

Projet : Réalisation d'un évaluateur - Mini Excel

Module : Compilation – 2CS

Travail de :

Belfodil Adnene

I. Introduction	3
II. Développement de l'évaluateur	4
1. Objet du Tableur	4
2. Schéma globale du comportement de l'évaluateur.....	4
3. Analyseur lexical.....	6
3.1. Description de l'analyseur lexical	6
3.2. développement de l'analyseur lexical	7
4. Analyseur Syntaxique	8
4.1. Grammaire naturelle.....	9
4.2. Grammaire LL(1)	10
4.3. Table d'analyse LL(1).....	11
4.4. Analyse descendante récursive	14
5. Analyseur Sémantique	15
5.1. Opérations binaire et unaire	16
5.2. Attribut utilisées	17
5.3. Variables ou cellules.....	19
5.4. Schéma de traduction	22
5.5. Amélioration des procédure de descente récursive	28
6. Gestion d'erreur	31
7. Interface de l'évaluateur.....	32
III. Tests et Fonctionnalités.....	34
1. Fonctionnalités de l'application.....	34
2. Tests de fonctionnement.....	36
2.1. Test d'évaluation d'une expression arithmétique	36
2.2. TEST d'évaluation d'une opération double point.....	36
2.3. Test d'evaluation d'une expression logique.....	36
2.4. Test de manipulation des matrices	36
2.5. Test de la gestion d'erreurs.....	37
2.5.1. TEST d'erreur de référencement Cyclique	37
2.5.2. TEST d'erreur de type d'arguments	37
Conclusion.....	38

I. INTRODUCTION

A l'issue du Module Compilation que nous avons eu l'occasion d'aborder cette année, nous avons été amenée à développer un évaluateur d'expressions arithmétiques sous forme d'un tableur (*ressemblant à MS-EXCEL*), le développement d'un tel outil nous a permis ainsi d'exploiter les compétences acquises et d'approfondir nos connaissances de ce module auquel nous suscitons un grand intérêt.

II. DEVELOPPEMENT DE L'EVALUATEUR

1. OBJET DU TABLEUR

L'objectif de ce petit projet, est de développer un tableur ayant les fonctionnalités basiques de *MS-Excel*, nous énumérons ci-dessus les fonctionnalités basiques de cet évaluateur :

1. **Evaluer les expressions arithmétiques** : comportant tous les opérations de bases multiplication, addition, soustraction, division ...
2. **Evaluer les expressions logiques** : l'évaluateur doit permettre d'évaluer une comparaison entre les résultats de deux expressions arithmétiques. Ainsi qu'à l'évaluation de certaines fonctions logiques.
3. **Evaluer les fonctions arithmétiques et booléennes** : Les expressions arithmétiques ou logiques doivent comporter la possibilité d'introduire des appels aux fonctions arithmétiques (*cos, sin, puissance...*), statistiques (*moyenne, variance ...*) ou booléennes (*et, ou, non ...*).
4. **Evaluer l'expression conditionnelle** : l'évaluateur doit pouvoir comporter l'expression *Si (Condition, Chaine1, Chaine2)*, ou cette dernière retourne selon la condition la première ou la deuxième chaîne, notons bien aussi que la chaîne peut être une expression quelconque arithmétique ou logique, ainsi dans le cas où ces chaînes sont des expressions ils devront être évalués avant d'être affichés.
5. **Amélioration** : les fonctionnalités ci-dessus sont ceux nécessaires, il faudra noter que l'évaluateur développé durant ce petit projet permet aussi faciliter la manipulation des données avec des nouvelles opérations ainsi que la manipulation des matrices.

2. SCHEMA GLOBALE DU COMPORTEMENT DE L'EVALUATEUR

Comme on a pu le voir à l'issue du cours de compilation, le petit projet consiste à développer un analyseur sémantique dirigé par la syntaxe. ainsi si nous voulons résumer les étapes de l'évaluation d'une expression quelconque nous aurons le schéma ci-dessous :

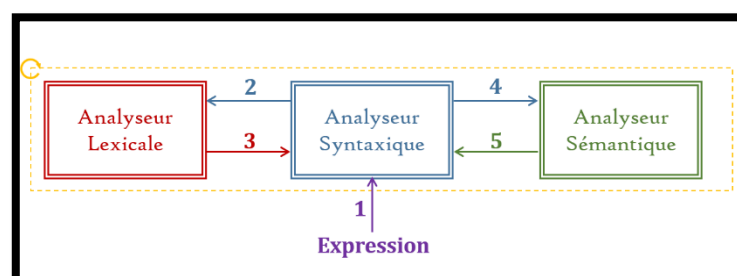


FIGURE 1 - SCHEMA GENERALE D'UN ANALYSEUR DIRIGE PAR LA SYNTAXE

Le schéma présenté dans la *figure 1*, présente un déroulement général d'un évaluateur dirigé par la syntaxe, ou le noyau est un **analyseur syntaxique**, lors de l'acquisition de l'expression (*étape 1*) l'analyseur syntaxique fait appel à l'analyseur lexicale pour reconnaître la première entité lexicale de l'expression (*étape 2*) qui à son tour reconnaît lettre par lettre les caractères de la chaîne en entrée jusqu'à reconnaître le premier symbole puis le retourne à l'analyseur lexicale (*étape 3*), à la réception de l'analyseur syntaxique de l'entité lexicale, l'interpréteur va devoir évaluer la sous chaîne selon le schéma de traduction (*défini ci-après dans le rapport*) pour pouvoir continuer l'analyse, pour pouvoir évaluer l'analyseur syntaxique fait appel à l'analyseur sémantique (qui n'est pas dissocié réellement de l'analyseur syntaxique) puis retourne la valeur correspondante, ce schéma se répète jusqu'à ce que l'analyseur soit capable de définir une valeur pour l'expression donnée en entrée, si c'est possible une valeur numérique ou booléenne est retournée, sinon une erreur est déclarée.

Il est à noter que l'abstraction de notre évaluateur définie dans la *figure 1* n'est pas utilisée directement comme elle est, ainsi le schéma est évolué dans cette figure suivante qui va expliquer en détail le comportement de notre évaluateur.

Ce schéma va définir ainsi les différentes classes d'interprétation de notre évaluateur, une division pareille est définie pour pouvoir maintenir et améliorer l'évaluateur.

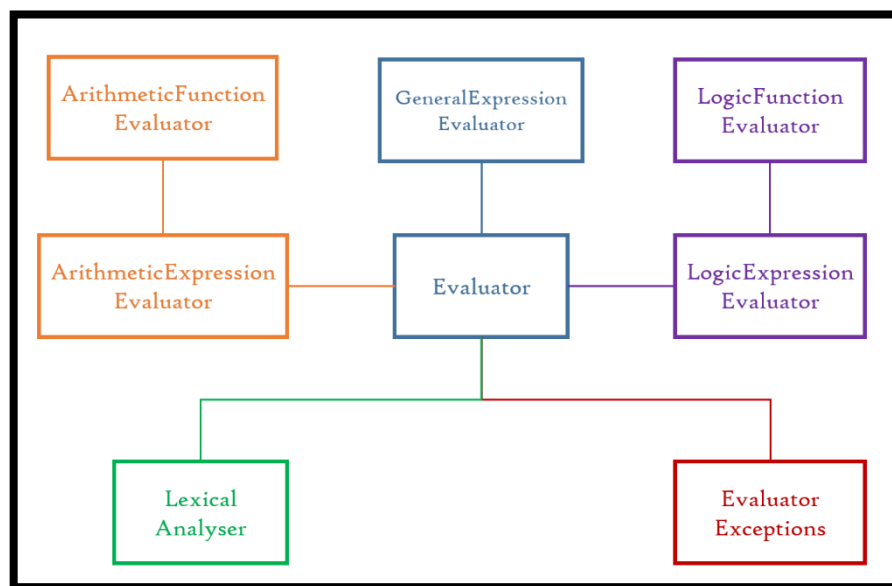


FIGURE 2 - COMPOSANTS DE L'EVALUATEUR

Comme les noms des composantes l'indiquent, chaque sous-évaluateur a un travail bien précis concernant une partie bien spécifique de l'interpréteur, nous abordons au fur et à mesure la fonction de chaque composant dans les titres qui vont suivre.

3. ANALYSEUR LEXICAL

3.1. DESCRIPTION DE L'ANALYSEUR LEXICAL

L'analyseur lexical, a pour rôle de reconnaître un symbole lexical d'une chaîne donnée, le premier symbole reconnu est retourné à l'évaluateur faisant requête d'une lecture d'un symbole, comme définie auparavant dans le titre précédent, l'évaluation d'une expression quelconque en entrée n'est pas faite en deux passes, mais dans une seule passe. Cela veut dire qu'on ne parcourt pas l'expression d'abord pour classer et reconnaître tous les symboles puis une seconde passe pour l'analyse lexicale, mais dans un seul tour, où au fur et à mesure que l'analyseur syntaxique (*voir sémantique*) évolue dans l'interprétation il sollicite l'analyseur lexicale pour extraire un symbole, à l'extraction on avance vers le prochain symbole.

