Transfert d'Informatique n^{o} 3

1ère année

Silke WEBER
Mathieu GOUTELLE
Groupe 2C (Responsable : Christian VIAL)

22 mai 2000

Texte mis en forme avec LaTeX 2ε

1 Présentation

1.1 Données du problème

On dispose d'un fichier texte sur disque contenant des instructions d'affectation du genre BASIC sous la forme identificateur = expression arithmétique entière sans opérateur unaire. Ce fichier contient une instruction par ligne, l'instruction pouvant être correcte ou erronée. On se propose de vérifier la correction syntaxique de chaque ligne, d'indiquer clairement les erreurs (place et libellé) quand il y en a et d'évaluer les expressions de manière à affecter des valeurs aux variables.

Un fichier test appelé `validat1.don` sera fourni sur les machines de l'ECL par les enseignants. Les programmes écrits par les élèves devront être capables de traiter ce fichier, ce qui n'empêche de proposer d'autres fichiers test, surtout dans la phase de mise au point. Le fichier test pourra contenir par exemple :

```
ada = 23
toto   = 78
d = 2*ada + 3*toto
e = 66 + 37/2
```

1.2 Résultats à produire

Pour chaque ligne lue sur le fichier test, on effectuera le travail suivant :

- écriture de la ligne sur l'écran ;
- écriture sur l'écran d'une indication de la validité ou non de cette ligne ;
- écriture du libellé de l'erreur s'il y en a une dans la ligne ;
- écriture de la valeur prise par la variable située à gauche du symbole =.

1.3 Structure de l'application

L'application finale sera composée de cinq modules :

- un module analyseur lexical pour extraire les symboles lexicaux du texte d'entrée ;
- un module analyseur syntaxique pour vérifier la correction grammaticale de chaque expression ;
- un module table pour gérer la table des symboles ;
- un module évaluation pour évaluer la ligne ;
- un module application appelant les trois modules précédents.

2 Analyse du problème

2.1 Table des symboles

Chaque identificateur sera stocké avec sa valeur dans une table qui sera initialisée lors de l'analyse lexicale et utilisée et mise à jour lors de l'évaluation. Elle sera représentée comme un tableau (cf. FIGURE 1). On pourra aussi lui adjoindre deux champs contenant respectivement la taille maximale de la table et l'indice de la première position libre.

Référence	Nom	Valeur
1	toto	15
2	lulu	20
3	x	

FIGURE 1 – *Exemple de table des symboles*

Des utilitaires d'initialisation et de gestion (recherche de nom, de valeur, affichage...) de la table devront être prévus afin de faciliter l'utilisation.

2.2 Évaluation — Arbre syntaxique

Pour réaliser l'évaluation de l'expression, on construit l'arbre syntaxique binaire correspondant lors de l'analyse syntaxique. Par exemple, on représentera les expressions $12+3*6$ et $x=2+y$ par les arbres de la FIGURE 2.

FIGURE 2 – *Exemples de représentations d'expressions sous forme d'arbre*

L'évaluation se fera récursivement pour chaque nœud de l'arbre : on évalue les fils gauche et droit du nœud et on renvoie le résultat de l'opération définie par le nœud traité. On définira, comme pour la table des symboles, une batterie d'utilitaires de création, d'évaluation et d'affichage des arbres pour faciliter leur utilisation.

2.3 Analyse lexicale

L'analyse lexicale consiste à lire l'expression et à identifier les différents lexèmes (constante entière, identificateur, opérateur, parenthèse...). Cette étape va produire une liste de lexèmes différenciés selon cinq grands types :

- les opérateurs pour les signes $\{+,-,*,/,(,),=\}$;
- les constantes entières ;
- les identificateurs ;
- la marque de fin de ligne ;
- un lexème codant la présence d'un caractère erroné.

C'est aussi le module d'analyse lexicale qui se charge d'ajouter une nouvelle entrée dans la table des symboles (cf. § 2.1) si un nouvel identificateur est trouvé.

Si le lexème est une constante, une variable ou un opérateur, il contiendra en plus de son type un champ d'information associée (la valeur de la constante, la référence de l'identificateur dans la table des symboles ou le type d'opérateur).

2.4 Analyse syntaxique

L'analyse syntaxique consiste principalement à vérifier la correction d'une liste de lexèmes en fonction d'un grammaire donnée. Dans notre cas, la grammaire définissant les expressions arithmétiques constantes entières peut être décrite, en notation BNF (rappelée à la FIGURE 3), comme à la FIGURE 4.

réécriture	::
terminal	**XXX**
non terminal	*XXX*
et	,
ou	;
zéro ou plusieurs itérations	{...}
une ou plusieurs itérations	{...}+
composant optionnel	[...]
regroupement	(...)
fin de production	.

FIGURE 3 – *Description de la notation* BNF

ligne :: [*instruction*] ; **finligne**.
instruction :: **identificateur**, **=**, *expression.*
expression :: *terme*, {(**+** ; **-**), *terme*}.
terme :: *facteur*, {(***** ; /), *facteur*}.
facteur :: **constante** ; **identificateur** ; (**(**, *expression*, **)**).

