

UE INF404 - Projet Logiciel

Calculatrice : étape 4 (fin !)

Evaluation d'une expression arithmétique générale

L2 Informatique

Année 2023 - 2024

Rappel des précédents épisodes (1)

Ecrire un interpréteur d'expressions arithmétiques

Version 1 = “Expressions Arithmétiques Simples” (EAS)

- les opérandes sont des entiers
- opérateurs arithmétiques usuels (+, -, *, /)
- pas de priorités (évaluation de gauche à droite)

Version 2 = “Expressions Arithmétiques Entièrement Parenthésées” (EAEP)

- les opérandes sont des entiers
- parenthèses obligatoires autour de chaque opération
- évaluation selon le parenthésage ...

Rappel des précédents épisodes (1)

Ecrire un interpréteur d'expressions arithmétiques

Version 1 = “Expressions Arithmétiques Simples” (EAS)

- les opérandes sont des entiers
- opérateurs arithmétiques usuels (+, -, *, /)
- pas de priorités (évaluation de gauche à droite)

Version 2 = “Expressions Arithmétiques Entièrement Parenthésées” (EAEP)

- les opérandes sont des entiers
- parenthèses obligatoires autour de chaque opération
- évaluation selon le parenthésage ...

Rappel des précédents épisodes (1)

Ecrire un interpréteur d'expressions arithmétiques

Version 1 = “Expressions Arithmétiques Simples” (EAS)

- les opérandes sont des entiers
- opérateurs arithmétiques usuels (+, -, *, /)
- pas de priorités (évaluation de gauche à droite)

Version 2 = “Expressions Arithmétiques Entièrement Parenthésées” (EAEP)

- les opérandes sont des entiers
- parenthèses obligatoires autour de chaque opération
- évaluation selon le parenthésage ...

Rappel des précédents épisodes (2)

“Expressions Arithmétiques Générales” (EAG)

- parenthésage lorsque nécessaire
- évaluation selon les règles de priorités usuelles ...

Programmation ?

- syntaxe spécifiée par une grammaire “structurée”
- analyse syntaxique =
procédures **récurives**, guidée par la grammaire (c.f. TP3)

La suite = **évaluation** d'une EAG ?

Rappel des précédents épisodes (2)

“Expressions Arithmétiques Générales” (EAG)

- parenthésage lorsque nécessaire
- évaluation selon les règles de priorités usuelles ...

Programmation ?

- syntaxe spécifiée par une grammaire “structurée”
- analyse syntaxique =
procédures **récurives**, guidée par la grammaire (c.f. TP3)

La suite = **évaluation** d'une EAG ?

Grammaire (structurée) des EAG

expression = séquence de termes séparés par PLUS, MOINS

terme = séquence de facteurs séparés par MUL, DIV

facteur = ENTIER ou expressions parenthésées ...

<i>eag</i>	→	<i>seq_terme</i>
<i>seq_terme</i>	→	<i>terme suite_seq_terme</i>
<i>suite_seq_terme</i>	→	<i>op1 terme suite_seq_terme</i>
<i>suite_seq_terme</i>	→	ϵ
<i>terme</i>	→	<i>seq_facteur</i>
<i>seq_facteur</i>	→	<i>facteur suite_seq_facteur</i>
<i>suite_seq_facteur</i>	→	<i>op2 facteur suite_seq_facteur</i>
<i>suite_seq_facteur</i>	→	ϵ
<i>facteur</i>	→	<i>ENTIER</i>
<i>facteur</i>	→	PARO <i>eag</i> PARF
<i>op1</i>	→	PLUS
<i>op1</i>	→	MOINS
<i>op2</i>	→	MUL

Arbres de dérivation

Soit $G = (V_t, V_n, Z, P)$ une grammaire et w un mot de $L(G)$.