Cette méthode est choisie car elle est plus performante vis-à-vis une grande expression, car une fois qu'il y'a une erreur au niveau syntaxique l'évaluateur ne va pas perdre le temps en essayant d'avoir tous les entités lexicales après une phase erronée de l'analyse syntaxique.

Pour pouvoir réaliser l'analyseur, nous avons eu usage des **expressions régulières** constituant l'outil majeur d'une analyse lexicale quelconque. Ainsi nous avons défini une expression régulière pour les symboles utilisés étant classées dans le tableau suivant :

Symbole	Expression régulière associée
Nombre réel	"^[0-9]+(\\.[0-9]+)?"
Opérations	"^\\+ ^\\- ^* ^\\/ ^"
Parenthèses	"^\\(^\\) "
Comparaison	"^>=? ^<=? ^== ^!="
Fonctions	"^[a-z]+[0-9]*\\(\\)"
Identificateurs de variables	"^[a-z][a-z]?[0123456789]+"
Constantes booléennes	"^true ^false"
Opérations spéciaux	"^:_ ^:\\+ ^:* ^:\\^ ^:"
Symbole de traitement	"^# ^, ^="

Note : les fonctionnalités des opérateurs spéciaux vont être définies plus tard durant l'analyse syntaxique, tandis que la virgule a pour rôle de séparer les

paramètres d'une fonction arithmétique ou logique de plusieurs paramètres, le symbole « = » permet de commencer l'énoncé d'une expression, et le # est un artefact ajouté après l'introduction de la chaîne pour reconnaître la fin de l'expression.

3.2. DEVELOPPEMENT DE L'ANALYSEUR LEXICAL

Pour pouvoir développer cet analyseur nous avons utilisé les expressions régulières dans **Java**, livré dans la **classe Patterns** qui permet les manipulations de bases d'une expression régulière, l'analyseur lexical de cet évaluateur manipule les classes suivantes :

- **Symbole** : définit une entité lexicale, ainsi il contient la **chaîne de caractère** lui représentant, le **type du symbole** et son **indice** dans l'expression, cette classe livre la fonction importante **value()** qui permet de retourner la valeur du symbole en un objet **CombinedReturnObject** adapté selon le type de symbole (retournant la valeur du réel en cas où le type du symbole représente un double, la constante booléenne vrai ou faux en cas où le symbole est une constante booléenne, sinon la chaîne elle-même).
- **SymboleType** : énumération définissant tous les types de symboles reconnus par l'analyseur, fournissant aussi des fonctions de reconnaissances par expression régulières des types de symboles selon leurs chaînes.
- **LexicalAnalyser** : noyau fonctionnel de l'analyseur lexical, les attributs qu'il manipule sont ceux suivants :
 1. **Expression** : expression traitée pour l'analyse, le traitement impliquant la suppression des séparateurs, la réduction des caractères en minuscules, et l'ajout de l'artefact # indiquant la fin de symbole.
 2. **tc** : curseur ou indice courant de l'analyse sur l'expression, qui est incrémenté par la longueur de l'entité lexicale reconnue à chaque requête d'avancement vers le prochain symbole.
 3. **actualsymbol** : symbole actuel de l'analyse qui est remplacé après chaque avancement du curseur par le symbole suivant.

Cet analyseur fournit trois fonctions importantes utilisées par l'analyseur syntaxique qui sont :

1. **startevaluation(expression)** : initialise l'analyse lexical d'une expression et donc met à zéro le curseur, et met à jour l'**attribut expression** par l'expression donnée en paramètre.
2. **actualSymbol()** : retourne le symbole actuel de l'analyse.

3. **nextSymbol()** : permet de reconnaître l'entité lexicale qui suit celle courante et par conséquent l'avancement du curseur, cette fonction utilise la classe `Pattern` pour reconnaître le premier symbole respectant une des expressions régulières définies auparavant, puis crée un nouveau **symbole** de cette partie reconnue respectant le type définie dans la classe **SymboleType**.

Note : Veuillez consultez le code en fichier joint détaillé pour visualiser les différentes classes utilisées pour l'élaboration de ce petit projet.

4. ANALYSEUR SYNTAXIQUE

Comme expliqué auparavant, l'analyseur syntaxique est découpé en plusieurs composants chacun assurant une fonctionnalité bien précise, cette décomposition n'a pour but que la séparation des entités afin de pouvoir mieux maîtriser le développement de l'évaluateur, mais tous ces composants dérivent en fait d'une seule grammaire globale décrivant l'analyseur syntaxique globale. Ainsi dans ce qui va suivre nous allons présenter la grammaire, ainsi que la table d'analyse.

Il est à noter comme dit dans l'énoncé du projet que la grammaire manipulée est **LL(1)**, mais pour simplifier la description de l'analyseur nous allons d'abord donner la grammaire naturelle (*qui sera utilisée pour décrire le schéma de traduction*), puis nous donnerons la grammaire traitée transformée en **LL (1)**.

Note : Toutes les grammaires vont être présentées sous la forme BNF (Backus-Naur form).

4.1. GRAMMAIRE NATURELLE

Nous allons présenter maintenant la grammaire naturelle permettant de présenter toutes productions possibles de notre langage, il est à noter que certains symboles (ou certaines productions ne vont pas être directement compris, on leur attribuera un sens au niveau sémantique, ces derniers font objet de l'amélioration de notre tableur).

```
<Axiome> ::= '=' <General> '#'  
  
<General> ::= <Expression> 'CompareOp' <Expression> | <Expression>  
  
<Expression> ::= <Expression> '+' <Term> | <Expression> '-' <Term>  
                | <Term>  
  
<Term> ::= <Term> '*' <Factor> | <Term> '/' <Factor> | <Factor>  
  
<Factor> ::= <Factor> '^' <Exponent> | <Exponent>  
  
<Exponent> ::= 'number' | '-' <Exponent> | '(' <General> ')'  
                | <LogicalExpression> | <ArithmeticFunction>  
                | 'CellName' | 'CellName' 'DoublePointOp' 'CellName'  
  
<ArithmeticFunction> ::= 'ArithmeticFnctIdentifier' '(' <ListOfExpression> ')'  
  
<ListOfExpression> ::= <ListOfExpression> ',' <Expression>  
                | <Expression>  
  
<LogicalExpression> ::= <LogicalFunction> | 'true' | 'false'  
  
<LogicalFunction> ::= 'LogicFnctIdentifier' '(' <ListOfGeneral> ')'  
  
<ListOfGeneral> ::= <ListOfGeneral> ',' <General> | <General>
```

Note : 'CompareOp' désigne les symboles terminaux utilisées pour la comparaison à savoir '=', '>=', '<=', '!=', '>', '<', et 'DoublePointOp' désigne les opérations commençant par un ':' utilisées pour définir un champ de cellules.

4.2. GRAMMAIRE LL(1)

1. $\langle \text{Axiome} \rangle ::= '=' \langle \text{General} \rangle \#'$
2. $\langle \text{General} \rangle ::= \langle \text{Expression} \rangle \langle \text{General2} \rangle$
3. $\langle \text{General2} \rangle ::= \text{CompareOp}' \langle \text{Expression} \rangle$
4. $\langle \text{General2} \rangle ::= \epsilon$
5. $\langle \text{Expression} \rangle ::= \langle \text{Term} \rangle \langle \text{Expression2} \rangle$
6. $\langle \text{Expression2} \rangle ::= '+' \langle \text{Term} \rangle \langle \text{Expression2} \rangle$
7. $\langle \text{Expression2} \rangle ::= '-' \langle \text{Term} \rangle \langle \text{Expression2} \rangle$
8. $\langle \text{Expression2} \rangle ::= \epsilon$
9. $\langle \text{Term} \rangle ::= \langle \text{Factor} \rangle \langle \text{Term2} \rangle$
10. $\langle \text{Term2} \rangle ::= '*' \langle \text{Factor} \rangle \langle \text{Term2} \rangle$
11. $\langle \text{Term2} \rangle ::= '/' \langle \text{Factor} \rangle \langle \text{Term2} \rangle$
12. $\langle \text{Term2} \rangle ::= \epsilon$
13. $\langle \text{Factor} \rangle ::= \langle \text{Exponent} \rangle \langle \text{Factor2} \rangle$
14. $\langle \text{Factor2} \rangle ::= '^' \langle \text{Exponent} \rangle \langle \text{Factor2} \rangle$
15. $\langle \text{Factor2} \rangle ::= \epsilon$
16. $\langle \text{Exponent} \rangle ::= \text{'number'}$
17. $\langle \text{Exponent} \rangle ::= '-' \langle \text{Exponent} \rangle$
18. $\langle \text{Exponent} \rangle ::= '(' \langle \text{General} \rangle ')'$
19. $\langle \text{Exponent} \rangle ::= \langle \text{LogicalExpression} \rangle$
20. $\langle \text{Exponent} \rangle ::= \langle \text{ArithmeticFunction} \rangle$
21. $\langle \text{Exponent} \rangle ::= \langle \text{Variable} \rangle$
22. $\langle \text{Variable} \rangle ::= \text{'CellName'} \langle \text{Variable2} \rangle$
23. $\langle \text{Variable2} \rangle ::= \text{'DoublePointOp'} \text{'CellName'}$
24. $\langle \text{Variable2} \rangle ::= \epsilon$
25. $\langle \text{ArithmeticFunction} \rangle ::= \text{'ArithmeticFncIdentifier'} (' \langle \text{ListeOfExpression} \rangle ')$
26. $\langle \text{ListeOfExpression} \rangle ::= \langle \text{Expression} \rangle \langle \text{ListeOfExpression2} \rangle$
27. $\langle \text{ListeOfExpression2} \rangle ::= ',' \langle \text{Expression} \rangle \langle \text{ListeOfExpression2} \rangle$
28. $\langle \text{ListeOfExpression2} \rangle ::= \epsilon$