FIGURE 4 – *Description de la grammaire en notation* BNF

Comme l'analyse lexicale nous donne une liste de lexèmes, il est plus intéressant de réécrire cette grammaire comme à la FIGURE 5 pour faciliter la construction récursive de cette liste. On peut aussi, sans difficulté supplémentaire, ajouter les opérateurs unaires de signe.

ligne :: [*expression*] ; **finligne**.
instruction :: **identificateur**, **=**, *expression.*
expression :: *terme*, [(**+** ; **-**), *expression*].
terme :: *facteur*, [(***** ; /), *terme*].
facteur :: ([**+** ; **-**], **constante**) ; **identificateur** ; (**(**, *expression*, **)**).

FIGURE 5 – *Description de la grammaire pour traitement récursif*

Ensuite, le module d'analyse syntaxique doit traduire cette grammaire pour tester la correction de la liste de lexèmes. Cela va être réalisé par un certain nombre de fonctions mutuellement récursives, correspondant à chaque ligne de la définition BNF, décrites respectivement dans les algorithmes suivants. Chaque fonction rendra comme résultat un booléen selon le succès ou non de l'analyse syntaxique au niveau de récursivité de la fonction et prendra soin de l'avancement dans la liste des lexèmes grâce à la fonction `suivant`. On prendra cette propriété comme une hypothèse de récurrence. La fonction `suivant` est simplement une fonction qui fait avancer la liste des lexèmes et rend faux si l'on atteint la fin de cette liste, vrai sinon.

Algorithme 1 Fonction `lit_ligne`

```
    Fonction lit_ligne() : booléen
2   Si (le lexème est une fin de ligne) Alors
        Retourner(Vrai)
4   Sinon
        Retourner(lit_instr() et (le lexème est une fin de ligne))
6   Fin Si
```

On devra néanmoins penser lors de l'implémentation à traiter les différents types d'erreur pour indiquer non seulement que l'expression est correcte ou incorrecte mais aussi, le cas échéant, une indication sur l'endroit où l'erreur a lieu et le type d'erreur rencontrée. Cela pourra être réalisé simplement en modifiant au fur et à mesure une variable contenant le type de la première erreur rencontrée lors de l'analyse lexicale (caractère non attendu) et syntaxique.

2.5 Construction de l'arbre — Évaluation

L'arbre syntaxique sera construit au fur et à mesure de l'analyse syntaxique en effectuant des actions sémantiques lors des différentes étapes. Cela revient à créer le sous-arbre correspondant aux non terminaux trouvés et à les associer par l'opération qui les relient. Ainsi, on construit récursivement l'arbre. Par exemple, si l'on se trouve dans la situation : *terme* + *expression*, on construit l'arbre associé à *terme*, puis celui associé à *expression* et enfin on crée l'arbre ayant pour racine '+' et pour fils gauche et droit respectivement *terme* et *expression*. L'évaluation reviendra alors à réaliser un parcours de l'arbre en évaluant chaque opération correspondant aux nœuds avec les valeurs des fils gauche et droit.

Algorithme 2 Fonction `lit_instr`

```
   Fonction lit_instr() : booléen
 2 Si (le lexème est un identificateur) Alors
     Retourner(suivant() et (le lexème est un '=') et lit_expr())
 4 Sinon
     Retourner(Faux)
 6 Fin Si
```

Algorithme 3 Fonction `lit_expr`

```
   Fonction lit_expr() : booléen
 2 Si (lit_terme()) Alors {On doit avoir un terme}
     Si (le lexème est un '+' ou un '-') Alors
 4     Retourner(suivant() et lit_expr())
     Sinon
 6     Retourner(Vrai)
     Fin Si
 8 Sinon
     Retourner(Faux)
10 Fin Si
```

Algorithme 4 Fonction `lit_terme`

```
   Fonction lit_terme() : booléen
 2 Si (lit_fact()) Alors {On doit avoir un facteur}
     Si (le lexème est un '*' ou un '/') Alors
 4     Retourner(suivant() et lit_terme())
     Sinon
 6     Retourner(Vrai)
     Fin Si
 8 Sinon
     Retourner(Faux)
10 Fin Si
```

Algorithme 5 Fonction `lit_fact`

```
   Fonction lit_fact() : booléen
 2 Si (le lexème est une '(') Alors
     Retourner(lit_expr() et (le lexème est une ')') et suivant())
 4 Sinon
     Retourner((lit_cste() ou (le lexème est un identificateur) et suivant()))
 6 Fin Si
```

Algorithme 6 Fonction `lit_cste`

```
   Fonction lit_cste() : booléen
2  Si (le lexème est un '+' ou un '-') Alors
     {On doit avoir une constante après}
4    Retourner(suivant() et (le lexème est une constante))
   Sinon
6    Retourner(le lexème est une constante)
   Fin Si
```

2.6 Application principale

Ce porgramme devra gérer l'ouverture, la lecture et la fermeture du fichier de test, mettre en œuvre chronologiquement les modules précédents et procéder aux différents affichages nécessaires.

