Implémentation d'un interpréteur élémentaire de fonctions : Théorie et pratique

Michel Rigo

Version provisoire (1999–2000)

Mots-clé : grammaire, arbre d'analyse, interpréteur.

1 Introduction

Le but de cette note est de permettre à toute personne possédant des connaissances de base en programmation de réaliser sans trop de difficultés un interpréteur de formules. Ainsi, le professeur de mathématiques désirant faire fonctionner des programmes de sa conception pourra rendre ceux-ci plus conviviaux puisque fonctions et autres expressions mathématiques pourront être introduites telles quelles par l'utilisateur sans que ce dernier ait à modifier le code source. Expliquons notre propos. Si vous désirez introduire des expressions comme $sin(\pi)+1$ ou $2+3*6$ dans un programme, vous coderez naturellement celles-ci dans la mémoire de l'ordinateur comme des chaînes de caractères. Il est dès lors impossible d'obtenir l'évaluation d'expressions aussi simples, puisque la formule est stockée de la même manière qu'un mot. Ainsi 2 est stocké en mémoire comme le caractère 2 faisant partie de l'alphabet de la machine et non pas comme le nombre entier 2 ! Il est donc clair qu'aucun calcul n'est envisageable sans réaliser une analyse détaillée de la chaîne de caractères que l'on fournit au programme.

Imaginons à titre d'exemple disposer d'un programme permettant l'affichage de fonctions à l'écran. Sans un interpréteur, l'utilisateur ne pourra entrer lui-même de nouvelles fonctions sans changer le code source du programme et recompiler ce dernier. Les applications d'un interpréteur sont donc nombreuses et variées.

Cet article peut être divisé en quatre parties. La première partie contient une introduction sommaire aux notions de grammaire et d'arbre d'analyse. Une fois ces notions introduites, nous construisons, dans la deuxième partie, la grammaire associée aux expressions mathématiques. La troisième partie de cet article est plus pratique et développe notre interpréteur. Celui-ci doit analyser une expression (une chaîne de caractères) pour en identifier les différents composants et agir en conséquence. Cette analyse est bien évidemment basée sur la grammaire décrivant les expressions mathématiques. La dernière partie de l'article, intimement liée à la précédente, contient le code source d'une calculatrice élémentaire ainsi qu'une application illustrant l'algorithme d'analyse. Pour conclure, nous donnons quelques suggestions pour compléter la calculatrice élémentaire et obtenir un interpréteur de fonctions complet (ajout d'autres fonctions, détection de formules inexactes, ...).

2 Grammaires et arbres d'analyse

Le but de cette section est de définir les concepts de grammaire hors-contexte, d'arbre d'analyse et de grammaire non-ambiguë. Nous illustrons ces concepts au moyen d'exemples simples.

Définition 1 Un *alphabet* est un ensemble fini. Ses éléments sont appelés symboles ou caractères. Ainsi $\Sigma = \{A, \ldots, Z\}$, $X = \{0, 1, \ldots, 9\}$ et $\Omega = \{+, -, *, /\}$ sont des alphabets. Si Σ est un alphabet, on note Σ^*, l'ensemble des mots sur Σ.

Définition 2 Une *grammaire hors-contexte* $G = (N, \Sigma, R, S)$ est la donnée de deux alphabets disjoints N et Σ, d'un ensemble fini R de règles, $R \subset N \times (N \cup \Sigma)^*$ et d'un symbole privilégié $S \in N$. Les éléments de N sont les symboles non terminaux et les éléments de Σ sont les symboles terminaux. On note $T \to u$ si $(T, u) \in R$.

Exemple 1 1. Voici un premier exemple de grammaire dont les alphabets sont

$$N = \{S, T\}\ ,\ \Sigma = \{a, b\}$$

et dont les règles sont données par $R = \{(S, TT),\ (T, Tb),\ (T, aT),\ (T, b)\}$, ce que l'on note généralement

$$S \to TT,\ T \to Tb,\ T \to aT,\ T \to b.$$

ou de manière plus concise par

$$S \to TT,\ T \to Tb \,|\, aT \,|\, b.$$

2. Un autre exemple de grammaire est fourni par $N = \{S\}$, $\Sigma = \{a, b\}$ et les règles

$$S \to aSb|\varepsilon$$

oε représente le mot vide (i.e., le mot ne contenant aucune lettre).