29. $\langle \text{LogicalExpression} \rangle ::= \langle \text{LogicalFunction} \rangle$
30. $\langle \text{LogicalExpression} \rangle ::= \text{'true'} \mid \text{'false'}$
31. $\langle \text{LogicalFunction} \rangle ::= \text{'LogicFncIdentifier'} \text{'('} \langle \text{ListeOfGeneral} \rangle \text{'')}$
32. $\langle \text{ListeOfGeneral} \rangle ::= \langle \text{General} \rangle \langle \text{ListeOfGeneral2} \rangle$
33. $\langle \text{ListeOfGeneral2} \rangle ::= \text{'>'} \langle \text{General} \rangle \langle \text{ListeOfGeneral2} \rangle$
34. $\langle \text{ListeOfGeneral2} \rangle ::= \epsilon$

Maintenant que notre grammaire est non récursive à gauche et factorisé à gauche, nous pouvons procéder à la construction de la table d'analyse qui facilitera la construction des procédures de l'analyse descendante récursive, la table d'analyse va aussi nous assurer que la grammaire est bien une **grammaire LL(1)**.

4.3. TABLE D'ANALYSE LL(1)

Commençons d'abord par construire les ensembles **Début** et **Suivant** :

	Non terminaux	Début	Suivant
1	Axiome	=	
2	General	number ; - ; (LogicFncIdentifier; true; false ; ArithmeticFncIdentifier; cellname	# ;) ; ,
3	General2	compareOp ; ϵ	# ;) ; ,
4	Expression	number ; - ; (LogicFncIdentifier; true; false ; ArithmeticFncIdentifier; cellname) ; , ; compareOp ; #
5	Expression2	+ ; - ; ϵ) ; , ; compareOp ; #
6	Term	number ; - ; (LogicFncIdentifier; true; false ; ArithmeticFncIdentifier; cellname	+ ; - ;) ; , ; compareOp ; #
7	Term2	* ; / ; ϵ	+ ; - ;) ; , ; compareOp ; #

8	Factor	number ; - ; (; LogicFncIdentifier; true; false ; ArithmeticFncIdentifier; cellname	* ; / ; + ; - ;) ; , ; compareOp ; #
9	Factor2	$\wedge ; \epsilon$	* ; / ; + ; - ;) ; , ; compareOp ; #
10	Exponent	number ; - ; (; LogicFncIdentifier; true; false ; ArithmeticFncIdentifier; cellname	$\wedge ; * ; / ; + ; - ;)$; , ; compareOp ; #
11	Variable	CellName	$\wedge ; * ; / ; + ; - ;)$; , ; compareOp ; #
12	Variable2	DoublePointOp ; ϵ	$\wedge ; * ; / ; + ; - ;)$; , ; compareOp ; #
13	ArithmeticFunction	ArithmeticFncIdentifier	$\wedge ; * ; / ; + ; - ;)$; , ; compareOp ; #
14	ListeOfExpression	number ; - ; (; LogicFncIdentifier; true; false ; ArithmeticFncIdentifier; cellname)
15	ListeOfExpression2	, ; ϵ)
16	LogicialExpression	LogicFncIdentifier ; true ; false	$\wedge ; * ; / ; + ; - ;)$; , ; compareOp ; #
17	LogicalFunction	LogicFncIdentifier	$\wedge ; * ; / ; + ; - ;)$; , ; compareOp ; #
18	ListeOfGeneral	number ; - ; (; LogicFncIdentifier; true; false ; ArithmeticFncIdentifier; cellname)
19	ListeOfGeneral2	, ; ϵ)

Du tableau ci-dessus, nous pouvons conclure directement que la grammaire est **LL(1)**, nous allons dans ce qui va suivre construire la table d'analyse qui nous sera utile pour la construction des procédures de l'analyse récursive descendante.

	=	compareOp	+	-	*	/	^	()	,	number	cellname	TRUE/FALSE	DoubleOp	ArithmeticFnIdentifier	LogicFunctionIdentifier	#
Axiome	1																
General	1			2				2			2	2	2		2	2	
General2		3							4	4							4
Expression				5				5			5	5	5		5	5	
Expression2		8	6	7					8	8							8
Term				9				9			9	9	9		9	9	
Term2		12	12	12	10	11		12		12							12
Factor				13				13			13	13	13		13	13	
Factor2		15	15	15	15	15	14		15	15							15
Exponent				17				18			16	21	19		20	19	
Variable											22						
Variable2		24	24	24	24	24	24		24	24				23			24
ArithmeticFunction															25		
ListOfExpression				26							26	26	26		26	26	
ListOfExpression2									28	27							
LogicalExpression													30				
LogicalFunction																29	
ListOfGeneral				32							32	32	32		32	32	
ListOfGeneral2									34	33							

FIGURE 3 - TABLE D'ANALYSE LL(1)

Les cellules coloriées en verts impliquent que l'analyseur peut continuer à faire des appels récursives sans erreurs, tandis que ceux rouges implique qu'il y a une erreur de '**symbole non attendu**'.

4.4. ANALYSE DESCENDANTE RECURSIVE

Depuis le tableau, nous pouvons facilement construire les procédures associées à chaque non terminal, nous présenterons ainsi dans ce qui va suivre une procédure générale permettant juste de décrire la structure d'une procédure de descente récursive quelconque.

L'application de cette règle de construction des procédures est utilisée pour tous les non terminaux, veuillez-vous référer au **code source en annexe**, détaillant toutes les procédures utilisées pour effectuer l'analyse syntaxique.

Notons que pour séparer ou alléger le code, nous avons prévu une classe pour chaque analyseur spécifique comme décrit dans le schéma de la **figure 2**, ces classes peuvent toujours être fusionnées dans une seule classe regroupant toutes les procédures.

On suppose que cette ligne de la table d'analyse correspond au non terminal **A** : avec α_{ij} peut désigner un symbole terminal ou non terminal et δ_i désigne un symbole terminal en haut de colonne **i** de la table d'analyse.

δ_1	δ_2	...	δ_i	...
$A ::= \alpha_{11} \alpha_{12} \alpha_{13} \dots$	$A ::= \alpha_{21} \alpha_{22} \alpha_{23} \dots$...	$A ::= \alpha_{i1} \alpha_{i2} \alpha_{i3} \dots$...

Soient $v_1, v_2 \dots v_k$ l'ensemble des symboles n'ayant pas des cases vides correspondants à la ligne associé au non terminal **A**.

Procédure A ()

Début

```

Si (actualSymbol() ==  $v_1$ ) Code( $\alpha_{v_11}$ ) ; Code( $\alpha_{v_12}$ ) ; Code( $\alpha_{v_13}$ ) ...
Sinon Si (actualSymbol() ==  $v_2$ ) Code( $\alpha_{v_21}$ ) ; Code( $\alpha_{v_22}$ ) ; Code( $\alpha_{v_23}$ ) ...
...
Sinon Si (actualSymbol() ==  $v_k$ ) Code( $\alpha_{v_k1}$ ) ; Code( $\alpha_{v_k2}$ ) ; Code( $\alpha_{v_k3}$ ) ...
Sinon RenvoyerErreur() ;

```

Fin

Avec **Code**(α_{ij}) est donné selon la nature de α_{ij} présenté ci-dessus dans le tableau :

Nature de α_{ij}	Code associée
<i>Epsilon</i> : ϵ	{ }
Non terminal	<pre>{ Si (actualSymbol()==α_{ij}) nextSymbol() sinon RenvoyerErreur() }</pre>
terminal	<pre>{ Si (actualSymbol() \in Debut(α_{ij})) α_{ij}() Sinon RenvoyerErreur() }</pre>

Notons que les deux fonctions **ActualSymbol** et **NextSymbol** permettent de faire des requêtes à l'analyseur lexical, la première permet de récupérer le symbole actuel de l'analyse tandis que la seconde fonction permet d'avancer vers la prochaine entité lexicale, En cas d'erreur (*case vide correspondante dans la table d'analyse*) la procédure **RenvoyerErreur()** renvoie l'erreur selon le symbole rencontré, cette customisation permet de mieux clarifier la source d'erreur et donc de rendre notre tableur plus ergonomique, nous allons aborder plus tard dans le rapport la gestion d'erreur dans le tableur.

