Analyse et évaluation d'expressions arithmétiques

L'objet de ce problème est l'écriture d'un programme qui lit une expression arithmétique, vérifie sa syntaxe et, si la syntaxe est correcte, donne la valeur de l'expression. Exemple (les interventions de l'utilisateur sont soulignées) :

```
A toi : 2 * 2 =
la syntaxe de l'expression est correcte
sa valeur est 4
A toi : (2 + 3) * (10 - 2) - 12 * (1000 + 15) =
la syntaxe de l'expression est correcte
sa valeur est -12140
A toi : 2 ++ 3 =
la syntaxe de l'expression est erronée
A toi : .
Au revoir...
```

DEFINITION DE LA SYNTAXE. La syntaxe des expressions arithmétiques qui nous intéressent sera définie par un ensemble de formules constituant ce qu'on appelle une grammaire *BNF* (*Backus-Naur form*). On dit aussi grammaire *non contextuelle* ou, en bon français, *context free*.

Une telle grammaire est définie par la donnée de :
- un ensemble de *symboles non terminaux*, qui sont les catégories grammaticales servant à décrire la grammaire ;
- un ensemble de *symboles terminaux*, qui sont des symboles devant apparaître tels quels dans les textes analysés ;
- un symbole de départ, qui est la catégorie la plus générale ;
- un ensemble de *règles de dérivation* (on dit aussi de *réécriture*).

Dans notre grammaire, donnée ci-dessous, les symboles non terminaux sont écrits en caractères penchés : *expression*, *terme*, etc. Les symboles terminaux sont écrits entre apostrophes, comme '+' (tous les symboles terminaux sont de simples caractères). Le symbole de départ est *expression*.

Nous utilisons les deux méta-symboles → et |. Le premier est le *méta-symbole fondamental de dérivation* : une règle de grammaire comme

$$s_0 \rightarrow s_1\ s_2\ \ldots\ s_n$$

signifie « s_0 se réécrit en $s_1 \ldots s_n$ », ce qui peut se comprendre de deux manières :
- point de vue de la synthèse (production de tous les textes corrects) : « pour écrire un s_0, il faut écrire un s_1, suivi d'un s_2, … suivi d'un s_k » ;
- point de vue de l'analyse (reconnaissance qu'un texte donné est correct) : « pour reconnaître un s_0, il faut reconnaître un s_1 suivi d'un s_2 … suivi d'un s_n ».

Le méta-symbole | exprime une disjonction. La règle « $s_0 \rightarrow s_1 \ldots s_n \mid t_1 \ldots t_m$ » équivaut aux deux règles « $s_0 \rightarrow s_1 \ldots s_n$ » et « $t_1 \ldots t_m$ », qui indiquent deux manières de dériver le symbole s_0.

Voici la grammaire des expressions arithmétiques qui nous intéressent.

expression	→	*terme opérateur-additif expression* | *terme*
terme	→	*facteur opérateur-multiplicatif terme* | *facteur*
facteur	→	*nombre* | '(' *expression* ')'
nombre	→	*chiffre nombre* | *chiffre*
chiffre	→	'0' | '1' | … | '9'
opérateur-additif	→	'+' | '–'
opérateur-multiplicatif	→	'*' | '/'

Une expression est correcte si on peut l'obtenir à partir du symbole de départ par une suite de dérivations. L'ensemble de telles expressions s'appelle le *langage* correspondant à la grammaire.

Par exemple, 2 + 3 est une expression arithmétique correcte, car

expression ⇒ *terme opérateur-additif expression* ⇒ *facteur opérateur-additif expression*
⇒ *nombre opérateur-additif expression* ⇒ *chiffre opérateur-additif expression*
⇒ '2' *opérateur-additif expression* ⇒ '2' '+' *expression* ⇒ '2' '+' *terme*
⇒ '2' '+' *facteur* ⇒ '2' '+' *nombre* ⇒ '2' '+' *chiffre* ⇒ '2' '+' '3'

Parlant familièrement, notre grammaire dit qu'une *expression* est une addition/soustraction de termes, qu'un *terme* est une multiplication/division de facteurs et qu'un *facteur* est soit un *nombre* entier, soit une expression entre parenthèses.

Il en découle (mais il faut réfléchir un petit moment) que les opérateurs multiplicatifs ont la priorité sur les opérateurs additifs, et que les parenthèses servent à contourner cette priorité.

