TD 06-07 – Analyse Syntaxique

Exercice 1. *analyse syntaxique*

1 Introduction

Dans ce TD, en s'appuyant sur l'analyse lexicale, on va réaliser l'analyse syntaxique du langage `FLO` à l'aide de `yacc` (`sly.Parser` en Python, `bisons` en C). Si vous travaillez en Python vous aurez à modifier les fichiers `analyse_syntaxique.py` et `arbre_abstrait.py`. Si vous travaillez en C, vous aurez à modifier les fichiers `analyse_syntaxique.y`, `arbre_abstrait.c` et `arbre_abstrait.h`.
Vous pourrez bien sûr revenir à tout moment sur l'analyse lexicale si vous vous rendez compte que la modifier vous permet de simplifier votre analyse syntaxique.
Note : Si vous travaillez en Python, vous pouvez ajouter la ligne

```
debugfile = 'parser.out'
```

dans la classe `FloParseur`. Quand vous ferez l'analyse syntaxique (ou quand vous ferez `make`) un fichier parser.out apparaîtra contenant la liste des règles et des informations utiles au débuggage.

2 Opérations arithmétiques

2.1 Ambiguïté et priorité

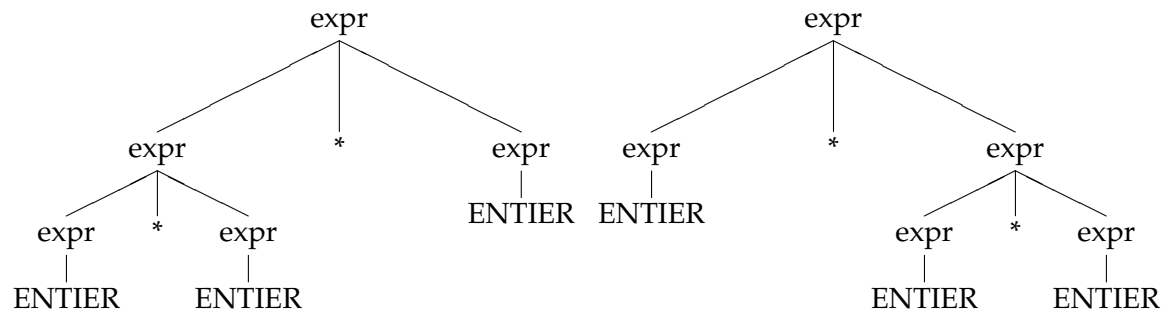
Le compilateur de base que l'on vous donne n'est capable de comprendre que des programme très simples : Une suite d'instructions `ecrire( exp );` où `exp` est une expression arithmétique faite de plus, de fois, de parenthèses et d'entiers. La grammaire est là suivante :

prog	$\longrightarrow$	*listeInstructions*
listeInstructions	$\longrightarrow$	*listeInstructions instruction* | *instruction*
instruction	$\longrightarrow$	*ecrire*
ecrire	$\longrightarrow$	ECRIRE (*expr*);
expr	$\longrightarrow$	ENTIER | (*expr*) | *expr* * *expr* | *expr* + *expr*

Cette grammaire pose deux problèmes. Le premier est qu'elle est ambigüe, la seconde est qu'elle ne respecte pas la priorité des opérateurs.

2.1.1 Ambiguïté

Considérons le mot ENTIER * ENTIER * ENTIER. Il est bien reconnu par la grammaire comme étant de type *expr* mais il correspond à deux arbres de dérivation différents :



L'arbre de gauche correspond au calcul (ENTIER * ENTIER) * ENTIER, celui de droite au calcul ENTIER * (ENTIER * ENTIER). Le standard est que si les opérations sont de même priorité, alors on les effectue de gauche à droite. [1]

L'ambiguïté crée des conflits de type décalage / réduction à `yacc` (*shift/reduce* en anglais) . En Python, il vous le signale via un message du type :

```
WARNING: 4 shift/reduce conflicts
```

En C, il vous le signale avec un message du type :

```
analyse_syntaxique.y: avertissement: 4 conflits par decalage/reduction
```

Au passage, si vous travaillez en C, tapez :

```
bison -Wcounterexamples analyse_syntaxique.y
```

pour que bison vous indique les 4 types de conflits.

Pour comprendre ce qu'il se passe il faut se rappeler que bison fait une analyse ascendante par décalage réduction.