On devra notamment penser à traiter les erreurs d'entrée/sortie (fichier non présent par exemple). La lecture du fichier texte se fera ligne par ligne jusqu'à la fin du fichier et on analysera chaque ligne du fichier, grâce aux fonctions d'analyse lexicale et syntaxique définies auparavant, que l'on évaluera si la syntaxe est correct.

3 Implémentation en C++

Nous avons pris ici le parti de ne pas commenter les sources des programmes afin de ne pas les surcharger. Les remarques que nous pouvons avoir à faire apparaissent dans les paragraphes suivant le programme. Ce programme a été compilé avec le compilateur C++ DJGPP *(norme* ISO/ANSI*) : il peut donc apparaître des différences par rapport au compilateur* Microsoft Visual C++ *présent sur les machines mises à disposition, notamment au niveau des bibliothèques et des déclarations des fonctions externes à celles-ci dans un projet.*

3.1 Types et structures

Le type de la table des symboles est plus ou moins un tableau de symboles (nom et valeur) auquel on a ajouté deux champs : l'un pour la taille maximale, l'autre pour désigner le premier emplacement libre de la table.

PROGRAMME 1: *Type de la table des symboles*

```
struct symbole {
2  char nom[20];
   int valeur;
4 };

6 struct table {
   int taille;
8  int libre;
   symbole *cases;
10 };
```

Le type des lexèmes doit être défini (cf. PROGRAMME 2). Pour cela, on peut le symboliser par une « boîte » contenant le type du lexème (opérateur, constante, identificateur, fin de ligne ou erreur) et un champ information qui vaudra, selon le type, la valeur de la constante, la référence de l'identificateur ou le nom de l'opérateur. Le type `genre` est donc un type énuméré selon les

différents types de lexèmes. La structure `boite` contient un champ de type `genre` et le champ d'information.

PROGRAMME 2: *Type des lexèmes*

```
enum genre {op, cste , ident , finl , err };

struct boite {
    genre g;
    union {char operateur ; int val ; int ref ;};
};
```

Pour l'analyse lexicale, il faut créer un type de liste de lexèmes : on utilise pour cela une structure contenant une `boite` et un pointeur vers l'élément suivant de la liste (cf. PROGRAMME 3). Finalement, le type `lexliste` sera un alias pour un pointeur vers le début de la liste.

PROGRAMME 3: *Type de liste de lexèmes*

```
struct lexeme {
    boite elt ;
    lexeme *succ;
};

typedef lexeme * lexliste ;
```

Les arbres syntaxiques sont des arbres binaires défini comme suit : chaque nœud sera un lexème comme décrit précédemment, les deux fils étant eux même des arbres syntaxiques. On ne différencie pas les feuilles de l'arbre qui seront représentées comme des nœuds ayant deux fils vides.

PROGRAMME 4: *Type des arbres syntaxiques*

```
struct noeud {
   boite racine ;
   noeud * filsg ;
   noeud * filsd ;
};

typedef noeud *arbre ;
```

3.2 Utilitaires

Pour faciliter la programmation, on a besoin de quelques utilitaires. D'abord, on écrit une fonction qui ajoute un lexème en tête d'une liste (cf. PROGRAMME 5). Ensuite, pour pouvoir afficher la liste des lexèmes, on écrit un utilitaire `affiche_liste` réalisant récursivement l'affichage de chaque lexème (cf. PROGRAMME 6). Ensuite, et sur le même principe que les fonctions prédéfinies `isdigit` et `isalpha`, on écrit une fonction `isop` (cf. PROGRAMME 7) qui permet de tester si un caractère est un opérateur (opération ou parenthèse) ou non.

Enfin, pour extraire des expressions les constantes et les noms des identificateurs, on définit les fonctions `extract_cste` et `extract_ident` (cf. PROGRAMME 8) qui à partir d'un indice *i* passé en argument extrait respectivement tous les chiffres consécutifs pour former l'entier correspondant ou tous les caractères consécutifs pour former le nom correspondant. Il faut noter qu'elle modifie

PROGRAMME 5: *Utilitaire* `cons`

```
lexliste cons(boite a, lexliste l) {
  lexliste temp = new lexeme;
  (temp -> elt) = a;
  (temp -> succ) = l;
  return(temp);
};
```

PROGRAMME 6: *Utilitaire* `affiche_liste`

```
void affiche_boite (boite b) {
  switch (b.g) {
    case op    : printf ("%c ", b.operateur ); break;
    case cste  : printf ("%d ", b.val ); break;
    case finl  : putchar ('\n'); break;
    case ident : printf ("%s ", tabl_ident .cases[b.ref ].nom); break;
    case err   : printf ("Erreur "); break;
  };
};

void affiche_liste ( lexliste l) {
  if (l != NULL) {
    affiche_boite (l -> elt );
    affiche_liste (l -> succ);
  };
};
```

PROGRAMME 7: *Utilitaire* `isop`

```
int isop(char c) {
  switch (c) {
    case '+' : ;
    case '-' : ;
    case '*' : ;
    case '/' : ;
    case '=' : ;
    case '(' : ;
    case ')' : return(1);
    default  : return(0);
  };
};
```

la valeur de l'indice *i* (passé par adresse, ligne 1) pour que sa valeur soit incrémentée du nombre de chiffres de l'entier ou du nombre de lettres du nom.