3. Ce dernier exemple de grammaire peut être vu comme la première étape de notre discussion. Les alphabets sont

$$N = \{S, T\}\ ,\ \Sigma = \{0, 1, \ldots, 9, +, -, *, /\}$$

et les règles de la grammaire sont

$$S \to T + T \,|\, T - T \,|\, T * T \,|\, T/T$$

$$T \to 0 \,|\, 1 \,|\, \ldots \,|\, 9 \,|\, 0T \,|\, 1T \,|\, \ldots \,|\, 9T.$$

Nous venons de définir les grammaires hors-contexte mais quelle est leur utilité ? Une grammaire est un moyen simple de décrire des ensembles de mots (que l'on appelle naturellement langages). Les mots décrits par la grammaire sont les mots formés sur l'alphabet Σ des symboles terminaux et qui sont obtenus à partir du symbole privilégié S en appliquant un nombre fini de fois des règles de la grammaire (on peut appliquer à plusieurs reprises la même règle). Ainsi, la grammaire (1) génère le mot *bbab*. En effet, on a successivement

$$\mathbf{S} \to \mathbf{T}T \to Tb\mathbf{T} \to \mathbf{T}baT \to bba\mathbf{T} \to bbab$$

à chaque étape, on remplace un non terminal S ou T (ici en gras) au moyen d'une des règles spécifiées. On parle de grammaire hors-contexte car le remplacement d'un non-terminal ne dépend pas des lettres entourant ce non-terminal, i.e., du contexte. Si un ensemble de mots est généré par une grammaire hors-contexte, on parle naturellement de *langage hors-contexte*.

Ainsi le langage formé des mots commençant par des a et se terminant par le même nombre de b est hors-contexte. Car il est clair que ce langage est généré par la grammaire (2).

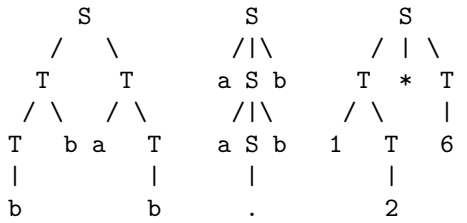
Regardons à présent la grammaire (3). On s'aperçoit aisément que les mots générés par cette grammaire sont exactement les expressions faisant intervenir uniquement des nombres naturels (commençant éventuellement par des zéros de tête) et une opération élémentaire comme + ou $*$. Ainsi $12*6$ est généré par la grammaire mais $2 - 5 * 4$ ne l'est pas. Notre but dans la section suivante sera de définir une grammaire générant l'ensemble des expressions mathématiques.

On suppose connue la notion d'arbre (pointé).

Définition 3 L'ensemble des *arbres d'analyse* d'une grammaire est défini récursivement de la façon suivante. Les arbres ne possédant qu'un seul élément sont les arbres de racine $a \in N \cup \Sigma$. Si $A_1, \ldots, A_m$ sont des arbres d'analyse de racines respectives $X_1, \ldots, X_m$ et si $X \to X_1 \ldots X_m$ est une règle de la grammaire, alors l'arbre de racine X et de sous-arbres $A_1, \ldots, A_m$ est encore un arbre d'analyse. On suppose implicitement que les sous-arbres sont ordonnés de gauche à droite.

Au mot *bbab* généré par la grammaire (1), on peut associer un arbre d'analyse. Construisons d'abord un arbre A_1 ayant uniquement la racine S. Ensuite on applique la règle $S \to TT$. L'arbre réduit à la racine T est aussi un arbre d'analyse. Donc si on le considère comme sous-arbre de l'arbre A_1, on obtient un arbre A_2 de racine S et possédant deux sous-arbres réduits au noeud T. Ensuite, on applique la règle $T \to Tb$. On fait donc pendre au premier noeud T deux nouveaux noeuds T et b, et ainsi de suite.

