République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumédiène

Faculté d'Electronique et d'Informatique
Département Informatique

Master Systèmes Informatiques intelligents

Module : Conception et Complexité des Algorithmes

Rapport de projet de TP

Réalisé par :

Année universitaire : 2021 / 2022

Chapitre 1

Représentation d'une expression arithmétique en arbre binaire

1.1 Description de l'objectif de l'algorithme

Une expression arithmétique est une succession de caractères mathématiques notamment des nombres, des opérateurs mathématiques (+ addition, - soustraction, * multiplication, / division et * modulo) et des symboles impropres pour indiquer la priorité(( ) les parenthèses). Il faut noter cependant que les opérateurs mathématiques ne possèdent pas tous la même priorité ; La multiplication et la division sont plus prioritaires que l'addition et la soustraction. Dans le cas où la priorité de deux opérateurs est la même, celui le plus à gauche dans l'expression arithmétique devient plus prioritaire que celui à droite. En considérant ces contraintes, une des représentations les plus adaptées pour représenter une expression arithmétique et sous forme d'arbre binaire.

Un arbre binaire est une structure de données complexe, il est caractérisé par un élément racine qui contient à son tour un chemin vers un ou deux autres éléments appelés fils droit et fils gauche. Chaque élément intermédiaire de l'arbre par la suite a la même structure que la racine, jusqu'à arriver aux éléments terminaux qui ne possèdent pas de fils qu'on nomme les feuilles de l'arbre.

Nous pouvons alors par extrapolation représenter chaque opération arithmétique par un élément de l'arbre, telle que l'élément de l'arbre content l'opérateur, et les deux opérandes sont contenus dans les deux fils de l'élément. Pour qu'un arbre binaire puisse représenter la structure d'une expression arithmétique correctement, ce dernier doit respecter l'ordre et la priorité des opérations. Les opérations les moins prioritaires sont en haut de l'arbre car elles dépendent du résultat des opérations plus prioritaires qui elles sont plus bas dans l'arbre binaire (voir Figure 1.1).

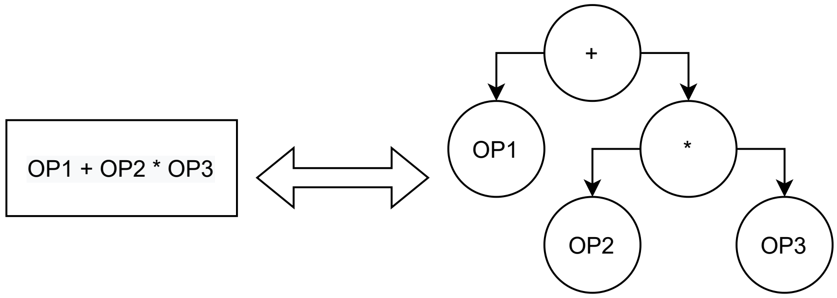

FIGURE 1.1 – Représentation d'une expression arithmétique en arbre binaire

1.2 Fonctionnement de l'algorithme

Le passage de l'expression arithmétique en arbre binaire passe par 2 étapes de traduction : l'analyse lexicale et puis la traduction dirigée par la syntaxe (voir Figure 1.2). L'expression arithmétique est d'abord lue du clavier sous forme de chaîne de caractères. Puis, un vecteur d'entités lexicales est généré à partir de cette chaîne. Enfin, on génère un arbre binaire à partir du vecteur d'entités lexicales.

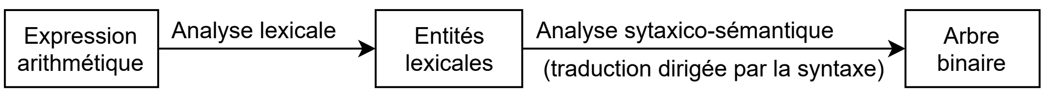

FIGURE 1.2 – Les étapes de la traduction

Les structures de données utilisées dans les algorithmes qui vont suivre sont les suivantes :

entité : représente une entité lexicale et contient les champs type de l'entité et la valeur.

noeud : représente un élément d'un arbre et contient une valeur et des pointeurs vers les fils droit et gauche du noeud.

arbre : représente un arbre binaire et contient un pointeur vers la racine de l'arbre.