Concentrons nous sur les 2 règles :

(1) *expr* $\longrightarrow$ ENTIER

(2) *expr* $\longrightarrow$ *expr* * *expr*

Regardons comment il analyserait ENTIER * ENTIER * ENTIER.

pile de symbole	terminaux non décalés	décalage,réduction (numéro)
	ENTIER * ENTIER * ENTIER	décalage
ENTIER	* ENTIER * ENTIER	réduction (1)
expr	* ENTIER * ENTIER	décalage
expr *	ENTIER * ENTIER	décalage
expr * ENTIER	* ENTIER	réduction (1)
expr * *expr*	* ENTIER	décalage / réduction (2)

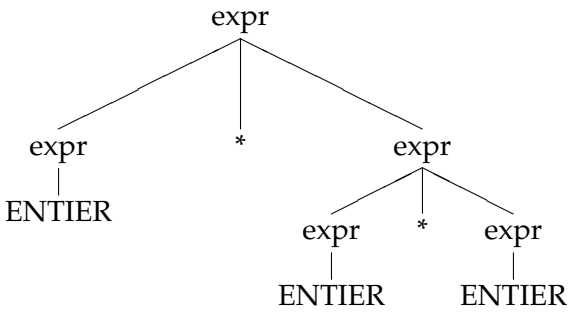
À la dernière étape il y a un conflit par décalage / réduction car il ne sait pas s'il doit transformer *expr* * *expr* en *expr* en appliquant la règle 2 ou décaler le prochain terminal * (car *expr* * pourrait être le début d'une règle *expr* * *expr*). Et c'est normal, la grammaire donnée est ambigüe et les deux options correspondent à des arbres de dérivations valides. [2] En cas de conflit décalage / réduction, bison favorise le décalage. La fin de l'analyse se passe donc comme ça :

1. La multiplication étant commutative, on pourrait penser que l'ordre choisi n'a pas d'importance. C'est un peu vrai mais par la suite on va introduire d'autre opérateurs non commutatif comme la division entière. De plus, on va plus tard introduire `lire()` qui pourra remplacer ENTIER et l'ordre des multiplication va impacter l'ordre dans lequel les entrées sont demandées à l'utilisateur.

2. Plus tard dans le TD, il est possible que vous trouviez des exemples où la grammaire n'est pas ambigüe mais où il y a quand même des conflits de décalage / réduction. C'est lié au fait que yacc ne reconnaît que les grammaires LR(1) : il ne peut voir qu'un seul terminal non décalé à l'avance.

pile de symbole	terminaux non décalés	décalage,réduction (numéro)
expr * *expr*	* ENTIER	**décalage** / réduction (2)
expr * *expr* *	ENTIER	décalage
expr * *expr* * ENTIER		réduction (1)
expr * *expr* * *expr*		réduction (2)
expr * *expr*		réduction (2)
expr		

L'arbre de dérivation obtenue est donc



Les opérations sont donc appliquées de droite à gauche : exactement ce que l'on ne veut pas. Une manière de corriger ce problème est d'introduire un nouveau non terminal et de changer la règle :

expr $\longrightarrow$ ENTIER | (*expr*) | *expr* * *expr* | *expr* + *expr*

en

expr $\longrightarrow$ *facteur* | *expr* * *facteur* | *expr* + *facteur*

facteur $\longrightarrow$ ENTIER | (*expr*)

Concentrons nous sur les 3 règles :

(1) *facteur* $\longrightarrow$ ENTIER

(2) *expr* $\longrightarrow$ *facteur*

(3) *expr* $\longrightarrow$ *expr* * *facteur*

Et regardons l'analyse de ENTIER * ENTIER * ENTIER

pile de symbole	terminaux non décalés	décalage,réduction (numéro)
	ENTIER * ENTIER * ENTIER	décalage
ENTIER	* ENTIER * ENTIER	réduction (1)
facteur	* ENTIER * ENTIER	réduction (2)
expr	* ENTIER * ENTIER	décalage
expr *	ENTIER * ENTIER	décalage
expr * ENTIER	* ENTIER	réduction (1)
expr * *facteur*	* ENTIER	réduction (3)

Notons qu'il n'y a pas de conflit ici :

— Si on utilise la règle 2 on se retrouve avec *expr* * *expr*, qui n'est un facteur d'aucune partie droite d'une règle. (et on ne pourrait pas revenir en arrière, *expr* ne pouvant pas se re-transformer en *facteur*)