On a représenté schématiquement ci-dessous des arbres d'analyse de *bbab* pour la grammaire (1), de *aabb* pour la grammaire (2) et de $12 * 6$ pour la grammaire (3).



On remarque que les extrémités de l'arbre (appelées feuilles) sont constituées uniquement de symboles terminaux et la lecture récursive de gauche à droite des feuilles donne bien le mot *bbab* pour le premier arbre, *aabb* pour le deuxième et $12 * 6$ pour le dernier.

Définition 4 Une grammaire est *non-ambiguë* si pour chaque mot w généré par la grammaire (formé uniquement de symboles terminaux) il existe un seul arbre d'analyse pour w.

Nos préoccupations ont clairement un caractère pratique : la rédaction d'un programme déterministe analysant des expressions. Il nous paraît dès lors évident que le caractère non-ambigu des grammaires est indispensable, le programme ne peut pas disposer de plusieurs arbres d'analyse pour une même expression. En effet, des arbres différents peuvent donner des interprétations différentes et donc des résultats différents.

3 La grammaire des expressions

Le but de la présente section est de fournir une grammaire non-ambiguë générant exactement les expressions mathématiques. De cette façon, à chaque expression, on peut associer un arbre d'analyse, généralement appelé *arbre d'expression*. La lecture de cet arbre dans un ordre convenable assurera alors l'obtention du bon résultat (i.e., de la valeur de l'expression envisagée).

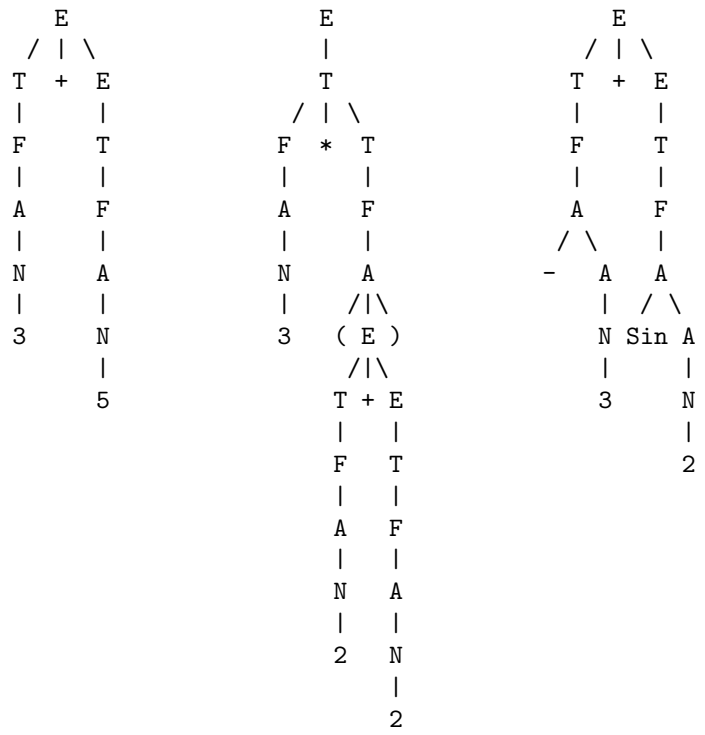
Pour construire une telle grammaire, il faut garder à l'esprit les règles de priorité des opérations. Il nous paraît plus simple de donner dès à présent les règles de la grammaire générant une expression mathématique quelconque. Pour ne pas charger inutilement les règles et par conséquent le code source correspondant, nous ne considérons ici que la fonction *sinus*, les autres fonctions pouvant être aisément ajoutées.