On appelle arbre de dérivation (ou arbre syntaxique) de w dans G tout arbre n -aire A dont les noeuds sont des éléments de $(V_t \cup V_n)$ et tel que :

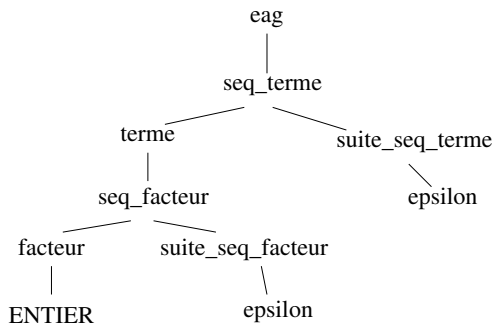
- la racine de A est Z (l'axiome de G) ;
- les feuilles de A sont des éléments de $V_t \cup \{\varepsilon\}$ et la séquence gauche-droite des feuilles de A est égale à w ;
- les noeuds non feuilles de A sont des éléments de V_n et si un noeud non feuille X a pour fils u_1, u_2, \dots, u_n dans A alors la règle $X \rightarrow u_1.u_2 \dots u_n$ doit appartenir à P .

A tout mot de $L(G)$ on peut associer un arbre de dérivation.

Si cet arbre est **unique** alors la grammaire est dite **non ambiguë**.

Exemple d'arbre de dérivation (1)

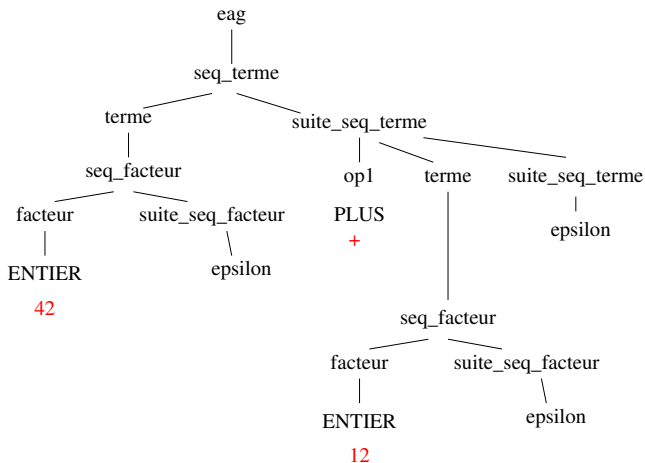
42



42

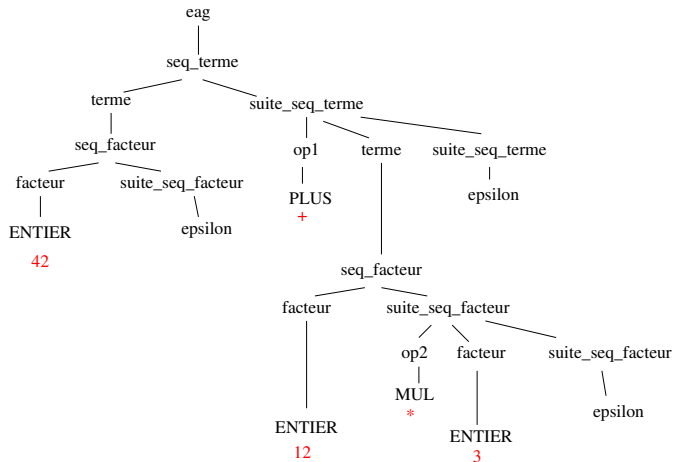
Exemple d'arbre de dérivation (2)

42 + 12



Exemple d'arbre de dérivation (3)

42 + 12 * 3



Arbre abstrait (Abstract Syntax Tree, AsT)

arbre de dérivation \sim exécution de l'analyseur syntaxique
(arbre des appels des procédures récursives “extraites” de la grammaire)

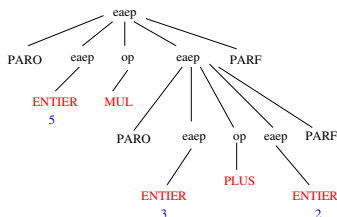
Arbre abstrait = Une “simplification” de l'arbre de dérivation

