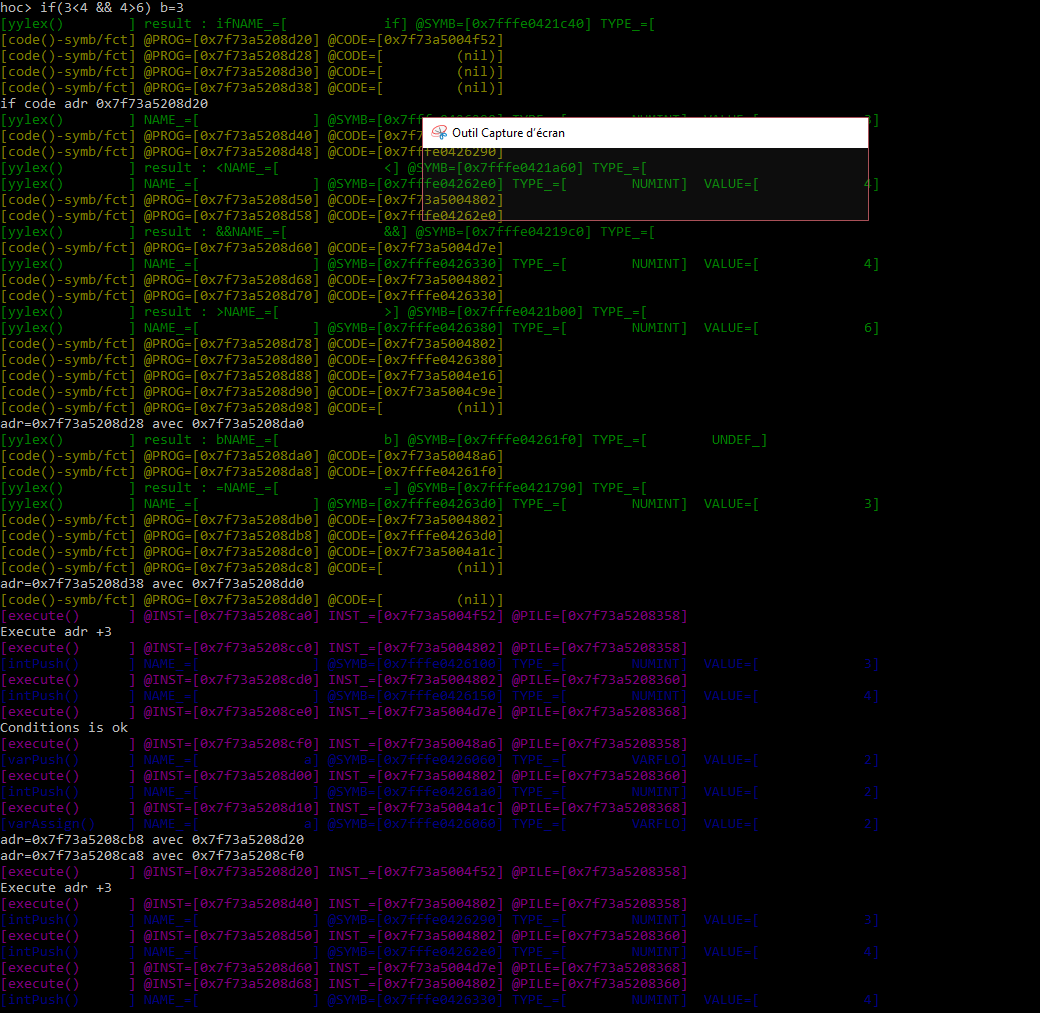
Définition des opérateurs de condition pour qu’ils puissent être reconnus en tant que symbole.

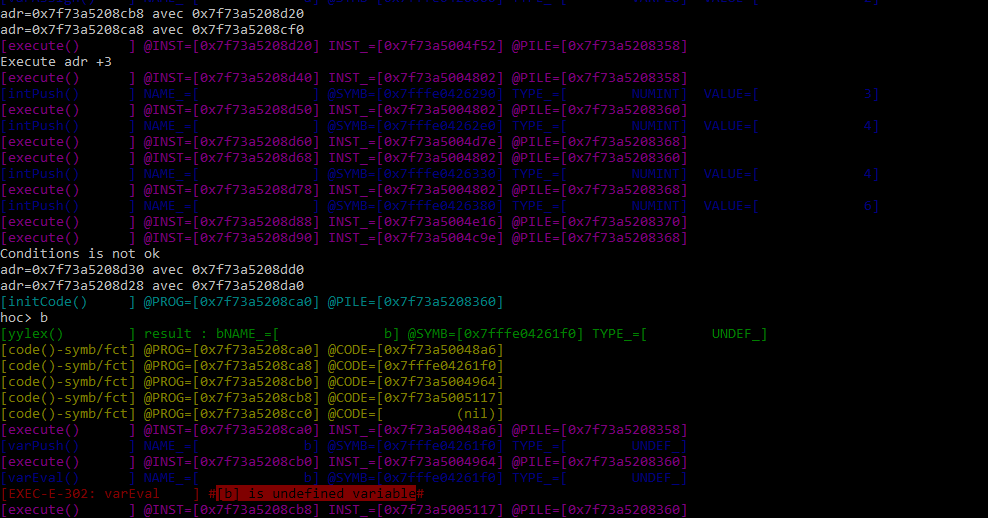
**Test de la génération de code et de l’exécution des symbolCond :**

Modèle  IF condition resultat et Modèle IF condition resultat else condition :

Modèle IF (condition) resultat et Modele IF (condition) resultat else condition :







## Traçabilité :