$$\begin{aligned} Expression &\rightarrow Terme + Expression \,|\, Terme - Expression \,|\, Terme \\ Terme &\rightarrow Facteur * Terme \,|\, Facteur / Terme \,|\, Facteur \\ Facteur &\rightarrow Atome^{Facteur} \,|\, Atome \\ Atome &\rightarrow Nombre \,|\, (Expression) \,|\, -Atome \,|\, Sin(Atome) \end{aligned}$$

Comme nous l'avons vu dans la définition 2, chaque grammaire possède un symbole privilégié. Ce symbole est ici *Expression*. Pour être complet, nous devons encore ajouter les règles générant les nombres. Pour des raisons de facilité, on s'autorise à écrire des nombres débutant par zéro.

$$\begin{aligned} Nombre &\rightarrow 0Nombre \,|1Nombre\,| \ldots |\, 9Nombre \,|\, , Decimal|0|1|\ldots|9 \\ Decimal &\rightarrow 0Decimal \,|1Decimal\,| \ldots |9Decimal \,|1|\ldots|9 \end{aligned}$$

On remarque que la hiérarchie des opérations entraîne une répartition des règles de la grammaire suivant des "couches" successives. Pour convaincre le lecteur que cette grammaire génère exactement les expressions mathématiques, construisons les arbres associés aux expressions $3 + 5$, $3 * (2 + 1)$ et $-3 + Sin(2)$. Pour simplifier les notations, *Expression* est remplacé par E, *Facteur* par F, ...



Nous pouvons faire une première remarque quant à l'évaluation d'une expression mise sous forme d'arbre d'analyse. Dans toutes les situations, pour effectuer une opération binaire comme par exemple +, il faut d'abord évaluer le sous-arbre de gauche puis celui de droite avant d'effectuer l'opération. Le cas des atomes est particulier puisque là, il n'y a qu'un sous-arbre à évaluer. On note donc encore la présence d'un algorithme récursif typique à la structure d'arbre. La démonstration du caractère non ambigu de la grammaire sort du cadre de cet article. Le lecteur peut cependant s'en convaincre aisément. Rappelons que le caractère non ambigu est nécessaire pour construire un algorithme d'analyse ! A chaque expression correspond un seul arbre, donc une seule évaluation possible.

4 Analyse d'une expression

La fin de la section précédente nous éclaire déjà sur la démarche à suivre pour analyser une expression. Nous allons ici procéder en détails à l'évaluation d'une formule. Pour une bonne compréhension de cette section, nous conseillons au lecteur de parcourir le code source de la section suivante et si besoin est, d'appliquer ce programme à l'un ou l'autre exemple simple. En regardant les arbres d'analyse introduits précédemment et en s'inspirant des remarques dégagées, nous voyons déjà une manière de procéder. La méthode proposée est récursive : en démarrant à la racine, évaluer le sous-arbre de gauche puis celui de droite et effectuer l'opération indiquée. Chaque sous-arbre étant lui-même un arbre, on applique la procédure jusqu'à arriver aux feuilles terminales qu'il n'est plus nécessaire d'évaluer.

Dans la suite, nous allons utiliser les notations de l'algorithme décrit à la section suivante pour permettre une compréhension plus simple de ce dernier. Ainsi, $formule$ représente la chaîne de caractères contenant l'expression à évaluer et Pos est un entier contenant le nombre de caractères déjà lus dans $formule$.

La stratégie à adopter est la suivante : on lit la chaîne de gauche à droite en se référant à la grammaire des expressions mathématiques et à chaque étape, on lit le caractère suivant (éventuellement un groupe de caractères) pour connaître la prochaine opération à effectuer. Cette action à réaliser est conservée dans la variable *Action*. On dispose d'autant de sous-programmes que de règles dans la grammaire. Pour des raisons de facilité, ces sous-programmes s'appellent *Expression*, $Terme$, ... , *Atome*. Si on se rappelle les règles de la grammaire, on voit que le sous-programme *Expression* doit faire appel au sous-programme $Terme$ mais également à lui-même. De la même manière, $Terme$ fait appel à $Facteur$ et à lui-même. On remarque que *Atome* peut faire appel à *Expression* et à lui-même. Notre programme possède donc une structure récursive.