→ permet de représenter les “informations utiles”

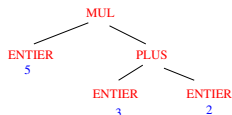
(ex : structure du texte lu, éléments importants)

⇒ **représentation intermédiaire** utilisée en sortie de l'analyseur

Exemple (sur les EAEP) : $(5 * (3 + 2))$



Arbre de Dérivation



Arbre Abstrait

Arbre abstrait (Abstract Syntax Tree, AsT)

arbre de dérivation \sim exécution de l'analyseur syntaxique
(arbre des appels des procédures récursives “extraites” de la grammaire)

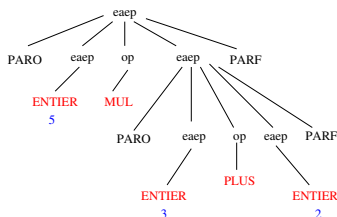
Arbre abstrait = Une “simplification” de l'arbre de dérivation

→ permet de représenter les “informations utiles”

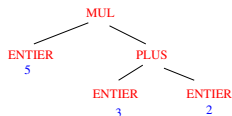
(ex : structure du texte lu, éléments importants)

⇒ **représentation intermédiaire** utilisée en sortie de l'analyseur

Exemple (sur les EAEP) : $(5 * (3 + 2))$



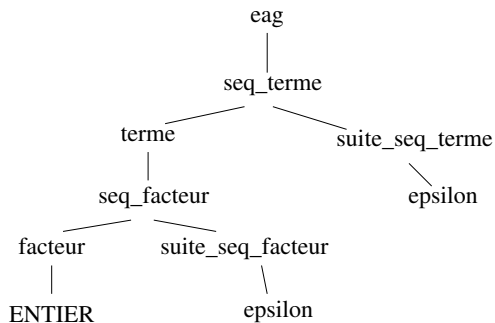
Arbre de Dérivation



Arbre Abstrait

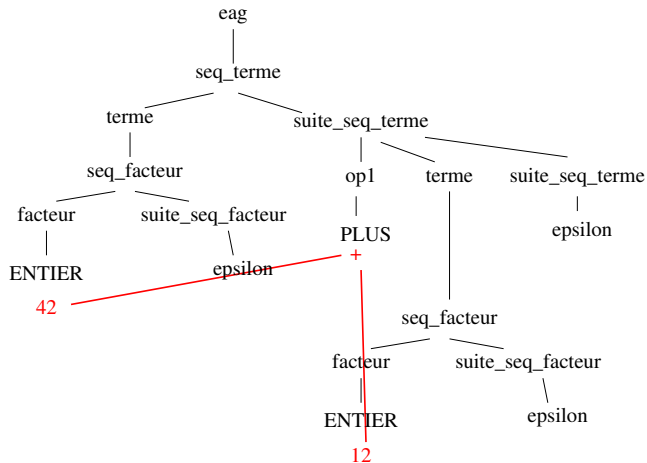
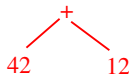
Exemple d'arbre abstrait : 42

42

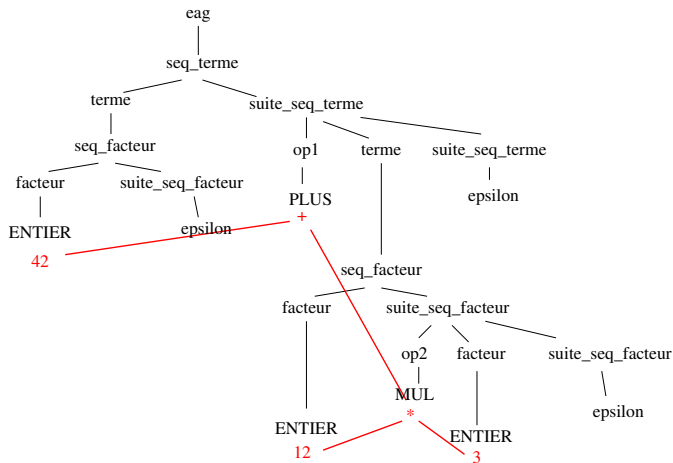
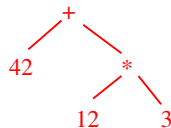