PROGRAMME 8: *Utilitaires* `extract_cste` *et* `extract_ident`

```
int   extract_cste (char * texte , int &i ) {
2   int  temp = 0;

4   while ( isdigit ( texte [ i ])) {
      temp = 10*temp + texte [ i ] - '0';
6     i++;
    };
8   return(temp);
};
10
void  extract_ident (char * texte , int &i , char result [20]) {
12  int  j = 0;

14  while ( isalpha ( texte [ i ]) && ! isop ( texte [ i ])) {
      if ( j < 20) result [ j ] = texte [ i ];
16    i ++; j++;
    };
18
    result [ j ] = '\0';
20 };
```

3.3 Table des symboles

Pour la table des symboles, nous avons besoin d'un certain nombre d'utilitaires. D'abord, une procédure d'initialisation de la table doit être écrite (cf. PROGRAMME 9). Elle permettra notamment d'allouer l'espace mémoire pour la table et rendra 1 si il y a assez de mémoire pour stocker la table et 0 sinon.

PROGRAMME 9: *Fonction d'initialisation de la table des symboles*

```
int  init ( table &t ) {
2   t . taille = 20;
    t . libre = 0;
4   t . cases = calloc (20, sizeof (symbole));
    return( t . cases != NULL);
6 };
```

Ensuite, deux fonctions de recherche dans la table sont définies (cf. PROGRAMME 10), l'une pour déterminer la référence d'un identificateur à partir de son nom, l'autre pour récupérer sa valeur à partir de la référence dans la table. La fonction `stricmp` de la bibliothèque `string.h` permet de comparer deux chaînes de caractères.

Ensuite, on écrit une fonction (cf. PROGRAMME 11) qui permet de changer la valeur d'un identificateur de la table déterminé par sa référence. Cette fonction rendra 0 si la référence ne correspond pas à un identificateur préalablement défini.

Pour ajouter un symbole dans la table, on utilisera la fonction `ajout` (cf. PROGRAMME 12). Cette fonction ajoute l'élément passé en paramètre s'il reste de la place dans la table. Sinon, elle alloue de l'espace supplémentaire pour la table (10 symboles). Si cette dernière opération échoue (mémoire insuffisante), elle renvoie 0 et 1 dans tous les autres cas. On définit aussi la fonction

PROGRAMME 10: *Fonctions de recherche dans la table des symboles*

```
int get_ref(table t, char *s) {
  int i;
  for (i = 0; i < t.libre; i++)
    if (!stricmp(t.cases[i].nom, s)) return(i);
  return(-1);
};

int get_val(table t, int ref) {
  return(t.cases[ref].valeur);
};
```

PROGRAMME 11: *Fonction d'affectation de valeur*

```
int set_val(table &t, int ref, int val) {
  if (ref < t.libre) {
    t.cases[ref].valeur = val;
    return(1);
  };
  return(0);
};
```

`maj` qui ajoute l'élément seulement s'il n'existe pas et qui renvoie la référence de l'élément déjà présent ou ajouté.

Il ne reste plus qu'à définir une procédure d'affichage de la table (cf. PROGRAMME 13).

3.4 Arbre syntaxique

On a besoin de deux fonctions de création d'arbres (cf. PROGRAMME 14) : l'une créant un arbre à partir d'un nœud et de deux fils et l'autre créant une feuille, *i.e.* un nœud dont les deux fils sont vides. La deuxième utilisera donc la première.

On ajoute, comme pour la table des symboles, une procédure d'affichage (cf. PROGRAMME 15), qui permet de reconstruire l'expression représentée par l'arbre. Il faut cependant observer qu'il est difficile de représenter le parenthésage correct de l'expression sans le rendre redondant avec les règles de priorité usuelles.

Le plus important est quand même la fonction d'évaluation de l'arbre syntaxique (cf. PROGRAMME 16). Elle parcourt en profondeur l'arbre syntaxique et évalue chaque nœud en appliquant l'opération définie par le nœud sur les valeurs des fils gauche et droit (lignes 8-24) et ainsi de suite récursivement jusqu'à trouver une constante ou un identificateur dont on connaît la valeur (lignes 25-28). Pour les identificateurs, on utilise une variable externe (`tabl_ident`) qui contient la table des symboles.

3.5 Analyse lexicale

Pour l'étape d'analyse lexicale (cf. PROGRAMME 17), nous allons utiliser une fonction récursive, particulièrement adaptée à la construction de la liste des lexèmes. Cette fonction reçoit en paramètre une chaîne de caractères contenant l'expression à analyser et un entier représentant l'index du caractère courant dans la chaîne (il vaut zéro lors du premier appel à la fonction).