5. ANALYSEUR SEMANTIQUE

Avant de définir le schéma de traduction de notre évaluateur, nous commencerons d'abord par définir l'environnement de travail, il est à noter que notre évaluateur permet de manipuler des matrices, des booléens ainsi que des réels. L'environnement de travail étant les opérations, les attributs manipulés (synthétisées et héritées) ainsi que les cellules du tableur.

5.1. OPERATIONS BINAIRE ET UNAIRE

Le tableau suivant permet d'étendre le sens des opérations basiques sur les matrices, Matrix désigne un argument qui est une matrice (*de dimension nbLigne*nbCol*) tandis que simple désigne un argument qui n'est pas une matrice (*de dimension 1*1*),

Opération	Argument1	Argument2	Résultat
-	Simple		(-1) * Simple
	Matrix		Donne une matrice composé des opposées de tous les éléments de la matrice en argument
-,+	Simple	Simple	Simple + Simple
	Simple	Matrix	Ajoute Simple à tous les éléments de la matrice en argument.
	Matrix	Simple	Ajoute Simple à tous les éléments de la matrice en argument.
	Matrix	Matrix	Additionne les deux matrices dans le cas où les deux ont strictement la même dimension.
*	Simple	Simple	Simple * Simple
	Simple	Matrix	Multiplie tous les éléments de la matrice en argument avec Simple
	Matrix	Simple	Multiplie tous les éléments de la matrice en argument avec Simple
	Matrix	Matrix	Multiplie les deux matrices dans le cas où ils peuvent être multipliées (nbcol1=nbLigne2)
/	Simple	Simple	Simple/Simple
	Simple	Matrix	Si la matrice en argument est inversible alors le résultat est Simple * inverse(Matrix)
	Matrix	Simple	Divise tous les éléments de la matrice en argument par Simple

	Matrix	Matrix	Si la seconde matrice (en 2ème argument) est inversible et que les deux matrices sont carrées et de mêmes dimensions, alors le résultat est Matrix1 multiplié par Inverse(Matrix2)
^	Simple	Simple	Simple ^ Simple, ^ signifie la puissance qui est associative à gauche.
	Matrix	Simple	Si la matrice est carrée, et que Simple est bien un entier, alors : <ul style="list-style-type: none"> • si Simple == 0 alors la matrice identité est retourné • Si Simple > 0 alors la matrice est multiplié Simple fois par elle-même. • Si Simple < 0 alors le résultat est la matrice inversée multiplié simple fois par elle même
compareOp	Simple /Matrix	Simple /Matrix	Toujours, les premiers éléments de la matrice (en coin haut gauche) sont comparés entre eux, le résultat donné est un booléen donnant la véracité de la comparaison entre les deux éléments.

Toutes autres opérations est considérés comme erreur dite **Opération non autorisée** ou **NotAllowedOperation**.

Notons aussi que les booléens **Vrai** et **faux** se verront transformées en nombre réels **1** et **0** respectivement quand ils sont manipulées dans une expression arithmétique, exemple : $1 + vrai * faux + faux/vrai + vrai = 2$.

5.2. ATTRIBUT UTILISEES

Vu les contraintes de l'évaluateur, permettant non seulement de manipuler des booléens, des réelles, des chaînes de caractères brutes et des matrices, en plus de cela une cellule (ou communément dite une variable de l'interpréteur) peut se référer à une ou plusieurs cellules dans son expression, nous avons ainsi défini un objet permettant de rassembler l'ensemble des valeurs manipulées au niveau du tableur afin qu'il puisse répondre aux critères exigées du tableur.

L'attribut d'évaluation ou l'attribut synthétisé des non terminaux utilisé est de type **CombinedReturnObjet** contenant les attributs suivants :

- **intValue** : La valeur entière manipulée de l'attribut.
- **doubleValue** : La valeur réelle manipulée de l'attribut.
- **booleanValue** : La valeur booléenne manipulée de l'attribut.
- **stringValue** : La chaîne de caractère manipulé de l'attribut.
- **As** : définissant comment l'objet se comporte actuellement c'est-à-dire s'il est un entier, un réel, un booléen ou une chaîne de caractère tout court.
- **errorInExpression** : Véhicule un booléen permettant de savoir si une expression est erronée ou non, cet attribut est beaucoup plus utilisée quand une cellule se réfère à une autre cellule, il permet ainsi de remonter l'erreur à la cellule qui mère.
- **isValue** : Permet de savoir si l'attribut ou l'objet est considéré comme une valeur ou non, par exemple pour le calcul, si on somme un réel avec une cellule tel que la cellule porte une chaîne et non une valeur (non réel et non booléen), **isValue** contiendra Faux pour décrire que l'attribut actuel n'est pas une valeur proprement dite, mais il est à noter que par exemple pour la fonction **si** une chaîne de caractère brut peut être considéré comme une valeur, et donc elle devra être affichée comme elle est dans la cellule correspondante après retour.
- **Matrix** : comme décrit auparavant, dans cet évaluateur il est possible de manipuler des matrices, ainsi si on veut généraliser, tous les objets combinés sont des matrices, ceux simples sont des matrices 1*1, tandis que les matrices ont une dimension i*j, ainsi **intValue**, **doubleValue**, **booleanValue** correspondent toujours à l'élément [1][1] de la matrice, cette dernière est en fait est une matrice de réelles et elle est de Type **Matrix**, classe permettant de définir les opérations de bases entre matrices.

La classe **CombinedReturnObject** (*qu'on va noter d'ici là COR*), définit un ensemble de méthodes permettant de réaliser les opérations de bases entre deux **COR** (addition, soustraction, multiplication, division et puissance), *exemple : **this.add(cor2)** permet de réaliser l'addition entre le CRO actuel et le COR en argument.*

Les attributs hérités par contre peuvent être des **CombinedObjectReturned** dans la plus part des cas mais aussi des **FunctionsAsAttribute** pour la manipulation des fonctions dans l'analyseur **ArithmeticFunctionEvaluator** et **LogicalFunctionEvaluator** (respectivement pour les non terminaux **<ArithmeticFunction>** et **<LogicalFunction>**).

FunctionsAsAttribute (ou **FAA**) est un attribut permettant de regrouper les informations nécessaires pour pouvoir calculer une fonction invoquée dans une expression ainsi **FAA** contiendra :

- **FunctionSymbol** : l'entité lexicale ou symbole représentant le nom de la fonction.
- **FunctionIdentifieur** : définissant réellement la fonction, ainsi elle porte le code réel permettant de calculer la fonction, ainsi que le nombre minimale et maximale de paramètres que la fonction peut manipuler.
- **listOfCombined** : liste de **COR** représentant les paramètres de la fonction.

Ces types d'attributs définis ci-dessus sont ceux utilisées entre les procédures de l'analyse descendante récursive qui permettent de véhiculer les résultats intermédiaires de l'évaluation de l'expression jusqu'à ce que l'évaluateur retourne la valeur décisive d'une expression donnée en entrée, la valeur décisive étant soit le résultat finale où l'erreur.

5.3. VARIABLES OU CELLULES

Pour pouvoir se rapprocher du fonctionnement de MS-Excel, nous devons définir qu'est-ce qu'une variable (ou cellule) pour l'interpréteur.

Une cellule est un Objet de la classe **Cell** permettant de définir tous les attributs et méthodes nécessaires pour pouvoir manipuler de façon convenable une cellule physique du tableur, cette cellule **logique** fournit ainsi les attributs suivants :

- **Name** : le nom de la cellule, rappelons l'expression régulière définissant un nom d'une cellule : «[^][a-z][a-z]?[0123456789]+».
- **Expression** : la chaîne de caractère représentant l'expression contenu de la cellule.
- **Value** : **COR** représentant la valeur de la cellule.
- **empty** : booléen permettant de décrire si la cellule est vide ou non.
- **containRecursiveReference** : permettant de savoir si une cellule s'auto-réfère de façon directe ou indirecte, le traitement de la récursivité de références va être expliqué plus tard.
- **myListners** : contient tous les cellules qui s'y réfèrent à celle-ci, cet attribut permet d'informer tous les cellules se référant à celle-ci en cas de changement de son contenu.
- **isExpression** : booléen Permet de décrire la chaîne contenue dans **Expression**, ainsi si une **Expression** commence par '=' alors elle est considéré comme une expression (**isExpression** est Vrai dans ce cas) sinon c'est une chaîne brut.
- **isExpanded** : booléen reflétant si la matrice contenu dans la cellule actuelle est distribué sur les cellules adjacentes ou non.
- **ExpandedRowsNumber**, **ExpandedColumnNumber** : comme leurs noms l'indiquent, ces deux attributs permettent de décrire le nombre de

colonnes et de lignes adjacents qu'occupe la cellule actuelle, c'est-à-dire dans le cas où la matrice est distribuée sur les cellules adjacentes, ces attributs vont définir le champ qu'occupent les informations de sa matrice.