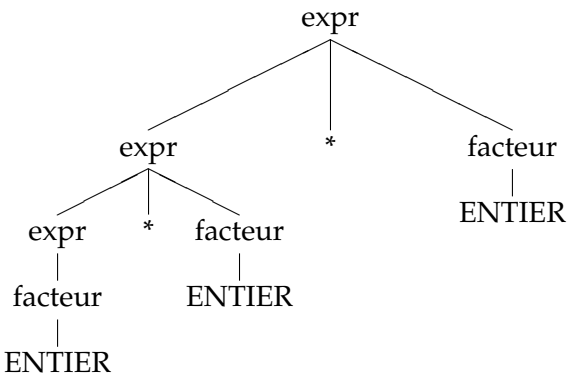
— Si on décale on se retrouve avec un *facteur* * qui n'est facteur d'aucune règle.

— La seule possibilité est d'appliquer la règle (3) et de transformer *expr* * *facteur* en *expr*.

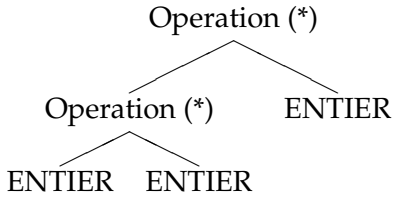
La fin de l'analyse :

pile de symbole	terminaux non décalés	décalage,réduction (numéro)
expr * *facteur*	* ENTIER	réduction (3)
expr	* ENTIER	décalage
expr *	ENTIER	décalage
expr * ENTIER		réduction(1)
expr * *facteur*		réduction(3)
expr		

L'arbre de dérivation obtenu est donc :



On a corrigé ambiguïté et les opérations sont donc appliqué de gauche à droite : ce que l'on voulait.
Bien sûre, le non-terminal *facteur* que l'on vient d'introduire ne servent qu'à l'analyse syntaxique et n'intéresse pas l'analyse sémantique. L'arbre abstrait produit par notre analyse syntaxique doit être de la forme :



★ **1.** Corrigez votre grammaire pour enlever l'ambiguïté et appliquer les opérations de gauche à droite. Il ne devrait plus rester de conflit décalage/réduction.

★ **2.** Créer un fichier `.flo` pour tester l'ordre des opération de votre analyse syntaxique.
Avec le contenu du fichier du type :

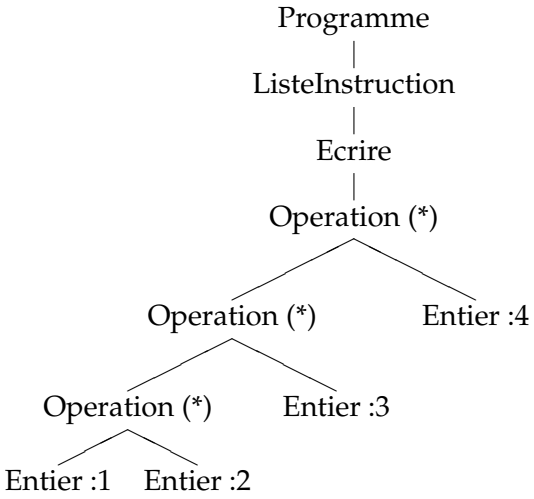
```
ecrire( 1 * 2 * 3 * 4);
```

la sortie de votre analyse syntaxique devrait être du type :

```
<programme>
 <listeInstructions>
  <ecrire>
   <operation "*" >
    <operation "*" >
     <operation "*" >
      [Entier:1]
      [Entier:2]
     </operation>
```

```
      [Entier:3]
     </operation>
     [Entier:4]
    </operation>
   </ecrire>
  </listeInstructions>
</programme>
```

ce qui correspond à :



2.1.2 Priorité des opérations

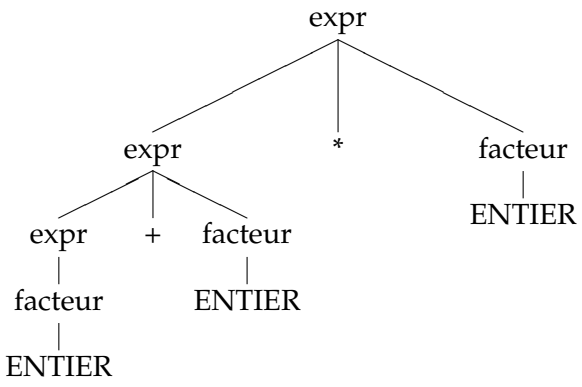
Le second problème de cette grammaire est qu'elle accorde la même priorité à tous les opérateurs. Or, la convention est d'effectuer d'abord les multiplications, et après les additions.