On teste d'abord si la chaîne est vide (lignes 6-9) : le cas échéant, on retourne la liste contenant le lexème correspondant à la fin de ligne. Sinon, on teste le caractère. S'il s'agit d'un opérateur, on crée le lexème correspondant et on continue l'analyse lexicale sur la suite (ligne 13). S'il s'agit d'un chiffre (lignes 16-20), on extrait la constante grâce à la fonction `extract_cste` (cf.

PROGRAMME 12: *Fonctions d'ajout d'identificateurs*

```
int ajout(table &t, char symb[20]) {
  if (t.libre < t.taille) {
    strcpy(t.cases[t.libre ++].nom, symb);
    return(1);
  }
  else {
    t.taille += 10;
    t.cases = realloc(t.cases, t.taille * sizeof(symbole));
    if (t.cases != NULL) {
      strcpy(t.cases[t.libre ++].nom, symb);
      t.cases[t.libre ++].valeur = 0;
      return(1);
    };
    return(0);
  };
};

int maj(table &t, char *s) {
  int temp = get_ref(t, s);

  if (temp == -1) {
    ajout(t, s);
    temp = t.libre - 1;
  };

  return(temp);
};
```

PROGRAMME 13: *Procédure d'affichage de la table des symboles*

```
void affiche_table (table t) {
  int j;
  for (j = 0; j < t.libre ; j++)
      printf ("%s = %d\n", t.cases[j].nom, t.cases[j].valeur );
};
```

PROGRAMME 14: *Fonctions de création d'arbres syntaxiques*

```
arbre enracine(arbre fg, boite x, arbre fd) {
  arbre result = new noeud;
  (result -> racine) = x;
  (result -> filsg) = fg;
  (result -> filsd) = fd;
  return(result);
};

arbre cree_noeud(boite x) {
  return(enracine(NULL, x, NULL));
};
```

PROGRAMME 15: *Fonction d'affichage d'arbres syntaxiques*

```
void affiche_arbre (arbre a) {
  if (a != NULL) {
    affiche_arbre (a -> filsg );
    affiche_boite (a -> racine );
    affiche_arbre (a -> filsd );
  };
};
```

PROGRAMME 16: *Fonction d'évaluation d'un arbre syntaxique*

```
int eval_arbre (arbre a) {
  boite temp;
  int calcul ;

  if (a != NULL) {
    temp = a -> racine ;
    switch (temp.g) {
      case op :
        switch (temp.operateur ) {
          case '+' : return( eval_arbre (a -> filsg ) + eval_arbre (a -> filsd ));
          case '-' : return( eval_arbre (a -> filsg ) - eval_arbre (a -> filsd ));
          case '*' : return( eval_arbre (a -> filsg ) * eval_arbre (a -> filsd ));
          case '/' : return( eval_arbre (a -> filsg ) / eval_arbre (a -> filsd ));
          case '=' : {
            calcul = eval_arbre (a -> filsd );
            set_val ( tabl_ident ,
                      ((a -> filsg ) -> racine ). ref ,
                      calcul );
            return( calcul );};
          };
      case ident : return( get_val ( tabl_ident , temp.ref ));
      case cste : return(temp.val );
    };
  };
  return (0);
};
```

PROGRAMME 8), on ajoute le lexème correspondant à la liste et on poursuit l'analyse. De même si l'on trouve un caractère (lignes 22-27), on extrait le nom de la variable et on utilise la fonction `maj` (cf. PROGRAMME 12) pour récupérer sa référence. Sinon, dans les autres cas (lignes 28-35), soit le caractère est un espace et on le « saute » pour poursuivre l'analyse, soit on ajoute un lexème `err` indiquant un caractère non reconnu. Dans ce cas, on met à jour une variable globale `erreur` (Pour plus de détails, voir la section suivante). On poursuit néanmoins l'analyse lexicale dans ce dernier cas.

PROGRAMME 17: *Fonction d'analyse lexicale*

```
lexliste an_lex(char *texte, int i) {
  int temp;
  boite tete;
  char name[20];

  if ((texte[i] == '\n') || (texte[i] == '\0')) {
    tete.g = finl;
    return(cons(tete, NULL));
  }
  else {
    if (isop(texte[i])) {
      tete.g = op; tete.operateur = texte[i];
      return(cons(tete, an_lex(texte, i+1)));
    }
    else {
      if (isdigit(texte[i])) {
        temp = extract_cste(texte, i);
        tete.g = cste; tete.val = temp;
        return(cons(tete, an_lex(texte, i)));
      }
      else {
        if (isalpha(texte[i])) {
          tete.g = ident;
          extract_ident(texte, i, name);
          tete.ref = maj(tabl_ident, name);
          return(cons(tete, an_lex(texte, i)));
        }
        else {
          if (isspace(texte[i]))
            return(an_lex(texte, i+1));
          else {
            tete.g = err;
            erreur = 1;
            return(cons(tete, an_lex(texte, i+1)));
          };
        };
      };
    };
  };
};
```

3.6 Analyse syntaxique

L'analyse syntaxique va utiliser une liste de lexèmes définie de façon externe à partir de la fonction principale. De même, pour le traitement des erreurs, on utilise une variable externe `erreur` de type entier qui représentera le type de la première erreur rencontrée.