42

Exemple d'arbre abstrait : 42 + 12

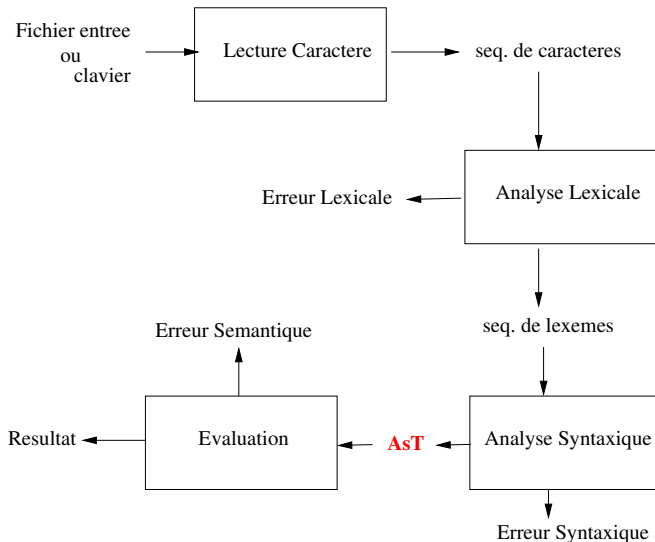


Exemple d'arbre abstrait : $42 + 12 * 3$



Structure de la calculatrice

Quatre composants/modules principaux ...



Implémentation de l'AsT

Un type Ast (type_ast.h)

```
typedef enum {OPERATION, VALEUR} TypeAst ;
typedef enum {N_PLUS, N_MUL, N_MOINS} TypeOperateur ;

typedef struct noeud {
    TypeAst nature ;
    TypeOperateur operateur ;
    struct noeud *gauche, *droite ;
    int valeur ;
} NoeudAst ;

typedef NoeudAst *Ast ;
```

Des primitives de construction (ast_construction.h)

```
Ast creer_operation(TypeOperateur opr , Ast op_gauche , Ast op_droit);
// renvoie un Ast (op_gauche, opr, op_droit) de nature OPERATION

Ast creer_valeur(int val) ;
// renvoie un Ast "feuille", de nature VALEUR et de valeur val
```

Implémentation de l'AsT

Un type Ast (type_ast.h)

```
typedef enum {OPERATION, VALEUR} TypeAst ;
typedef enum {N_PLUS, N_MUL, N_MOINS} TypeOperateur ;

typedef struct noeud {
    TypeAst nature ;
    TypeOperateur operateur ;
    struct noeud *gauche, *droite ;
    int valeur ;
} NoeudAst ;

typedef NoeudAst *Ast ;
```

Des primitives de construction (ast_construction.h)

```
Ast creer_operation(TypeOperateur opr , Ast op_gauche , Ast op_droit);
// renvoie un Ast (op_gauche, opr, op_droit) de nature OPERATION

Ast creer_valeur(int val) ;
// renvoie un Ast "feuille", de nature VALEUR et de valeur val
```

Construire un Ast lors de l'analyse

construction ascendante : feuilles \rightarrow racine

\Rightarrow **Enrichir l'analyseur** : ajouter des paramètres à chaque procédure

- le (sous) AsT **résultat** produit
- *parfois*, un sous AsT **donné** auxiliaire

```
Rec_facteur (resultat A : AsT) =  
  selon LC.nature  
    cas ENTIER : A := creer_valeur (LC.valeur) ; Avancer  
    cas PARO : Avancer ; Rec_eag (A) ;  
              si LC.nature = PARF alors Avancer sinon Erreur  
    autre : Erreur  
fin
```