★ **3.** Créer un fichier `.flo` pour tester l'analyse syntaxique de l'expression $2 + 3 * 5$. Faites l'analyse sémantique dessus, compilez le et exécutez-le.

Avec le contenu du fichier du type :

```
ecrire(2+3*5);
```

Pour le moment, la grammaire donne la même priorité à tous les opérateurs, l'arbre syntaxique va être :



Le programme va donc calculer $(2 + 3) * 5 = 5 * 5 = 25$ à la place de $2 + (3 * 5) = 2 + 15 = 17$.
Le problème vient de la règle :

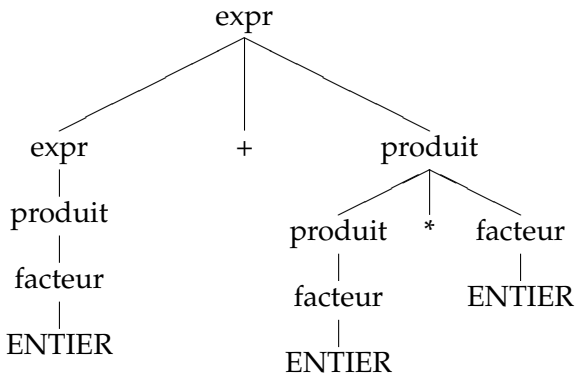
expr $\longrightarrow$ | *expr* * *facteur*

Cette règle fait que le programme va calculer (*expr*)**facteur* même si *expr* est dérivé en un mot contenant des +. Si par exemple *expr* est dérivé en 2+3 et *facteur* est 5, ça va nous donner *expr* dérivé en (2+3) * 5

On peut corriger le problème en introduisant un nouveau non terminal appelé *produit* qui ne peut pas être dérivé en un mot contenant des +.

expr $\longrightarrow$ *produit* | *expr* + *produit*
produit $\longrightarrow$ *facteur* | *produit* * *facteur*
facteur $\longrightarrow$ ENTIER | (*expr*)

Le nouvel arbre de dérivation de ENTIER + ENTIER * ENTIER va être :



★ **4.** Modifiez votre grammaire pour corriger la priorité de l'opérateur * sur l'opérateur +. Vous ne devez pas créer de nouveau conflits décalage/réduction ou réduction/réduction en faisant ça.

★ **5.** Testez de nouveau l'analyse syntaxique et l'exécution du programme sur plusieurs exemples `.flo`. Assurez-vous que les priorités soient bien respectées.

2.2 Autres opérateurs arithmétiques

On peut maintenant ajouter d'autres opérateurs arithmétiques. Il y en a 4 :

— la division / et le modulo % qui ont la même priorité que le $*$.
— Le moins $-$ comme opérateur binaire comme pour $7 - 8$ qui a la même priorité que le $+$.
— le moins $-$ comme opérateur unaire comme pour $-(7 + 3)$ (rien à gauche du moins). Dans ce cas là, sa priorité est égale à celle de la multiplication car c'est comme si on faisait $(-1) * (7 + 3)$.

★ **6.** Implémentez les nouveaux opérateurs, testez votre analyse syntaxique et vérifiez que vous ne créez pas de nouveaux conflits décalage/réduction.

Même si vous avez bien réalisé la partie analyse syntaxique de l'implémentation de ces nouveaux opérateurs, pour le moment votre compilateur ne peut pas compiler les programme avec des moins, des divisions ou des modulos car la partie génération de code correspondante n'est pas encore implémenté.

★ **7. (optionnelle)**. Si vous voulez compiler avec les nouveaux opérateurs, vous pouvez prendre de l'avance et regarder la partie génération de code en vous inspirant de ce qui a été fait dans la fonction `gen_operation` en Python ou `nasm_operation` en C pour le + et le *.