Avant toute chose, on peut créer un certain nombre de tests (cf. PROGRAMME 18) facilitant la programmation et la lecture des fonctions d'analyse syntaxique. Ces fonctions vont tester la présence ou non d'un ou plusieurs opérateurs (+ et - par exemple). À la base de toutes ces fonctions est la fonction `lit_op` qui prend pour argument un caractère et teste si le lexème courant est l'opérateur correspondant à ce caractère. On met aussi à jour la variable `erreur` s'il y a lieu, *i.e.* si le lexème courant n'est pas une parenthèse fermante et si une erreur ne s'est pas déjà produite. Cette dernière condition est indispensable pour conserver la première erreur trouvée lors de l'analyse. On fait de même si le lexème courant n'est pas un identificateur ou le signe égal comme attendu.

PROGRAMME 18: *Tests de présence des opérateurs*

```
int test_g(genre x) { return((lexemes -> elt).g == x); };

int lit_op(char signe) {
  if (test_g(op))
    return((lexemes -> elt).operateur == signe);
  else
    return(0);
};

int lit_pm() { return(lit_op('+') || lit_op('-')); };

int lit_md() { return(lit_op('*') || lit_op('/')); };

int lit_po() { return(lit_op('(')); };

int lit_pf() {
  if (lit_op(')'))
    return(1);
  else
    if (erreur == 0) erreur = 3;
  return(0);
};

int lit_ident() {
  if (test_g(ident))
    return(1);
  else
    if (erreur == 0) erreur = 5;
  return(0);
};

int lit_eg() {
  if (lit_op('='))
    return(1);
  else
    if (erreur == 0) erreur = 6;
  return(0);
};
```

On utilisera aussi une fonction `suivant()` (cf. PROGRAMME 19) qui fera avancer la liste des lexèmes et renverra vrai si cette liste n'est pas vide et faux sinon.

PROGRAMME 19: *Procédure d'avancement de la liste*

```
int  suivant () {
  lexemes = lexemes –> succ;
  return(lexemes != NULL);
};
```

La fonction d'analyse syntaxique a déjà été décrite algorithmiquement au paragraphe 2.4. Il suffit donc de traduire ces algorithmes en C++ (cf. PROGRAMME 20) en ajoutant la gestion des erreurs. On commence (cf. PROGRAMME 20) par définir le prototype des fonctions d'analyse et par écrire que la fonction `an_syn` est en fait un alias pour la fonction `lit_ligne`.

PROGRAMME 20: *Prototypes des fonctions d'analyse lexicale*

```
int   lit_cste  ();
int   lit_fact  ();
int   lit_terme ();
int   lit_expr  ();
int   lit_instr  ();
int   lit_ligne  ();

int  an_syn() {
  return( lit_ligne  ());
};
```

Ensuite, on traduit simplement les algorithmes (cf. PROGRAMMES 21 et 22). Le test de la ligne 8 sert à conserver le code de la première erreur produite. Le code des erreurs est le suivant :

0 : Aucune erreur ne s'est produite ;

1 : Caractère non attendu dans l'expression ;

2 : Constante attendue ;

3 : Parenthèse fermante attendue ;

4 : Fin de ligne attendue ;

5 : Identificateur attendu ;

6 : Signe = attendu.

L'arbre syntaxique est créé récursivement. À chaque niveau de récursion, la variable externe `synt_tree` contient l'arbre déjà créé par les niveaux de récursion plus profonds. Ainsi, il suffit d'utiliser cette variable et de la mettre à jour selon la valeur du lexème courant.

3.7 Application principale

On commence (cf. PROGRAMME 23) par déclarer les variables externes (la variable contenant la liste des lexèmes, celle contenant le code de l'erreur, celle contenant la table des symboles et celle contenant l'arbre syntaxique). Ensuite, pour améliorer la lisibilité de la fonction principale, on écrit une fonction externe (cf. PROGRAMME 24) qui affiche le message d'erreur correspondant à la valeur de la variable `erreur`. Les valeurs 0 et 1 sont traités auparavant. Si toutefois, une autre valeur est traitée (lignes 16-19), un message d'erreur prévient l'utilisateur[1].