- **Error** : contient l'erreur décrite avec précision de la cellule où plus précisément l'erreur résultant de l'expression contenu dans la cellule en cas où l'expression est erronée.
- **IsSpecialValue** : booléen utilisé actuellement que pour les cellules contenant des expressions commençant par 'si' permettant de valider si la cellule actuelle est considérée comme valeur spéciale, et donc en cas où cet attribut est vrai les informations calculé de l'expression de la cellule correspondante devront être affichées comme elle le sont même si le **COR value** n'est pas considéré comme une valeur proprement dite, et donc cet attribut ne sert qu'à l'affichage.
- **PrintedValue** : Chaîne de caractère permettant de synthétiser la valeur **Value** de l'expression ainsi que les paramètres contenu dans cette cellule afin de rendre un résultat affiché.

La Classe Cell, permet aussi de fournir des méthodes pour la manipulation d'une cellule, à savoir :

- **Evaluate(Expression)** : qui permet d'évaluer une expression dans une cellule.
- **InformMyListeners()** : qui permet d'informer toutes les cellules qui se réfèrent à cette cellule d'un changement quelconque sur l'information porté par cette cellule.
- **expandMatrix ()** : permet comme son nom l'indique d'étendre la matrice sur les cellules adjacents.
- **compressMatrix()** : permet comme son nom l'indique de compresser la matrice sur une seule cellule et donc d'annuler les valeurs sur les cellules adjacents.
- **FullTreeOfListner()** : Permet d'avoir l'arbre complète des cellules qui se réfèrent à cette cellule, **exemple** : si A écoute B et B écoute C alors l'arborescence complète des écouteurs de C est bien A et B.

5.3.1. Organisation des cellules

Les cellules où les variables dans l'évaluateur sont tous organisées dans un tableau associatives ayant comme *clé* le nom de la cellule, et comme *valeur* la cellule elle-même, ainsi l'évaluateur permet de fournir des méthodes d'accès vers les cellules à savoir : la sélection d'une cellule, l'écriture et la lecture d'information d'une cellule à partir de son nom ...,

On définira dans ce qui va suivre les fonctions importantes :

- **getVariableValue(cellname)** : permet de retourner un **cor** ayant la valeur de la cellule portant comme nom **cellname**.
- **getVariableFamilyValues(cellname1, cellname2)** : permet d'avoir une liste de **combinedReturnObject** portant les valeurs des cellules contenus dans le « carrée » ayant comme coins les cellules possédant les noms cellname1 et cellname2.

Note : Quand nous parlons **cellule** ici, cela ne veut pas dire que c'est la cellule physique réellement affichées dans le tableur mais la représentation logique d'une cellule dans un évaluateur, le traitement graphique étant complètement séparées du noyau de l'interpréteur.

Note 2 : Nous avons défini durant cette section du rapport l'environnement clé de l'analyse sémantique faite par l'interpréteur, cela est fait pour mieux comprendre le schéma sémantique qui va être présentées ci-dessous

5.4. SCHEMA DE TRADUCTION

5.4.1. Schéma de traduction sur la grammaire naturelles

Notons, que dans le schéma de traduction qu'on va présenter ci-dessous nous allons utiliser les opérations entre **COR**, les différentes opérations ainsi seront interprétées comme il était défini dans le tableau des opérations du **titre 5.1**.

Production	Résultat
<Axiome> ::= '=' <General> '#'	Axiome.cor = general.cor
<General> ::= <Expression>₁ 'CompareOp' <Expression>₂	general.cor = compare(expression₁.cor, expression₂.cor)
<General> ::= <Expression>	general.cor = expression.cor
<Expression>₁ ::= <Expression>₂ '+' <Term>	expression₁.cor = expression₂.cor + terme.cor
<Expression> ::= <Expression> '-' <Term>	expression₁.cor = expression₂.cor - terme.cor
<Expression> ::= <Term>	expression.cor = terme.cor
<Term>₁ ::= <Term>₂ '*' <Factor>	term₁.cor = term₂.cor * factor.cor
<Term>₁ ::= <Term>₂ '/' <Factor>	term₁.cor = term₂.cor / factor.cor
<Term> ::= <Factor>	term.cor = factor.cor
<Factor>₁ ::= <Factor>₂ '^' <Exponent>	factor₁.cor = factor₂.cor ^ exponent.cor
<Factor> ::= <Exponent>	factor.cor = exponent.cor
<Exponent> ::= 'number'	exponent.cor = Value(number)
<Exponent>₁ ::= '-' <Exponent>₂	exponent₁.cor = - exponent₂.cor
<Exponent> ::= '(' <General> ')'	exponent.cor = general.cor
<Exponent> ::= <LogicalExpression>	exponent.cor = logicalExpression.cor
<Exponent> ::= <ArithmeticFunction>	exponent.cor = arithmeticFunction.cor
<Exponent> ::= 'CellName'	exponent.cor = getVariableValue(CellName)
<Exponent> ::= 'CellName'₁ ':' 'CellName'₂	exponent.cor = getVariablefamilyValues (CellName1, CellName2)

<Exponent> ::= 'CellName' ₁ ':' 'CellName' ₂	exponent.cor = getVariablefamilyValues (CellName1, CellName2).size().
<Exponent> ::= 'CellName' ₁ ':' '+' 'CellName' ₂	exponent.cor = sum(getVariablefamilyValues (CellName1, CellName2)).
<Exponent> ::= 'CellName' ₁ ':' '*' 'CellName' ₂	exponent.cor = multiply(getVariablefamilyValues (CellName1, CellName2)).
<Exponent> ::= 'CellName' ₁ ':' '^' 'CellName' ₂	exponent.cor = sumsquare(getVariablefamilyValues (CellName1, CellName2)).
<ArithmeticFunction> ::= 'ArithmeticFncIdentifier' ('<ListOfExpression>')	<ul style="list-style-type: none"> • ListOfExpression.faa_h.functionIdentifier = getFunctionIdentifier (ArithmeticFncIdentifier) • ListOfExpression.faa_h.listofcombined = newList(). • ArithmeticFunction.cor = ListOfExpression.faa_s.functionIdentifier .execute(ListOfExpression.faa_s.listofcombined)
<ListOfExpression> ₁ ::= <ListOfExpression> ₂ ',' <Expression>	ListOfExpression₁.faa = ListOfExpression₂.faa_s.add(Expression.cor)
< ListOfExpression > ::= <Expression>	ListOfExpression.faa.add(Expression.cor)
<LogicalExpression> ::= <LogicalFunction>	LogicalExpression.cor = LogicalFunction.cor
<LogicalExpression> ::= 'true'	LogicalExpression.cor = boolean(true)
<LogicalExpression> ::= 'false'	LogicalExpression.cor = boolean(false)
<LogicalFunction> ::= 'LogicFncIdentifier' ('<ListOfGeneral>')	<ul style="list-style-type: none"> • ListOfGeneral.faa_h.functionIdentifier = getFunctionIdentifier (LogicFncIdentifier) • ListOfGeneral.faa_h.listofcombined = newList(). • LogicalFunction.cor = ListOfExpression.faa_s.functionIdentifier .execute(ListOfGeneral.faa_s.listofcombined) ⁽¹⁾

$\langle \text{ListOfGeneral} \rangle_1 ::= \langle \text{ListeOfGeneral} \rangle_2$,', <General>	ListOfGeneral ₁ .faa = ListeOfGeneral ₂ .faas.add(General.cor)
$\langle \text{ListOfGeneral} \rangle ::= \langle \text{General} \rangle$	ListOfGeneral.faa.add(General.cor)

⁽¹⁾ **faa_h** représente l'attribut faa hérité tandis que **faas** représente l'attribut faa synthétisé

Note 1 : l'attribut synthétisé **cor** est représenté comme étant un objet **CombinedReturnObject** et donc il définit l'ensemble de ses attributs ainsi que tout le set des méthodes de manipulations de données notamment ceux concernant les matrices.

Note 2 : L'attribut **faa** porté par les non terminaux **ListOfExpression** et **ListOfGeneral** est un objet FunctionAsAttribute comme il a été décrit auparavant durant l'énoncé des attributs utilisés dans l'évaluateur. Les fonctions **add** et **execute** permettent de :

- **add(combinedReturnObject)** : permet d'étendre la liste du **faa** par un nouvel objet **cor**.
- **execute (liste)** : permet d'exécuter la fonction ayant comme identifiant functionIdentifier sur la **liste des cor** donnée comme paramètre.

Note 3 : Quand on dit **cor = cor1 + cor2**, réellement dans le code c'est **cor = cor1.add(cor2)**, mais nous avons préféré de les représenter sous forme d'opération standard pour mieux abstraire le schéma de traduction du code fonctionnel.