Notre analyseur doit être capable de lire des blocs simples comme un nombre, un opérateur ,... Il faut donc en plus de faire une lecture caractère par caractère, être capable de donner un sens à une suite cohérente de lettres comme *sin* ou de chiffres comme 154.5. Cette distinction des éléments fondamentaux est souvent appelée "tokenisation[1]". Cette tâche est réalisée par les sous-programmes *LireReel* et *LireSymbole*. Voyons comment procéder. Si on fait appel à la procédure *LireSymbole*, celle-ci lit le contenu de $formule$ à la position Pos. Si le contenu est un opérateur comme +, la valeur de *Action* passe à la valeur correspondant à l'opérateur, ici *plus*. Si le contenu est une lettre, il faut continuer à lire les lettres suivantes jusqu'à obtenir autre chose qu'une lettre. On doit concaténer les différentes lettres lues pour obtenir un mot. On regarde alors le mot obtenu et on agit en conséquence en modifiant la valeur de *Action*. Si le contenu est à présent un chiffre, on fait alors appel à la procédure *LireReel*. On procède à peu près de la même manière qu'avec des lettres, on lit les chiffres un à un et on calcule la valeur du nombre obtenu. Rappelons que placer un chiffre c à la droite d'un nombre x ne contenant pas de partie décimale revient à multiplier ce nombre par 10 et à ajouter c. On aura dès lors une affectation de la forme $x \leftarrow 10 * x + c$. Si le nombre contient une partie décimale, ajouter un chiffre c à droite revient à ajouter $c/10^n$ pour un n convenable dépendant de la position, donc de la partie déjà lue. La lecture d'un nombre doit modifier la valeur de *Action* pour spécifier qu'un nombre vient d'être lu et ce nombre est stocké dans la variable $Valeur$.

Au lancement de notre programme, $formule$ doit contenir la chaîne de caractères à traiter et Pos doit pointer sur le premier caractère de la formule. On fait alors appel une première fois au sous-programme *LireSymbole*. De cette manière *Action* a pour contenu le type du premier bloc élémentaire lu. On fait à présent appel à *Expression* pour l'évaluation. Ce sous-programme fait appel à $Terme$ qui lui-même fait directement appel à $Facteur$, ce dernier faisant appel à *Atome* (nous rappelons que ce découpage est nécessaire pour respecter l'ordre de priorité des différentes opérations). Dans le sous-programme *Atome*, on agit suivant la valeur de *Action*. Si l'action décrite par cette variable tombe dans la catégorie définie par la grammaire, on agit en conséquence et dans ce cas, on appelle directement *LireSymbole* pour aller lire le symbole suivant (dès lors, *Action* aura été réactualisé pour la prochaine action à effectuer). Si l'action décrite par *Action* n'est pas dans cette catégorie, on ne fait rien et on quitte ce sous-programme (c'est que l'opération à effectuer est d'un niveau de priorité plus faible). Cette manière de procéder est utilisée à tous les niveaux. Par exemple, si on examine la procédure $Terme$, on retrouve la même philosophie : quand la variable *Action* a pour valeur $fois$ ou *divi*, on appelle la procédure *LireSymbole* et on agit en conséquence en effectuant une multiplication ou une division (ce qui nécessite un nouvel appel à $Terme$). Sinon on quitte le sous-programme et on retourne à un niveau hiérarchique moins élevé. Car si on se trouve dans la procédure $Terme$, c'est que l'on a quitté les procédures *Atome* et $Facteur$ donc *Action* ne peut plus contenir que *plus*, *moins* ou éventuellement prendre la valeur fin pour spécifier que toute la formule a été lue et qu'il n'y a donc plus rien à faire.

Nous espérons que ces quelques explications seront suffisantes à la compréhension du code source présenté ci-après. Vous trouverez de plus après ce code un exemple sur une expression simple reprenant les différents appels aux sous-programmes et le contenu des différentes variables.