Pour l'utilisation du programme principal, le nom du fichier contenant les expressions doit être donné en argument de ligne de commande. Pour cela, on déclare (cf. PROGRAMME 25) la

1. Normalement, ce message ne doit jamais apparaître.

PROGRAMME 21: *Fonctions d'analyse syntaxique*

```
int   lit_ligne  () {
  if  ( test_g ( finl ))
    return (1);
  else
    if  (  lit_instr  () && test_g ( finl ))
      return (1);
    else  {
      if  ( erreur  == 0)  erreur  = 4;
      return (0);
    };
};

int   lit_instr  () {
  arbre  temp;
  boite  b;

  if  (  lit_ident  ()) {
    temp = cree_noeud(lexemes -> elt );
    if  ( suivant  ()) {
      b = lexemes -> elt ;
      if  ( lit_eg () && suivant () && lit_expr  ()) {
          synt_tree  =  enracine (temp , b ,  synt_tree );
          return (1);
      };
    };
  };
  return (0);
};

int   lit_expr  () {
  arbre  temp;
  boite  b;

  if  ( lit_terme  ())
    if  ( lit_pm ()) {
      temp =  synt_tree ;  b = lexemes -> elt ;
      if  ( suivant () && lit_expr  ()) {
        synt_tree  =  enracine (temp , b ,  synt_tree );
        return (1);
      };
      return (0);
    }
    else
      return (1);
  else
    return (0);
};
```

PROGRAMME 22: *Fonctions d'analyse syntaxique (Suite)*

```
48 int  lit_terme () {
     arbre temp;
50   boite b;

52   if ( lit_fact ())
       if (lit_md ()) {
54       temp = synt_tree ; b = lexemes –> elt ;
         if ( suivant() && lit_terme ()) {
56         synt_tree = enracine(temp, b, synt_tree );
           return(1);
58       };
         return(0);
60     }
       else
62       return(1);
     else
64     return(0);
   };
66
   int  lit_fact () {
68   if ( lit_po ())
       return( suivant() && lit_expr() && lit_pf() && suivant ());
70   else {
       erreur = 0;
72     if ( lit_ident ()) {
         synt_tree = cree_noeud(lexemes –> elt );
74       return( suivant ());
       }
76     else {
         erreur = 0;
78       return( lit_cste () && suivant ());
       };
80   };
   };
82
   int  lit_cste () {
84   if (lit_pm ()) {
       if ( suivant() && test_g ( cste )) {
86       synt_tree = cree_noeud(lexemes –> elt );
         return(1);
88     };
     }
90   else
       if ( test_g ( cste )){
92       synt_tree = cree_noeud(lexemes –> elt );
         return(1);
94     };
     erreur = 2;
96   return(0);
   };
```

PROGRAMME 23: *Déclaration des variables globales*

```
 lexliste  lexemes;
int  erreur ;
table  tabl_ident ;
arbre  synt_tree ;
```

PROGRAMME 24: *Traitement des erreurs*

```
void  traite_erreur  () {
  switch ( erreur ) {
    case 2 :
          printf ("Constante␣ou␣ identificateur ␣attendu .\n");
         break;
    case 3 :
          printf (")␣attendue .\n");
         break;
    case 4 :
          printf ("Fin␣de␣ligne␣attendue .\n");
         break;
    case 5 :
          printf ("  Identificateur  ␣attendu .\n");
         break;
    case 6 :
          printf ("Signe␣=␣attendu.\n");
         break;
    default  :
          printf ("Erreur␣non␣prevue.\n");
         break;
  };
};
```

fonction `main` comme prenant deux arguments (ligne 1) : l'un étant le nombre d'arguments et l'autre contenant un tableau de chaîne de caractère contenant chaque argument. On teste donc d'abord si l'utilisateur a bien spécifié un nom de fichier. Sinon, on affiche un message d'erreur et on arrête le programme. Il est à noter que le premier argument est le nom de l'exécutable lui-même. On affiche le nom du fichier et on teste si ce fichier est présent en tentant de l'ouvrir en lecture. S'il n'existe pas, on affiche un message d'erreur idoine.

Sinon et jusqu'à la fin du fichier, on lit une ligne de texte, on affiche la liste des lexèmes après analyse lexicale. Si `erreur` vaut 1, un caractère illégal a été trouvé. Sinon, on réalise l'analyse syntaxique de l'expression. Si elle est correcte, on continue après avoir affiché un message signalant le succès de l'analyse et donnant la valeur de l'expression. Sinon, on affiche la liste des lexèmes de l'expression à partir desquels il y a une erreur, puis on traite l'erreur. Lorsque la fin du fichier est atteinte, on ferme le fichier et on sort du programme.

4 Essais

Pour tester le module de gestion de la table des symboles, on écrit le programme décrit ci-après (cf. PROGRAMME 26). On utilise toutes les fonctions et procédures définies et on réalise des affichages successifs de la table pour contrôler les résultats. On commence par ajouter 20 symboles à la table et on en ajoute un autre pour tester l'allocation dynamique de la mémoire à la table. Puis, on teste la mise à jour, l'affectation de la valeur d'un symbole puis les recherches de référence et de valeur. Cela produit le résultat suivant (cf. FIGURE 6).

Pour tester les fonctions de manipulations d'arbres syntaxiques, on utilise le programme décrit ci-après (cf. PROGRAMME 27), qui va créer l'arbre correspondant à l'expression $12+3*6$ puis l'afficher. On observera qu'il est relativement fastidieux de créer un tel arbre à la main... Le résultat est consigné à la FIGURE 7.