1.2.1 Analyse lexicale

Dans cette partie, on extrait les entités lexicales de la chaîne de caractères contenant l'expression arithmétique. On parcourt la chaîne de caractères depuis le début, et on génère les entités lexicales selon les caractères lus.

```
Algorithme 1 : Analyse lexicale
  Données : expression : chaîne de caractères
  Résultat : entites : vecteur d'entités
  Variables :
  e : entité;
  i, j : entier;
  début
  |  Initialiser le vecteur entites à vide;
  |  pour i ← 1à expression.taille() faire
  |  |  si expression[i] est un opérateur alors
  |  |  |  // Reconnaitre un opérateur
  |  |  |  e.type←opérateur;
  |  |  |  e.operation←type_opération;
  |  |  |  entites.ajouter(e);
  |  |  sinon si expression[i] est une parenthèse alors
  |  |  |  // Reconnaitre une parenthèse
  |  |  |  e.type←parenthèse;
  |  |  |  e.parenthèse←type_parenthèse;
  |  |  |  entites.ajouter(e);
  |  |  sinon si expression[i] est un chiffre ou un point alors
  |  |  |  // Reconnaitre un nombre
  |  |  |  j←1;
  |  |  |  tant que i + j ≤ expression.taille() et expression[i] est un chiffre ou un point
  |  |  |   faire j←j + 1;
  |  |  |  e.type←nombre;
  |  |  |  e.nombre←en_nombre(expression.sous_chaine(i, j));
  |  |  |  entites.ajouter(e);
  |  |  |  i←i + j − 1;
  |  |  sinon
  |  |  |  Lever une exception;                              // Erreur lexicale
  |  |  fin si
  |  fin pour
  fin
```

1.2.2 Analyse syntaxico-sémantique ou traduction dirigée par la syntaxe

Nous utilisons l'algorithme de descente récursive pour parcourir le vecteur d'entités et générer l'arbre binaire. Dans cet algorithme chaque MGP (membre de gauche de production) est associé à une fonction dans le programme. L'algorithme commence en appelant une première fois l'axiome de la grammaire et se termine une fois que tout le vecteur est parcouru.

La grammaire utilisée est comme suit :

$$E \rightarrow T + T * |T - T*$$

$$T \rightarrow F + F * |F - F*$$

$$F \rightarrow nb|(E)| + F| - F$$

Algorithme 2 : Descente récursive

Données : expression : chaîne de caractères

Résultat : entites : vecteur d'entités

Variables :

tc : entier;

bt : arbre;

début

ct←0;

si $tc = entites.taille()$ **alors** Lever une exception `// Expression vide;`

bt←e(entites, tc); `// Appel de la fonction e`

retourner bt ;

fin

Cette fonction e traite les opérations d'addition et de soustraction qui sont moins prioritaires, et génère les noeuds correspondants. Le traitement des autres opérations est délégué à la fonction t qui va terminer son exécution avant la fonction e ; c-à-d. que les noeuds générés par les opérations plus prioritaires seront retournés à la fonction e qui va les ajouter dans les niveaux plus bas.

```
Fonction e(entites : vecteur d'entités, Entrée/Sortie tc : entier) : arbre
  Variable :
  bt, rt, lt : arbre;
  e : entité;
  début
    bt←t(entites, tc);                                   // Appel de la fonction t
    tant que tc ≤ entites.taille() et entites[tc] est un opérateur d'addition ou de
     soustraction faire
      e←entites[tc];
      lt←bt;
      tc←tc + 1;
      si tc > entites.taille() alors Lever une exception      // Erreur syntaxique;
      rt←t(entites, tc);                                 // Appel de la fonction t
      bt.racine←e;
      bt.fils_gauche←lt;
      bt.fils_droit←rt;
    fin tq
    retourner bt;
  fin
```