5.4.2. Schéma de traduction sur la grammaire LL(1)

Maintenant que le schéma de traduction est bien clair pour la grammaire naturelle, la transformation du schéma de traduction pour la grammaire LL(1) est facile, il suffit de rajouter des attributs héritées afin d'assurer la disponibilité des valeurs au moment où l'opération doit être exécuté. Le tableau suivant montre le nouveau schéma de traduction.

Production	Résultat
<Axiome> ::= '=' <General> '#'	Axiome.cor = general.cor
<General> ::= <Expression> <General2>	General2.cor_h = expression.cor general.cor = General2.cor_s
<General2> ::= CompareOp' <Expression>	General2.cor_s = compare(General2.cor_h, Expression.cor)
<General2> ::= ε	General2.cor_s = General2.cor_h
<Expression> ::= <Term> <Expression2>	Expression2.cor_h = term.cor_s Expression.cor_s = Expression2.cor_s
<Expression2>₁ ::= '+' <Term> <Expression2>₂	Expression2₂.cor_h = Expression2₁.cor_h + term.cor_s Expression2₁.cor_s = Expression2₂.cor_s
<Expression2> ::= '-' <Term> <Expression2>	Expression2₂.cor_h = Expression2₁.cor_h - term.cor_s Expression2₁.cor_s = Expression2₂.cor_s
<Expression2> ::= ε	Expression2.cor_s = Expression2.cor_h
<Term> ::= <Factor> <Term2>	Term2.cor_h = factor.cor_s term.cor_s = term2.cor_s
<Term2>₁ ::= '*' <Factor> <Term2>₂	Term2₂.cor_h = Term2₁.cor_h * factor.cor_s Term2₁.cor_s = Term2₂.cor_s
<Term2>₁ ::= '/' <Factor> <Term2>₂	Term2₂.cor_h = Term2₁.cor_h / factor.cor_s Term2₁.cor_s = Term2₂.cor_s
<Term2> ::= ε	Term2.cor_s = Term2.cor_h

<Factor> ::= <Exponent> <Factor2>	Factor2.cor_h = Exponent.cor_s Factor.cor_s = Factor2.cor_s
<Factor2>₁ ::= '^' <Exponent> <Factor2>₂	Factor2₂.cor_h = Factor2₁.cor_h * Exponent.cor_s Factor2₁.cor_s = Factor2₂.cor_s
<Factor2> ::= ε	Factor2.cor_s = Factor2.cor_h
<Factor>₁ ::= <Factor>₂ '^' <Exponent>	factor₁.cor = factor₂.cor ^ exponent.cor
<Factor> ::= <Exponent>	factor.cor = exponent.cor
<Exponent> ::= 'number'	exponent.cor = Value(number)
<Exponent>₁ ::= '-' <Exponent>₂	exponent₁.cor = - exponent₂.cor
<Exponent> ::= '(' <General> ')'	exponent.cor = general.cor
<Exponent> ::= <LogicalExpression>	exponent.cor = logicalExpression.cor
<Exponent> ::= <ArithmeticFunction>	exponent.cor = arithmeticFunction.cor
<Exponent> ::= <Variable>	exponent.cor = variable.cor
<Variable> ::= 'CellName' <Variable2>	Variable2.namevar_{1h} = CellName Variable.cor = Variable2.cor
<Variable2> ::= ε	Variable2.cor = getVariableValue(Variable2.namevar_{1h})
<Variable2> ::= ':' 'CellName'	Variable2.cor = getVariablefamilyValues (Variable2.namevar_{1h}, CellName')
<Variable2> ::= ':' '_' 'CellName'	Variable2.cor = getVariablefamilyValues (Variable2.namevar_{1h}, CellName').size()
<Variable2> ::= ':' '+' 'CellName'	Variable2.cor = sum(getVariablefamilyValues (Variable2.namevar_{1h}, CellName'))
<Variable2> ::= ':' '*' 'CellName'	Variable2.cor = multiply (getVariablefamilyValues (Variable2.namevar_{1h}, CellName'))
<Variable2> ::= ':' '^' 'CellName'	Variable2.cor = sumsquare (getVariablefamilyValues (Variable2.namevar_{1h}, CellName'))

<ArithmeticFunction> ::= 'ArithmeticFncIdentifier' ('<ListOfExpression>')	<ul style="list-style-type: none"> • ListOfExpression.faa_h.functionIdentifier = getFunctionIdentifier (ArithmeticFncIdentifier) • ListOfExpression.faa_h.listofcombined = newList(). • ArithmeticFunction.cor = ListOfExpression.faa_s.functionIdentifier.execute(ListOfExpression.faa_s.listofcombined)
< ListOfExpression > ::= <Expression>< ListOfExpression2 >	<ul style="list-style-type: none"> • ListOfExpression2.faa_h = ListOfExpression.faa_h.add(expression.cor) • ListOfExpression.faa_s = ListOfExpression2.faa_s
< ListOfExpression2 >₁ ::= ', '<Expression>< ListOfExpression2 >₂	<ul style="list-style-type: none"> • ListOfExpression22.faa_h = ListOfExpression21.faa_h.add(expression.cor) • ListOfExpression21.faa_s = ListOfExpression22.faa_s
< ListOfExpression2 > ::= ε	ListOfExpression2.faa_s = ListOfExpression2.faa_h
<LogicalExpression> ::= <LogicalFunction>	LogicalExpression.cor = LogicalFunction.cor
<LogicalExpression> ::= 'true'	LogicalExpression.cor = boolean(true)
<LogicalExpression> ::= 'false'	LogicalExpression.cor = boolean(false)
<LogicalFunction> ::= 'LogicFncIdentifier' ('<ListOfGeneral>')	<ul style="list-style-type: none"> • ListOfGeneral.faa_h.functionIdentifier = getFunctionIdentifier (LogicFncIdentifier) • ListOfGeneral.faa_h.listofcombined = newList(). • LogicalFunction.cor = ListOfExpression.faa_s.functionIdentifier.execute(ListOfGeneral.faa_s.listofcombined)
< ListOfGeneral > ::= <General>< ListOfGeneral2 >	<ul style="list-style-type: none"> • ListOfGeneral2.faa_h = ListOfGeneral.faa_h.add(General.cor) • ListOfGeneral.faa_s = ListOfGeneral2.faa_s

$\langle \text{ListeOfGeneral2} \rangle_1 ::= ', \langle \text{General} \rangle \langle \text{ListeOfGeneral2} \rangle_2$	<ul style="list-style-type: none"> • $\text{ListeOfGeneral2}_2.\text{faah} = \text{ListeOfGeneral2}_1.\text{faah}.\text{add}(\text{general.cor})$ • $\text{ListeOfGeneral2}_1.\text{faas} = \text{ListeOfGeneral2}_2.\text{faas}$
$\langle \text{ListeOfGeneral2} \rangle ::= \epsilon$	$\text{ListeOfGeneral2}.\text{faas} = \text{ListeOfGeneral2}.\text{faah}$
$\langle \text{ListOfGeneral} \rangle_1 ::= \langle \text{ListeOfGeneral} \rangle_2 ', \langle \text{General} \rangle$	$\text{ListOfGeneral}_1.\text{faa} = \text{ListeOfGeneral}_2.\text{faas}.\text{add}(\text{General.cor})$
$\langle \text{ListOfGeneral} \rangle ::= \langle \text{General} \rangle$	$\text{ListOfGeneral}.\text{faa}.\text{add}(\text{General.cor})$

5.5. AMELIORATION DES PROCEDURE DE DESCENTE RECURSIVE

Le schéma de traduction de la grammaire LL(1) décrit dans la table ci-dessous nous permet facilement d'évoluer nos procédures de descente récursive utilisées pour l'analyse syntaxique, en fonctions permettant d'effectuer l'analyse sémantique d'une expression donnée et donc l'évaluation.

Pour cela on insère les parcelles de codes décrites pour chaque production dans les procédures associées à leurs **Membres Droits de Productions (MDP)**.

Les fonctions retourneront ainsi leurs **attributs synthétisés**, et requêtent comme paramètres leurs **attributs hérités**, tandis que la passation d'arguments et la synthétisation se fait par l'effectuation d'un appel d'un **MDP** d'une production appelant un non terminal faisant partie du **MGP** de cette même production.

Dans ce qui va suivre nous présenterons un exemple de l'amélioration d'une procédure d'analyse descendante par l'insertion des opérations sémantiques dans leurs emplacement spécifiques, pour voir l'ensemble des fonctions de descente récursives de l'évaluation veuillez-vous référer au code source en annexe.