5 Code source

Nous avons choisi d'illustrer l'algorithme d'analyse par un programme rédigé en C simulant une calculatrice. L'utilisateur entre les expressions de son choix à l'aide du clavier. Le programme fournit alors la valeur numérique de cette expression. Par convention, on quitte le programme en introduisant le mot "exit". De même, on représente une exponentiation a^b par la chaîne `a^b`. Le programme présenté ici peut aisément être adapté à un autre langage de programmation comme le Pascal. Nous avons choisi le langage C pour son code clair et concis.

Les différentes fonctions *Expression*, *Terme*, ... font appel les unes aux autres de manière emboitée. Il est donc naturel (et même souvent indispensable) de définir la fonction *Atome* avant la fonction *Facteur* qui doit pouvoir l'appeler. De même *Facteur* doit être défini avant *Terme*. Dès lors, *Expression* est défini en dernier. Mais *Atome* peut éventuellement faire appel à *Expression* défini par après. C'est pour cette raison que l'on spécifie l'en-tête de la fonction *Expression* en début de programme et que la fonction proprement dite est définie plus loin dans le code.

```
#include <stdio.h>
```

[1] Le terme anglais *token* pourrait se traduire ici par élément de base.

```
#include <string.h>
#include <math.h>

enum {nomb,plus,moin,fois,divi,parg,pard,expo,fsin,fin,erreur} Action;
char Formule[80];
double Valeur;
int Pos;

double Expression();
/* ================================================================ */
char LireCaractere()
{
if (Pos<strlen(Formule)-1) return Formule[++Pos];
else {Pos++; return ' ';}
}
/* ================================================================ */
double LireReel()
{
double x=0,d=10;
char c=Formule[Pos];
while ('0'<=c&&c<='9')
 {
  x=10*x+c-'0';
  c=LireCaractere();
 }
if (c=='.'||c==',')
 {
 c=LireCaractere();
 while ('0'<=c&&c<='9')
  {
   x+=(c-'0')/d;
   d*=10;
   c=LireCaractere();
  }
 }
return x;
}
/* ================================================================ */
void LireSymbole()
{
char Bloc[80]="\0",d[2];
while (Formule[Pos]==' ') Pos++;
if (Pos>=strlen(Formule)) Action=fin;
else
 {
 d[0]=Formule[Pos];
 d[1]='\0';
 if (('0'<=d[0]&&d[0]<='9')||d[0]=='.') {Valeur=LireReel();
Action=nomb;}
 else
  {if (('a'<=d[0]&&d[0]<='z')||('A'<=d[0]&&d[0]<='Z'))
   {int OK=1;
    while
(Pos<strlen(Formule)&&(('a'<=d[0]&&d[0]<='z')||('A'<=d[0]&&d[0]<='Z')))

{strcat(Bloc,d); Pos++; if (Pos<strlen(Formule)) d[0]=Formule[Pos];}
    if (strcasecmp(Bloc,"pi")==0) {Action=nomb; Valeur=3.141592654;
```

```
OK=0;}
    if (strcasecmp(Bloc,"e")==0) {Action=nomb; Valeur=2.7182818285;
OK=0;}
    if (strcasecmp(Bloc,"sin")==0) {Action=fsin; OK=0;}
    if (OK==1) Action=erreur;
   }
   else {Pos++;
    switch (d[0])
    {case '*' : Action=fois; break;
     case '+' : Action=plus; break;
     case '-' : Action=moin; break;
     case '/' : Action=divi; break;
     case '(' : Action=parg; break;
     case ')' : Action=pard; break;
     case '^' : Action=expo; break;
     default : Action=erreur;
    }
   if (Pos>=strlen(Formule)&&Action!=pard) Action=erreur;
   }
  }
 }
}
/* ================================================================ */
double Atome()
{
double Inter;
switch (Action)
 {case nomb : LireSymbole(); Inter=Valeur; break;
  case parg : LireSymbole(); Inter=Expression();
              if (Action!=pard) {Action=erreur; Inter=1;}
              else LireSymbole();
              break;
  case moin : LireSymbole(); Inter=-Atome(); break;
  case fsin : LireSymbole(); Inter=sin(Atome()); break;}
return Inter;
}
/* ================================================================ */
double Facteur()
{
double Inter=Atome();
int sortie=0;
while (sortie==0)
 switch (Action)
  {case expo : LireSymbole();
               if (Inter>0) {Inter=exp(log(Inter)*Facteur());}
               else {Action=erreur; Inter=1;}
               break;
   default : sortie=1;}
return Inter;
}
/* ================================================================ */
double Terme()
{
double Inter=Facteur(),Divi;
int sortie=0;
while (sortie==0)
 switch (Action)
```

```
  {case fois : LireSymbole(); Inter*=Terme(); break;
   case divi : LireSymbole(); Divi=Terme();
               if (Divi==0) {Action=erreur; Inter=1;}
               else Inter/=Divi; break;
   default : sortie=1;}
return Inter;
}
/* ================================================================ */
double Expression()
{
double Inter=Terme();
int sortie=0;
while (sortie==0)
 switch (Action)
  {case plus : LireSymbole(); Inter+=Expression(); break;
   case moin : LireSymbole(); Inter-=Expression(); break;
   default : sortie=1;}
return Inter;
}
/* ================================================================ */
main()
{
 double resultat;
 int compteur=1;
 printf("%d --> ",compteur++);
 gets(Formule);
 while (strcasecmp(Formule,"exit")!=0)
 {
  Pos=0;
  LireSymbole();
  resultat=Expression();
  if (Action==erreur) printf("La formule propos\'ee est erron\'ee ! \n");
  else printf(" = %f \n",resultat);
  printf("%d --> ",compteur++);
  gets(Formule);
 }
}
```