Cette fonction t traite les opérations de multiplication, de division et de modulo. Elle utilise la fonction f pour générer les noeuds des nombres et des sous-expressions plus prioritaires comme les parenthèses.

```
Fonction t(entites : vecteur d'entités, Entrée/Sortie tc : entier) : arbre
  Variable :
  bt, rt, lt : arbre;
  e : entité;
  début
    bt←f(entites, tc);                                    // Appel de la fonction f
    tant que tc ≤ entites.taille() et entites[tc] est un opérateur de multiplication ou de
     division ou de modulo faire
      e←entites[tc];
      lt←bt;
      tc←tc + 1;
      si tc > entites.taille() alors Lever une exception       // Erreur syntaxique;
      rt←f(entites, tc);                                  // Appel de la fonction f
      bt.racine←e;
      bt.fils_gauche←lt;
      bt.fils_droit←rt;
    fin tq
    retourner bt;
  fin
```

Cette fonction f génère les noeuds des nombres et des opérateurs unaires, elle repasse le contrôle à la fonction e en cas d'utilisation de parenthèses pour traiter la sous-expression contenue dedans.

```
Fonction f(entites : vecteur d'entités, Entrée/Sortie tc : entier) : arbre
  Variable :
  bt, rt, lt : arbre;
  e, gauche, droit : entité;
  début
    si entites[tc] est une parenthèse ouvrante alors
      tc←tc + 1;
      si tc > entites.taille() alors Lever une exception        // Erreur syntaxique;
      bt←e(entites, tc);                                       // Appel de la fonction e
      si tc ≤ entites.taille() et entites[tc] est une parenthèse fermante alors
        tc←tc + 1;
      sinon
        Lever une exception                                     // Erreur Syntaxique
      fin si
    sinon si entites[tc] est un opérateur d'addition ou de soustraction alors
      // Cas d'un opérateur unaire
      e←entites[ct];
      gauche.type←type_nombre;
      gauche.nombre←0;
      tc←tc + 1;
      si tc > entites.taille() alors Lever une exception        // Erreur syntaxique;
      rt←f(entites, tc);                                       // Appel de la fonction f
      lt.racine←gauche;
      bt.racine←e;
      bt.fils_gauche←lt;
      bt.fils_droit←rt;
    sinon
      // Cas d'un nombre (feuille de l'arbre)
      e←entites[tc];
      bt.racine←e;
      tc←tc + 1;
    fin si
    retourner bt;
  fin
```

1.3 Calcul de complexité

1.3.1 Complexité temporelle

La complexité de l'analyse lexicale est toujours égale à la longueur de la chaîne c-à-d. : $\mathcal{O}(n)$ tel que n est la longueur de l'expression en de caractères.

La complexité de l'analyse syntaxique est quant à elle égale à la longueur du vecteur d'entités lexicales, car il est parcouru une seule fois, c-à-d. : $\mathcal{O}(n')$ tel que n' est la longueur du vecteur d'entités.

La complexité temporelle de l'algorithme devient alors : $\mathcal{O}(n + n')$.

1.3.2 Complexité spatiale

L'expression arithmétique est d'abord stockée dans une chaîne de caractères de longueur n, puis dans un vecteur d'entités de longueur n', puis dans un arbre binaire qui contient lui aussi n' éléments.

Sachant qu'une entité contient le type de l'entité et la valeur en elle-même, cependant la valeur est stockée comme un nombre réel qui prend 8 octets comparé à la représentation en chaîne de caractères où chaque caractère est sous 1 octet.

Donc la taille d'un élément d'une chaîne de caractères est 1 unité, la taille d'une entité est 9 unités, et la taille d'un élément d'un arbre est égale à $9 + 4 * 2$ unités (la taille d'un pointeur est égale à 4 unités) donc 17 unités.