Prenons l'exemple de la fonction associé au non terminal **<terminal2>** qui permet d'analyser la multiplication, la première procédure ci-dessous concerne celle de l'analyse syntaxique :

Terme2()**Debut**

Si (actualSymbol() == OPMULT ou actualSymbol() == OPDIVIDE ^(*)) alors

Dsi

nextSymbol()

si (ActualSymbol() ∈ Debut(ExpressionArithmetic))

Dsi

Si (actualSymbol() ∈ Debut(Terme2) ou actualSymbol() ∈ Suivant(Terme2))

terme2()

Sinon

RenvoyerErreur()

Fsi

Sinon **RenvoyerErreur()**

Fsi

Sinon si (actualSymbol() ∈ Suivant(Terme2)) ; *//rien faire*

Sinon **RenvoyerErreur()**

Fin

(*) **OPMULT** et **OPDIVIDE** c'est des éléments de l'énumération permettant de représenter les opérations * et / respectivement.

La procédure suivante est **terme2()** évolué, permettant ainsi de faire l'analyse sémantique ou autrement d'évaluer la multiplication et la division au sein d'une expression, il est à noter que pour le faire on ajoute comme paramètre son attribut hérité qui n'est en fait qu'un **COR**, cette fonction permet de synthétiser la valeur d'une multiplication ou d'une division qui est aussi un **COR** comme montré dans le schéma de traduction en le retournant.

```

CombinedReturnObject terme2(CombinedReturnObject factor)
Debut
CombinedReturnObject ret = factor
Si (actualSymbol() == OPMULT ou actualSymbol() == OPDIVIDE) alors
Dsi
    Symbol op = actualSymbol()
    nextSymbol()
    si (ActualSymbol() ∈ Debut(ExpressionArithmetic))
    Dsi
        CombinedReturnObject factor2 = factor()
        Si (op == OPMULT) factor.mult(factor2)
        Sinon factor.divide(factor2)
        Si (actualSymbol() ∈ Debut(Terme2) ou actualSymbol() ∈ Suivant(Terme2))
            Ret = terme2(factor)
        Sinon
            RenvoyerErreur()
        Fsi
    Sinon RenvoyerErreur()
Fsi
Sinon si (actualSymbol() ∈ Suivant(Terme2)) ret = factor
Sinon RenvoyerErreur()
Retourner ret
Fin

```

Note : toutes les procédures d'analyse syntaxique sont améliorées de cette manière après l'insertion des routines sémantiques dans leurs emplacements spécifiques.

6. GESTION D'ERREUR

Pour maîtriser toutes les erreurs, nous avons complété les cellules vides de la table d'analyse par l'envoi des erreurs correspondante, pour pouvoir le réaliser nous avons créé une classe d'**Exception** permettant de regrouper l'ensemble des erreurs possibles, une erreur dans l'évaluateur étant un **message**, le **symbole causant erreur**, et son indice dans l'**expression source**.

Ainsi au niveau des procédures on lance les exceptions dans leurs emplacements appropriés comme décrit dans la table d'analyse et la méthode de créations de procédure d'analyse descendante récursive du titre 4.4.

Maintenant que les des erreurs concernant le fait de trouver **un symbole non attendue** sont réglées. Nous aborderons l'**erreur de récursivité de référencement**. Cette erreur concerne le fait qu'une cellule se réfère soit à elle-même, soit à une chaîne de référencement entre cellule qui contienne un **auto-référencement**.

Pour pouvoir traiter cette erreur, comme dit, une cellule contient une table d'écouteurs, les écouteurs étant les cellules qui se réfèrent à cette cellule, à chaque fois qu'une cellule tente de récupérer la valeur d'une autre cellule, elle s'insère comme étant un écouteur de cette dernière, lors de cette insertion, on vérifie d'abord si la cellule référent n'est pas en fait la même cellule référent, si c'est le cas alors on lance une exception pour dire qu'il y'a une récursivité directe ou un auto-référencement directe.

Sinon pour ce qui concerne la récursivité indirecte, on attend d'abord la fin de l'évaluation de l'expression, puis au moment où la cellule tente d'informer tous ces écouteurs (les écouteurs au complet ne sont pas seulement qui se trouvent dans sa table d'écouteurs, mais aussi se trouvant dans les tables respectives des cellules écouteurs et ainsi de suite), ainsi on trace d'abord l'arbre complet des écouteurs de la cellule, puis à ce moment si l'arbre contient deux fois la même cellule, cela implique qu'il y'a une récursivité de référencement, c'est à ce moment qu'une erreur est signalé pour dire que cette cellule fait partie d'une chaîne de référencement récursive.

Reste maintenant les erreurs sémantiques concernant les opérations et les fonctions, étant donné que notre évaluateur évalue aussi les opérations entre matrices, de nombreuses opérations peuvent causer des erreurs dite **erreurs d'opération non autorisées**, ces erreurs proviennent des opérations binaires concernant deux matrices ne respectant pas les conditions de dimensions (exemple : pour l'addition entre matrices, les deux matrices doivent avoir strictement le même nombre de lignes et le même nombre de colonnes), ou le fait d'essayer d'inverser une

matrice non inversible (exemple : une matrice étant non carré ne peut pas être inversé).

L'autre erreur sémantique concerne le nombre d'arguments, chaque fonction définie pour l'évaluateur a en fait un nombre prédéfinie d'arguments minimale et maximale, ainsi durant l'évaluation dans la procédure correspondante (celle concernant *arithmeticFunctionEvaluator* et *logicFunctionEvaluator*), on compte le nombre d'arguments passées (il est à noter qu'on peut passer un champ de variables par l'opération ':' considéré comme **nbligne*nbcolonne** arguments pour la fonction), si ça dépasse au cours des appels pour l'ajout des arguments (dans la procédure concernant le non terminal **<listOfExpression>**, on signale une erreur '**nombre invalide d'arguments**', sinon on teste au niveau de l'exécution de la fonction pour vérifier que le nombre d'arguments en paramètre respecte bien la limite prédéfinie.

La dernière erreur provoquant une exception dans notre interpréteur est celle concernant le fait d'introduire une expression ne commençant pas par le symbole '=', mais ceci n'implique pas un affichage d'erreur, le fait que le tableur peut toujours contenir des chaînes de caractères qui ne sont pas des expressions.

Il est à noter, qu'une **erreur de valeur** peut survenir, en cas où une cellule tente d'exécuter une opération arithmétique quelconque sur une cellule référencée ne contenant pas une valeur proprement dite (par exemple: =a2+3 avec a2 une cellule contenant 'chaîne').

7. INTERFACE DE L'EVALUATEUR

Maintenant que l'interpréteur est mis en place, et que les cellules logiques ont bien été définies dans le noyau fonctionnel il est facile, de lier chaque **cellule physique** de l'interface avec une cellule logique, puis à chaque changement au niveau de la cellule logique, la cellule physique est informée et l'affichage est effectué. Sinon l'interface d'autres composants qui permettent de faciliter l'utilisation de l'application en lui offrant les raccourcis, la possibilité de distribuer les valeurs d'une cellule-matrice, le recensement des fonctions existantes, le copier-coller d'une expression d'une cellule ...

Le schéma suivant permet de décrire l'échange entre le noyau fonctionnel, et le tableur contenant les cellules physiques de l'évaluateur.

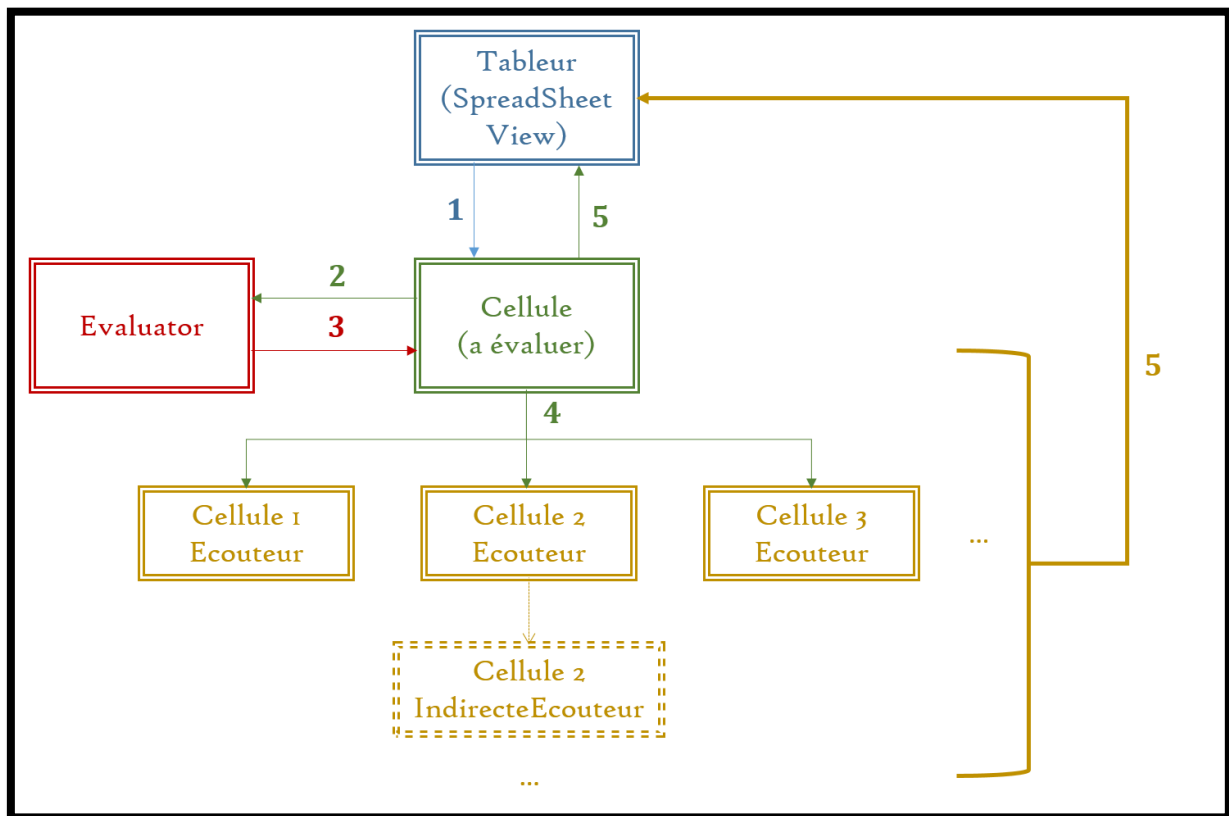


FIGURE 4 - SYNTHETISATION DU COMPORTEMENT DE L'EVALUATEUR

Décrivons maintenant les étapes définies dans la figure ci-dessus :