Finalement, pour tester l'application, on utilise le fichier `validat1.don` fourni sur les machines de l'ECL. Ce fichier est présenté à la FIGURE 8 et le résultat de l'exécution à la figure FIGURE 9.

On observe d'une part que les analyses syntaxique et lexicale sont effectuées correctement et que l'évaluation est correcte. Ensuite, les messages d'erreur sont adaptés sans être forcément toujours clairs (« Fin de ligne attendue. » particulièrement). Enfin, le fait que les identificateurs possèdent la valeur 0, même lorsqu'ils n'ont jamais été affecté d'une valeur, peut poser des problèmes (évaluation de `toto = toto + lili` alors que `lili` a été mal déclaré).

PROGRAMME 25: *Fonction principale*

```
void main (int argc, char *argv []) {
  FILE * fichier ;
  char expr[255];
  char *nom_fic;

  if (argc == 1) {
     printf ("Nom de fichier manquant en parametre de ligne de commande.\n");
     exit (0);
  }
  else
     nom_fic = argv[1];

  printf (" Fichier : %s\n\n", nom_fic);
  if ((fichier = fopen(nom_fic, "r")) == NULL) {
     printf (" Fichier de test introuvable. Sortie du programme.\n");
     exit (0);
  };

  init ( tabl_ident );

  while (! feof( fichier )) {
     fgets (expr, 255, fichier );

     erreur = 0;
     synt_tree = NULL;
     lexemes = an_lex(expr, 0);
      affiche_liste (lexemes);

     if ( erreur )
        printf ("Caractere non attendu dans l'expression.\n");
     else {
        if (an_syn ()) {
           printf ("L'expression est correcte et vaut : ");
           printf ("%d\n", eval_arbre ( synt_tree ));
        }
        else {
           printf ("L'expression est incorrecte a partir de : ");
           affiche_liste (lexemes);
           traite_erreur ();
        };
     };
  };
  fclose ( fichier );
};
```

PROGRAMME 26: *Programme de test de la table*

```
#include <stdio.h>
#include "table.hpp"
#include "arbre.hpp"

int erreur;
lexliste lexemes;
table tabl_ident;
arbre synt_tree;

void main() {
  int i;
  table t;

  init(t);
  printf("%d␣%d\n", t.taille, t.libre);

  for(i=0; i<=19; i++)
    ajout(t, "toto");

  ajout(t, "titi");
  affiche_table(t);
  putchar('\n');

  printf("%d␣%d\n", maj(t, "titi"), maj(t, "tata"));
  set_val(t, 0, 1);  set_val(t, 20, 2);  set_val(t, 40, 3);
  affiche_table(t);
  putchar('\n');

  printf("%d␣%d\n", get_ref(t, "titi"), get_ref(t, "tutu"));
  printf("%d␣%d\n", get_val(t, 0), get_val(t, 20));
};
```

```
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
titi = 0

20 21
toto = 1
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
toto = 0
titi = 2
tata = 0

20 -1
1 2
```

FIGURE 6 – *Résultat du test de la table*

```
12 + 3 * 6
```

FIGURE 7 – *Résultat du programme de test des arbres*

```
   toto = 67
lili 45 * 8
   g=  3  +  7  * 9
toto = toto + lili
x= 67   toto
ec = toto = lili
4 + 7 *(5 -2)
y= 4 + 5*(toto   + toto)
lili = toto +4
```

FIGURE 8 – *Fichier* `validat1.don`

PROGRAMME 27: *Programme de test des arbres*

```
#include "arbre.hpp"
#include "table.hpp"

int erreur;
 lexliste lexemes;
table tabl_ident;
arbre synt_tree;

boite cree_boite(genre a, int b) {
  boite temp;
  temp.g = a;
  if (a=op)
    temp.operateur = b;
  else
    temp.val=b;
  return(temp);
};

void main() {
   synt_tree = enracine(cree_noeud(cree_boite(cste,12)),
                        cree_boite(op,'+'),
                        enracine(cree_noeud(cree_boite(cste,3)),
                                 cree_boite(op, '*'),
                                 cree_noeud(cree_boite(cste,6))));

   affiche_arbre(synt_tree);
};
```

```
Fichier : validat1.don

toto = 67
L'expression est correcte et vaut : 67
lili 45 * 8
L'expression est incorrecte a partir de : 45 * 8
Signe = attendu.
g = 3 + 7 * 9
L'expression est correcte et vaut : 66
toto = toto + lili
L'expression est correcte et vaut : 67
x = 67 toto
L'expression est incorrecte a partir de : toto
Fin de ligne attendue.
ec = toto = lili
L'expression est incorrecte a partir de : = lili
Fin de ligne attendue.
4 + 7 * ( 5 - 2 )
L'expression est incorrecte a partir de : 4 + 7 * ( 5 - 2 )
Identificateur attendu.
y = 4 + 5 * ( toto + toto )
L'expression est correcte et vaut : 674
lili = toto + 4
L'expression est correcte et vaut : 71
```

FIGURE 9 – *Essai sur le fichier* `validat1.don`