2.3 Autres facteurs

En plus des entiers, les facteurs pourraient êtres :
- Le contenu d'une variable de la forme `nomVariable` .
- L'instruction `lire()`.
- Le retour d'un appel de fonction de la forme `nomFonction(exp1,exp2,...expk)` où `expi` est une expression (*expr*).

★ **8.** Ajouter ces 3 nouveaux types de *facteurs* dans votre grammaire. Vous aurez besoin d'ajouter des équivalents dans la partie `arbre_abstrait` de votre programme .

3 Expression booléennes

En plus des types entiers, le langage `flo` autorise des valeurs booléennes. Une valeur booléenne peut être :
- un mot clé `Vrai` ou `Faux`,
- le résultat d'une comparaison entre deux valeurs entières par exemple $a == 10$.
- le résultat des 3 opérations logiques a ou b, a et b, non a.
- Le retour d'un appel de fonction ou le contenu d'une variable.

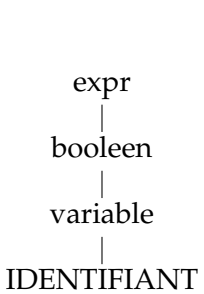
3.1 Problème de type

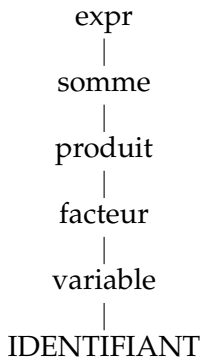
La fonction `ecrire` admet comme argument aussi bien des expressions à valeur entière (dans ce cas là, elle affiche le nombre), que des expressions à valeur booléenne (dans ce cas là elle affiche 1 ou 0 pour vrai ou faux).
On pourrait être tenté de faire quelque chose dans ce goût là :

expr	$\longrightarrow$	*somme* | *booleen*
somme	$\longrightarrow$	*produit* | *somme* + *produit*
produit	$\longrightarrow$	*facteur* | *produit* * *facteur*
facteur	$\longrightarrow$	*variable* | ...
variable	$\longrightarrow$	IDENTIFIANT
booleen	$\longrightarrow$	VRAI | FAUX | variable | ...

Le problème est que cette grammaire est ambigüe. En effet,
- une expression peut-être une expression booléenne ou une expression entière,
- une expression booléenne peut-être (entre autres) une variable
- une expression entière peut être (entre autres) une variable aussi.

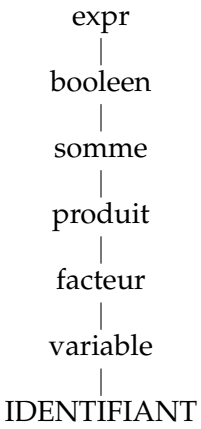
Par exemple, pour le programme suivant `ecrire(maVariable);`, `maVariable` est interprété comme un IDENTIFIANT, lui-même interprété comme *expr*. Mais IDENTIFIANT peut être dérivé en *expr* de deux manières différentes :





La manière la plus facile (mais contre-intuitive) de régler ce problème est de faire que somme se dérive de booléens :

expr	$\longrightarrow$	*booleen*
booleen	$\longrightarrow$	VRAI | FAUX | *somme* | ...
somme	$\longrightarrow$	*produit* | *somme* + *produit*
produit	$\longrightarrow$	*facteur* | *produit* * *facteur*
facteur	$\longrightarrow$	*variable* | ...
variable	$\longrightarrow$	IDENTIFIANT



Cette nouvelle grammaire n'est pas abigüe. En revanche, elle va autoriser des absurdités du type.

```
ecrire(non 5);
```

Malheureusement, écrire les règles de manière à éviter ce genre de problème sans pour autant qu'elle soit ambigüe la rendrait très compliquée.
Si on veut détecter ce genre de problème le plus tôt possible, le plus simple est de faire des tests au niveau des actions sémantiques. Par exemple, quand on réalise l'action associée à la règle *booleen* $\longrightarrow$ NON *booleen*, on en profite pour vérifier que *booleen* n'est pas de type entier.
Mais de toute façon, certains problèmes ne seront pas détectés pendant l'analyse syntaxique. Par exemple des problèmes du type :

```
entier a = 8;
...
ecrire(non a);
```

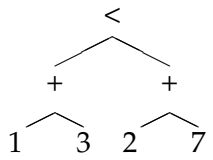
ne peuvent pas être résolu avec une grammaire hors-contexte. Elle implique de mémoriser le nom des variables et leur type pour pouvoir savoir plus tard si on peut les utiliser dans une négation ou une addition.
Donc en résumé, les problèmes de types seront détectés pendant l'analyse sémantique.