- **Etape 1** : l'utilisateur choisit une cellule, dans le tableur physique, à la sélection, la cellule logique est sélectionnée et on passe l'expression saisie (exécution de la fonction `cellule.evaluate(Expression)`).
- **Etape 2,3** : La cellule évalue l'expression, par l'intermédiaire des analyseurs puis le résultat est retourné à la cellule appelante.
- **Etape 4** : La cellule informe tous ses écouteurs pour leur dialoguer la modification effectuée.
- **Etape 5** : une mise à jour de l'affichage est effectuée au niveau des cellules physiques (le tableur).

III. TESTS ET FONCTIONNALITES

1. FONCTIONNALITES DE L'APPLICATION

L'application MiniCompilTableur développé à l'issue de ce petit projet offre les fonctionnalités suivantes :

- **Evaluation des expressions** : objectif principal du tableur, permet d'évaluer des expressions arithmétiques ou logiques.
- **Manipulation des matrices** : permet d'exécuter des opérations sur les matrices et de les visualiser (**une cellule** peut contenir une matrice si elle se réfère à un champ de cellule, on peut ainsi étendre la matrices sur les cellules adjacents pour pouvoir la visualiser avec la fonction '**expand matrix**').
- **Gestion d'erreurs avancées** : en cas d'erreur au niveau lexicale, syntaxique ou sémantique, la cellule portant l'expression générant erreur se colorie et permet de donner des détails sur l'erreur généré, l'interpréteur détecte aussi les erreurs sur les nombres, types d'arguments de fonctions, ainsi que les erreurs de références cycliques.
- **Conservation de l'historique** : l'application fournit les fonctions **Undo** et **Redo** pour permettre le retour en arrière, l'historique peut garder jusqu'à 100 pas d'évaluation, ou chaque pas est considéré si l'on a un changement de valeur/expression d'une ou plusieurs cellules du tableur.
- **Manipulation de fonctions aléatoires** : les cellules peuvent munir des expressions utilisant des fonctions retournant des valeurs aléatoires, l'application ainsi fourni l'option '**Refresh**' qui permet d'actualiser les valeurs contenus dans des cellules dites aléatoire, la fonctionnalité **Refresh** ainsi offre la possibilité de faire des expériences de probabilités sur le tableur en générant des échantillons aléatoires.
- **Manipulation des fichiers** : l'application offre la possibilité de **créer**, **sauvegarder** et **charger** des feuille de calcules, les fichiers sont stockées sous le format '**mcx**'.

La capture suivante donne une idée sur l'interface de l'application : ou l'ensemble des tests sont faits sur des cellules aléatoires, ainsi à chaque rafraichissement l'ensemble des valeurs est recalculé.

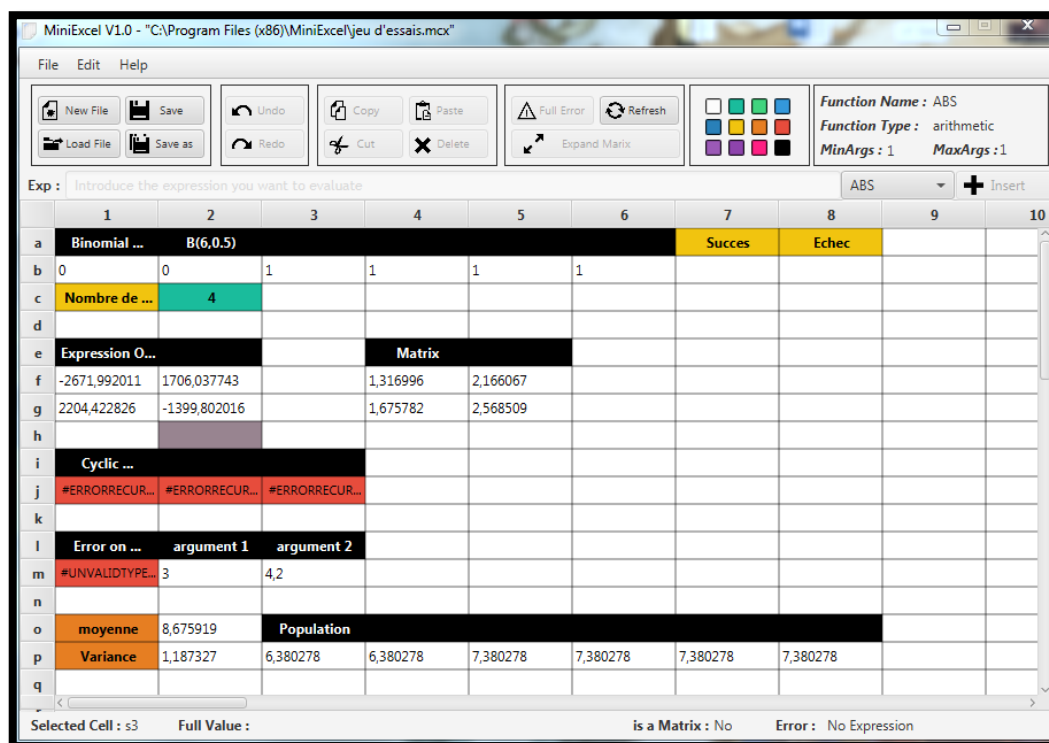


FIGURE 5 - VISION GLOBALE DE L'INTERFACE DU TABLEUR

2. TESTS DE FONCTIONNEMENT

2.1. TEST D'ÉVALUATION D'UNE EXPRESSION ARITHMETIQUE

Exp : $=(\max(11^2+3, 3^4+6)+3)--1/3*2$			
	1	2	3
a	127,666667		
b			

2.2. TEST D'ÉVALUATION D'UNE OPERATION DOUBLE POINT

Exp : $=a1:^a5/a1:_a5 - (a1:+a5/a1:_a5)^2$					
	1	2	3	4	5
a	1	3	4	2	5
b	moyenne	3			
c	Variance	2			
d					
e					

2.3. TEST D'ÉVALUATION D'UNE EXPRESSION LOGIQUE

Exp : $=\text{si}(\text{rand}()>1/2,1,0)$						
	1	2	3	4	5	6
a	0	0	1	1	0	0
b	Nombre de succes	2				

2.4. TEST DE MANIPULATION DES MATRICES

Exp : $=1/b5:d7$							ABS
	1	2	3	4	5	6	7
a	Matrice Inverse				Matrice		
b	-0,5	0,0625	0,5625		1	6	2
c	0,25	-0,09375	0,15625		4	5	3
d	0,25	0,15625	-0,59375		2	7	1


2.5. TEST DE LA GESTION D'ERREURS

2.5.1. TEST D'ERREUR DE REFERENCEMENT CYCLIQUE

Exp :	=a1			
	1	2	3	4
a	#ERRORRECURSIVE!	#ERRORRECURSIVE!	#ERRORRECURSIVE!	
b				

2.5.2. TEST D'ERREUR DE TYPE D'ARGUMENTS

Exp :	=pgcd(2.3,4)		
	1	2	3
a	#UNVALIDTYPEOFARGUMENTS!		
b			
c			
d			
e			
f			
g			
h			
i			

Error detail
 All Argument of the function PGCD must be integers !

Note : un plan de test approprié et concis est prévu pour le jour de la démonstration pour un balayage des différents fonctionnalités.

CONCLUSION

Le Projet réalisé, a l'issue de ce module a été enrichissant sur plusieurs plans, non seulement il nous a permis d'approfondir nos connaissances vis-à-vis des notions de compilations, mais en plus il nous a offert la possibilité de voir que les notions de compilations ne sont pas restreints sur la réalisation des compilateurs, mais ont une utilisation plus large dans plusieurs domaines.