ANALYSEUR SYNTAXIQUE. Le travail de programmation qui vous est demandé sera décomposé en deux temps. Tout d'abord, vous écrirez un *analyseur syntaxique* utilisant la grammaire ci-dessus. Un analyseur syntaxique est un programme qui lit un texte et répond à la question « ce texte est-il correct pour la grammaire donnée ? », c'est-à-dire « ce texte peut-il être obtenu à partir du symbole de départ en appliquant les règles de réécriture de la grammaire ? ». Dans un deuxième temps, vous augmenterez l'analyseur précédent afin qu'en plus de vérifier la syntaxe, il obtienne une information extraite de l'expression analysée : la valeur de cette dernière.

Un analyseur syntaxique se déduit de la grammaire *BNF* correspondante par une démarche très systématique qu'on peut résumer de la manière suivante :

1. L'analyseur doit lire le texte analysé, du début vers la fin, *caractère par caractère*. A tout instant, un seul caractère du texte se trouve en mémoire, dans une variable globale nommée, par exemple, *calu* (« caractère lu »). Veillez à maintenir constamment vraie la propriété suivante : *la valeur de calu est le premier caractère non encore examiné*.

2. On doit écrire une fonction pour chaque non-terminal de la grammaire. La fonction associée au non-terminal *S* s'appellera *reconnaître-S* ou tout simplement *S* (nous aurons ainsi la fonction *expression*, la fonction *terme*, la fonction *facteur*, etc.) Le rôle de la fonction *S* est de déterminer si un texte donné constitue une réécriture correcte du non-terminal *S*, elle est donc l'analyseur correspondant à la sous-grammaire dont *S* est le symbole de départ.

3. Le corps de la fonction *S* se déduit du membre droit de la règle ayant le non-terminal *S* pour membre gauche. Chaque non-terminal apparaissant dans le membre droit donnera lieu à l'appel de la fonction correspondante. Par exemple, si la règle est $S \rightarrow s_0\, s_1 \ldots s_k$ le corps de la fonction *S* sera « appeler s_0, appeler s_1, … appeler s_k » c'est-à-dire en C, `{` s_0`();` s_1`();` … s_k`();` `}`

4. L'apparition d'un symbole terminal dans le membre droit d'une règle se traduit simplement par la vérification du fait que le symbole courant (*calu*) coïncide avec ce terminal. Si ce n'est pas le cas, on annonce une erreur et, pour ce qui concerne l'analyseur, on stoppe le travail.

5. Lorsqu'une règle se réécrit de plusieurs manières, c'est-à-dire lorsqu'une disjonction | apparaît au membre droit, la valeur du symbole courant (*calu*) permet de choisir, le moment venu, parmi les réécritures possibles. Cela est une importante qualité des grammaires qui nous intéressent que n'ont pas des langages plus complexes, pour lesquels il faut écrire des analyseurs « avec backtracking ».

ÉVALUATION DE L'EXPRESSION. Elle se fera au fur et à mesure de l'analyse de la chaîne donnée : chacune des fonctions qui constituent l'analyseur rendra comme résultat la valeur de l'expression, terme, facteur, etc., qu'elle aura reconnu.

Il en découlera que, contrairement à l'habitude, entre opérateurs de même priorité, l'évaluation se fera de la droite vers la gauche. Par exemple, l'expression 3 + 5 * 125 / 7 - 6 + 10 sera évaluée comme si elle avait été écrite 3 + ((5 * (125 / 7)) - (6 + 10)), ce qui donnera la valeur 72.

Il est vivement recommandé d'écrire d'abord l'analyseur syntaxique, de s'assurer qu'il est correct, et seulement alors de lui ajouter le travail d'évaluation des expressions.

PROGRAMME. Votre programme doit permettre l'évaluation de plusieurs expressions arithmétiques. Chacune d'elles se terminera par le caractère '='. La liste des expressions elle-même sera suivie d'un point '.' qui joue le rôle de marque de fin de session. Voyez l'exemple au début de ce sujet.

Pour faciliter l'emploi de votre programme, vous permettrez que des caractères blancs (espaces, tabulations, fins de ligne) puissent être insérés à tout endroit, sauf à l'intérieur d'un nombre. Pratiquement, cela se manifestera par le fait que les valeurs de calu seront acquises à travers une fonction de « lecture améliorée », nommée par exemple *lire_utile()*, chargée de réitérer l'appel de *getchar()* jusqu'à l'obtention d'un caractère non blanc.

TRAITEMENT DES ERREURS. Dans une première version de votre programme, la rencontre d'une erreur de syntaxe dans une expression doit provoquer l'affichage d'un message et la terminaison de la session. S'il vous reste du temps, vous écrirez une deuxième version où les erreurs ne tuent pas le programme, mais uniquement l'évaluation en cours (comme dans l'exemple du début de ce sujet).

• • •