Attention : en C les paramètres résultats sont des **pointeurs** !

```
*A = creer_valeur (LC.valeur)
```

Construire un Ast lors de l'analyse

construction ascendante : feuilles \rightarrow racine

\Rightarrow **Enrichir l'analyseur** : ajouter des paramètres à chaque procédure

- le (sous) AsT **résultat** produit
- *parfois*, un sous AsT **donné** auxiliaire

```
Rec_facteur (resultat A : AsT) =  
  selon LC.nature  
    cas ENTIER : A := creer_valeur (LC.valeur) ; Avancer  
    cas PARO : Avancer ; Rec_eag (A) ;  
              si LC.nature = PARF alors Avancer sinon Erreur  
    autre : Erreur  
fin
```

Attention : en C les paramètres résultats sont des **pointeurs** !

***A** = creer_valeur (LC.valeur)

Construction de l'AsT (1)

eag \rightarrow *seq_terme*

Rec_eag(**A** : resultat AsT) =
 Rec_seq_terme(**A**) ;

seq_terme \rightarrow *terme suite_seq_terme*

Rec_seq_terme(**A** : resultat AsT) =
A1: Ast
Rec_terme(**A1**) ; Rec_suite_seq_terme(**A1**, **A**) ;

Construction de l'AsT (1)

eag \rightarrow *seq_term*

Rec_eag(*A* : resultat AsT) =
 Rec_seq_term(*A*) ;

seq_term \rightarrow *terme suite_seq_term*

Rec_seq_term(*A* : resultat AsT) =
A1: Ast
Rec_term(*A1*) ; Rec_suite_seq_term(*A1*, *A*) ;

Construction de l'AsT (2)

suite_seq_terme \rightarrow *op1 terme suite_seq_terme* [PLUS, MOINS]
suite_seq_terme \rightarrow ε

```
Rec_suite_seq_terme(Ag : donnee Ast, A : resultat AsT) =  
  Ad, A1: Ast  
  Op : TypeOperateur  
  selon LC().nature // LC est le lexeme_courant()  
    cas PLUS, MOINS :  
      Rec_op1(Op) ; Rec_terme(Ad) ;  
      A1 = creer_operation(Op, Ag, Ad) ;  
      Rec_suite_seq_terme(A1, A) ;  
    autre : A = Ag // Ne pas oublier !!!  
fin
```

Idem pour Rec_seq_facteur() et Rec_suite_seq_facteur() ...

Construction de l'AsT (2)

suite_seq_terme \rightarrow *op1 terme suite_seq_terme* [PLUS, MOINS]
suite_seq_terme \rightarrow ε

```
Rec_suite_seq_terme(Ag : donnee Ast, A : resultat AsT) =  
  Ad, A1: Ast  
  Op : TypeOperateur  
  selon LC().nature // LC est le lexeme_courant()  
    cas PLUS, MOINS :  
      Rec_op1(Op) ; Rec_terme(Ad) ;  
      A1 = creer_operation(Op, Ag, Ad) ;  
      Rec_suite_seq_terme(A1, A) ;  
    autre : A = Ag // Ne pas oublier !!!  
fin
```

Idem pour Rec_seq_facteur() et Rec_suite_seq_facteur() ...

Construction de l'AsT (2)

suite_seq_terme \rightarrow *op1 terme suite_seq_terme* [PLUS, MOINS]

suite_seq_terme \rightarrow ε

```
Rec_suite_seq_terme(Ag : donnee Ast, A : resultat AsT) =  
  Ad, A1: Ast  
  Op : TypeOperateur  
  selon LC().nature // LC est le lexeme_courant()  
    cas PLUS, MOINS :  
      Rec_op1(Op) ; Rec_terme(Ad) ;  
      A1 = creer_operation(Op, Ag, Ad) ;  
      Rec_suite_seq_terme(A1, A) ;  
    autre : A = Ag // Ne pas oublier !!!  
fin
```

Idem pour Rec_seq_facteur() et Rec_suite_seq_facteur() ...