La complexité spatiale est égale à la somme des trois complexités : $n * 1 + n' * 9 + n' * 17 = n + 26 * n' \approx \mathcal{O}(n + n')$

1.4 Expérimentation

Le tableau suivant représente les temps d'exécution en nanoseconde de l'algorithme selon la variation de la taille de l'expression arithmétique en d'opérande.

N	10	50	100	500	1000	5000	10000	100000	1000000
t1(ns)	177584	437769	381575	418252	686466	3111420	6786260	60986100	642610000
t2(ns)	101653	193648	339100	354777	2322770	3307840	20424500	59967200	639145000
t3(ns)	31972	47574	216601	354751	2115200	3375350	7450650	60956700	639422000
t4(ns)	100708	149167	259485	337421	820084	3283810	19698800	61631500	641354000
t5(ns)	24343	140434	95821	345987	683119	5158840	7575520	61750100	640897000
t6(ns)	30901	161277	337395	807634	671850	3292160	21818400	61473700	641686000
t7(ns)	52941	140159	288308	329206	693785	4360970	8418610	60870500	667895000
t8(ns)	69185	163825	242197	520649	2050330	5460620	22419800	63008000	639987000
t9(ns)	69510	97011	93270	356979	692529	5279490	6420450	61725500	640873000
t10(ns)	83819	142928	92484	1071680	803473	13108600	6558390	60088300	639895000
Moyenne(ns)	74262	167379	234624	489734	1153961	4973910	12757138	61245760	643376400

La figure suivante (voir Figure 1.3) représente l'évolution du temps d'exécution selon la longueur de l'expression arithmétique en d'opérande.

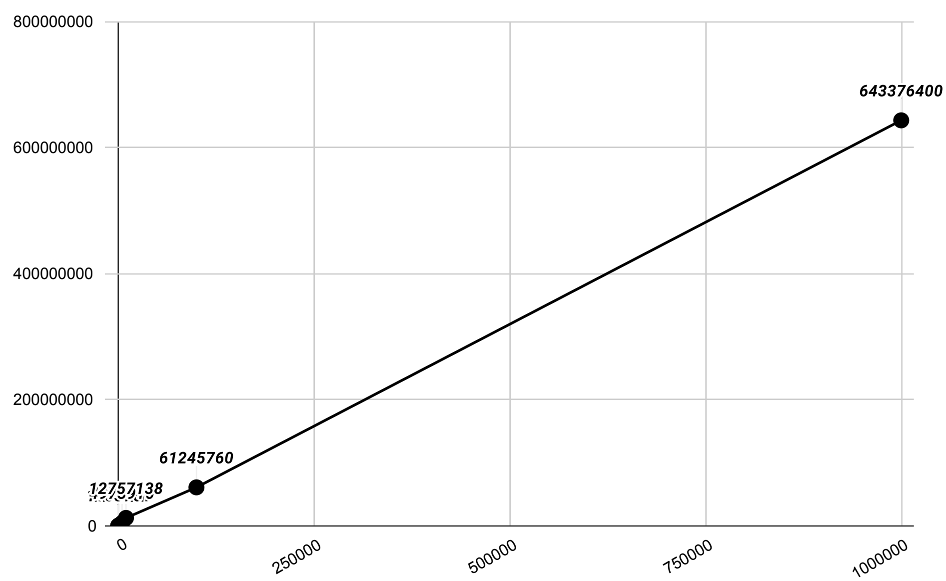

FIGURE 1.3 – Temps d'exécution du programme selon la longueur de l'expression en d'opérandes

Depuis le graphe, la courbe est sous forme d'une droite, on observe que le temps d'exécution évolue de manière linéaire avec l'augmentation de la taille du problème, ce qui correspond bien à la complexité théorique calculée auparavant.

1.5 Conclusion

L'algorithme de la descente récursive propose une complexité optimale pour la représentation de l'expression arithmétique en arbre binaire, égale à la longueur de l'expression arithmétique, car il parcourt l'expression un nombre minimal de fois (une seule fois). Nous avons bien vu pendant les expériences que l'évolution du temps d'exécution d'une telle complexité est relativement contrôlée et suit une courbe droite et linéaire.