Exécution du programme avec $formule =$"`3*sin(2)`",

		$Symbole$	$Valeur$
$executionLireSymbole$		$Nombre$	
$executionLireReel$		$Nombre$	3
$appel\ Expression$		$Nombre$	3
$appel\ Terme(1)$		$Nombre$	3
$appel\ Facteur$		$Nombre$	3
$appel\ Atome$		$Nombre$	3
$execution\ LireSymbole$		$*$	3
$fin\ Atome$	$\rightarrow 3$	$*$	3
$fin\ Facteur$	$\rightarrow 3$	$*$	3
$retour\ Terme(1)$		$*$	3
$execution\ LireSymbole$		$fsin$	3
$appel\ Terme(2)$		$fsin$	3
$appel\ Facteur$		$fsin$	3
$appel\ Atome(1)$		$fsin$	3
$execution\ LireSymbole$		$Nombre$	3
$execution\ LireReel$		$Nombre$	2
$appel\ Atome(2)$		$Nombre$	2
$execution\ LireSymbole$		fin	2
$fin\ Atome(2)$	$\rightarrow 2$	fin	2
$fin\ Atome(1)$	$\rightarrow sin(2)$	fin	2
$fin\ Facteur$	$\rightarrow sin(2)$	fin	2
$fin\ Terme(2)$	$\rightarrow 3 * sin(2)$	fin	2
$fin\ Terme(1)$	$\rightarrow 3 * sin(2)$	fin	2
$fin\ Expression$	$\rightarrow 3 * sin(2)$	fin	2

6 Quelques compléments

L'interpréteur présenté dans les sections précédentes peut être amélioré de différentes façons. Certaines de ces améliorations ont été effectivement implémentées, d'autres pas.

- Le problème des majuscules et des minuscules : en général, la chaîne de caractères "`SIN`" est différente de "`sin`". Rien n'empêche l'utilisateur d'utiliser des majuscules et il faut dès lors tenir compte de cette contrainte lors de la rédaction de notre interpréteur. Dans le code source, on a modifié le sous-programme *LireSymbole* en conséquence. L'implémentation réalisée ici[2] utilise l'instruction `strcasecmp` qui compare deux chaînes de caractères sans tenir compte de la casse. On aurait également pu construire un sous-programme transformant une chaîne de caractères en une autre dans laquelle il n'y aurait que des minuscules (certains langages proposent d'ailleurs de telles fonctions comme par exemple en C, `strlwrcase`).