Construction de l'AsT (3)

facteur → ENTIER

facteur → PARO eag PARF

```
Rec_facteur (resultat A : AsT) =  
  selon LC.nature  
    cas ENTIER : A := creer_valeur (LC.valeur) ; Avancer  
    cas PARO : Avancer ; Rec_eag (A) ;  
        si LC.nature = PARF alors Avancer sinon Erreur  
    autre : Erreur  
fin
```

op1 → PLUS

op1 → MOINS

```
Rec_op1 (resultat Op : TypeOperateur) =  
  selon LC.nature  
    cas PLUS : Op := N_PLUS ; Avancer  
    cas MOINS : Op := N_MOINS ; Avancer  
    autre : Erreur  
fin
```

Construction de l'AsT (3)

facteur → ENTIER

facteur → PARO eag PARF

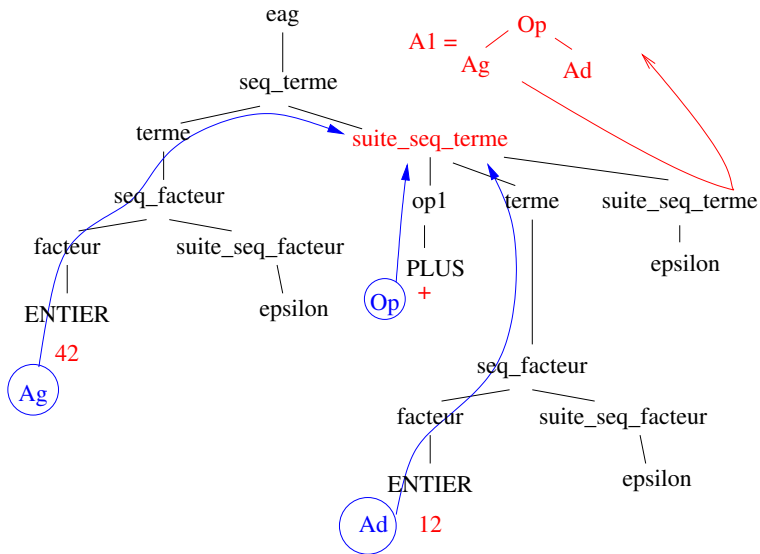
```
Rec_facteur (resultat A : AsT) =  
  selon LC.nature  
    cas ENTIER : A := creer_valeur (LC.valeur) ; Avancer  
    cas PARO : Avancer ; Rec_eag (A) ;  
    si LC.nature = PARF alors Avancer sinon Erreur  
  autre : Erreur  
fin
```

op1 → PLUS

op1 → MOINS

```
Rec_op1 (resultat Op : TypeOperateur) =  
  selon LC.nature  
    cas PLUS : Op := N_PLUS ; Avancer  
    cas MOINS : Op := N_MOINS ; Avancer  
  autre : Erreur  
fin
```

Exemple : 42 + 12



Calculer la valeur d'une EAG

Un (simple !) parcours **récuratif** de l'AsT ...

```
fonction Evaluer (A : Ast) : entier
  Vg, Vd : entier
  selon A.Nature
    cas VALEUR : // feuille
      retourner A.valeur
    cas OPERATION : // operateur
      Vg = evaluer(A.fg) ;
      Vd = evaluer(A.fd) ;
      selon A.Operateur :
        cas N_PLUS : retourner Vg + Vd
        cas N_MOINS : ...
      fin
  fin
fin
```

Dans la suite (le dernier TP “calcullette” !) ...

- ❶ **Finir** d'écrire l'analyseur syntaxique (sans construction de l'AsT) pour le langage *eag* à partir de la **Grammaire**.
- ❷ Etendre cet analyseur syntaxique pour produire l'AsT.
- ❸ Ecrire la fonction de calcul de la valeur d'un Ast
- ❹ Intégrer le tout dans le programme principal ...