★ **9.** Modifiez votre grammaire pour y ajouter les expressions booléennes. Assurez vous de ne pas la rendre ambigüe.

3.2 Comparateur

En `FLO`, on peut créer une expression booléenne en comparant deux expressions entières grâce au 6 opérateurs de comparaison (==,!=,<,<=,>,>=).
Au niveau de la précédence, ces opérations doivent être effectuées après toutes les opérations arithmétiques. Normalement, étant donné la façon dont nous avons défini les expressions booléennes, la précédence sera déjà respectée.

En effet 1 + 3 < 2 + 7 doit être interprété comme



Normalement, ajouter les comparaisons ne va pas rendre la grammaire ambigüe car le langage `FLO` n'autorise pas les opérations du type 2 < 3 < 4.

★ **10.** Ajouter les comparateurs à votre grammaire et à votre arbre abstrait. Dans votre arbre abstrait, les comparaisons peuvent être stocké dans la même structure que vos opérations arithmétiques ou dans de nouveaux objets. À vous de voir.

3.3 Opérateurs logiques

En `FLO`, il existe 3 opérateurs logiques :
- la négation : `non a`
- la conjonction : `a ou b`, la disjonction : `a et b`.

Pour les mêmes raisons que nous avons vu pour les multiplications et l' addition (ambiguïté et priorité des opérateurs) il est hors de question de faire ça :

booleen $\longrightarrow$ VRAI | FAUX | *somme* | NON *booleen* | *booleen* OU *booleen* | *booleen* ET *booleen*

La solution sera similaire à celle employée pour l'addition et la multiplication.
Au niveau de la priorité, la négation est prioritaire sur la conjonction qui est elle-même prioritaire sur la disjonction.

★ **11.** Ajouter les opérateurs logiques à votre grammaire et à votre structure d' arbre abstrait.

4 Autres instructions

Pour le moment la seule instruction autorisée par votre compilateur est l'instruction *ecrire*. Nous allons ajouter d'autres instructions.
- Déclaration : `TYPE IDENTIFIANT;` où `TYPE` peut-être booléen ou entier.
- Affectation `IDENTIFIANT = expr;` (*expr* doit être du même type que la variable identifié par `IDENTIFIANT` mais pas contrôlé pendant l'analyse syntaxique).
- Déclaration-Affectation `TYPE IDENTIFIANT = expr;` (TYPE doit correspondre à *expr* mais pas contrôlé pendant l'analyse syntaxique).
- Instruction conditionnelle :

```
SI( expr ){
listeInstructions
}
SINON_SI(expr){
listeInstructions
}
...
SINON{
listeInstruction
}
```

où *expr* est une expression booléenne (pas contrôlé pendant l'analyse syntaxique) et avec les `sinon si` et `sinon` optionnels.

— Instruction boucle `TANTQUE(`*`expr`*`) { listeInstructions }` *expr* est une expression booléenne (pas contrôlé pendant l'analyse syntaxique).
— Instruction `retourner` *expression* `;` (dans une fonction uniquement, mais le fait d'être dans une fonction n'est pas contrôlée pendant l'analyse syntaxique)
— Appel de fonction `nomFonction(arg1,...,argn);` (Similaire à l'appel de fonction qui donne une expression, mais la valeur de retour est ignorée.)

★ **12.** Ajouter toutes ces instructions à votre grammaire et à votre structure d' arbre abstrait.

5 Définitions Fonctions

Jusqu'à là les programme `FLO` compilés étaient de la forme :
listeInstructions.
Ils seront maintenant de la forme plus générale :

`listeFonctions listeInstructions`

où *listeFonctions* et une suite (potentiellement vide) d'élément *fonction*.
Une *fonction* est définie comme suit :

`TYPE IDENTIFIANT(TYPE IDENTIFIANT, ..., TYPE IDENTIFIANT){`
`listeInstructions` `}`

`TYPE IDENTIFIANT, ..., TYPE IDENTIFIANT` est la suite d'argument de la fonction. Une fonction peut ne pas avoir d'arguments.

★ **13.** Essayez d'implémenter la déclaration de fonction dans votre grammaire. Attention : le langage `FLO` autorise les programmes sans fonctions.