- Détection des formules erronées. Ici, lorsque le programme rencontre une formule inexacte, le sous-programme dans lequel l'erreur s'est produite modifie la valeur d'*Action* à *erreur* (cf. exponentiation incorrecte, division par zéro, problème de parenthèsage, erreur de syntaxe). Cette partie pourrait être améliorée en précisant explicitement à l'utilisateur quel type d'erreur il a commis.

- L'ajout d'autres fonctions *cos*, *ln*, ... est très simple à réaliser. Pour ne pas allonger le code nous les avons évitées. Les modifications à apporter sont les suivantes. La variable *Action* peut prendre de nouvelles valeurs qu'il faut donc ajouter dans l'énumération

  ```
  enum {nomb,plus,...,fsin,fcos,fln,...} Action;
  ```

  Dans le sous-programme *LireSymbole* qui permet d'identifier les "tokens", il faut ajouter des lignes de la forme

[2] sous Linux et le compilateur gcc.

```
if (strcasecmp(Bloc,"cos")==0) {Action=fcos; OK=0;}
if (strcasecmp(Bloc,"ln")==0) {Action=fln; OK=0;}
```

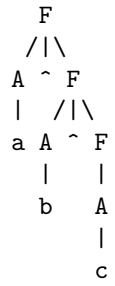
Bien évidemment les règles de la grammaire se compliquent. On a

$$Atome \rightarrow Nombre \,|\, (Expression) \,|\, -Atome \,|\, Sin(Atome) |\, Cos(Atome) |\, ln(Atome)$$

et le sous-programme *Atome* s'en trouve naturellement modifié,

```
case fcos : LireSymbole(); Inter=cos(Atome()); break;
case fln : LireSymbole(); Inter=log(Atome()); break;
```

- Les constantes π, e ont été définies. On pourrait également définir d'autres constantes suivant les besoins.
- Suivant le langage de programmation utilisé, il se peut que certaines fonctions ne soient pas définies. Une application du théorème de l'ouvert connexe suffit souvent pour remédier à ce problème. Voici un exemple, il est peut probable que la fonction $arcsh(x)$ soit définie dans votre langage de programmation. Par contre $ln(x)$ et $\sqrt{x}$ sont certainement définis, il suffit dès lors de se rappeler que
$$arcsh(x) = ln(x + \sqrt{x^2 + 1})$$
pour implémenter sans difficulté cette fonction.
- L'exponentiation peut aussi être améliorée. En effet, dans notre programme, il n'est pas possible de prendre une puissance entière d'un nombre négatif.
- La dernière remarque constitue plutôt une mise en garde sur les exponentiations multiples. La stratégie adoptée par notre interpréteur est de lire les formules de gauche à droite. Si nous regardons une partie de l'arbre d'expression associé à la formule `a^b^c`, nous avons



Dès lors la chaîne `a^b^c` est interprétée comme $a^{(b^c)}$ et non pas comme a^{b^c}. Pour obtenir l'évaluation de a^{b^c}, il faut recourir au parenthèsage et introduire la formule `(a^b)^c`.

7 Bibliographie

[1] A. Aho, J. Ullman, *Concepts fondamentaux de l'informatique*, Dunod, Paris, 1993.

[2] D. E. Knuth, *The art of computer programming. Vol.1: Fundamental algorithms*, Addison-Wesley, London, 1973.

[3] P. Lecomte, *Théorie des algorithmes*, note de cours en licence en sciences mathématiques, Université de Liège, 1993-1994.

[4] J. Schmets, *Analyse mathématique*, note de cours en première candidature en sciences mathématiques et en sciences physiques, Ed. Derouaux, 1993.

[5] P. Wolper, *Introduction à la calculabilité*, InterEditions, Paris, 1991.