COMMENT COMPILER OU INTERPRÉTER UNE EXPRESSION ARITHMÉTIQUE

Raymond SEROUL

1. Le problème

Que se passe-t-il dans un ordinateur ou une calculatrice entre le moment où je tape sur le clavier

$$\sin(3 * x + \exp(1 + \cos(x * x))) + 3 * x * x - 7 * x + 1$$

et le moment où s'affiche le résultat ?
En entrée, l'ordinateur (ou la calculatrice) reçoit une *chaîne de caractères*. En sortie, j'obtiens la *valeur* de l'expression arithmétique que j'ai tapé au clavier.

Dans cet article, nous allons étudier les différentes étapes qui séparent l'entrée de la sortie. Et, chemin faisant, nous apprendrons à écrire un *interpréteur*, un *compilateur* d'expressions arithmétiques et à *produire* diverses sortes de code intermédiaire. Cela vous permettra de saisir directement au clavier une expression arithmétique ou une fonction sans être obligé de recompiler votre programme à chaque fois.

Le langage utilisé sera le PASCAL.

2. Expressions arithmétiques

2.1 — Par "expression arithmétique" nous entendrons ici une chaîne de caractères obtenue en concaténant :

- les lettres 'a' jusqu'à 'z';
- les symboles '$+$' et '$*$';
- les parenthèses '(' et ')'.

Nous avons appris petit à petit à discerner les "bonnes" expressions arithmétiques. Nous savons par exemple que $a * x * x + b * x + c$ est une "bonne" expression et que $a * x+)b$ n'est pas une "bonne" expression. Et cela, sans avoir jamais rencontré de définition précise. Vous allez comprendre pourquoi....

© L'OUVERT 59 (1990)

2.2 — Les "bonnes" expression arithmétiques se subdivisent en trois catégories : les *expressions* proprement dites, les *termes* et les *facteurs* dont voici la définition.

2.2.1 Nous noterons $\mathcal{E}$ (et nous appellerons *expression arithmétique*) une chaîne de caractères qui est un *terme* ou une *somme de termes*, c'est-à-dire une chaîne de caractères qui s'obtient en concaténant un terme, un signe '+', un autre terme, etc.

2.2.2 Nous noterons $\mathcal{T}$ (et nous appelerons *terme*) une chaîne de caractères qui est un *facteur* ou un *produit de facteurs*, c'est-à-dire une chaîne qui s'obtient en concaténant un facteur, un signe '$*$', un autre facteur, etc.

2.2.3 Enfin, nous noterons $\mathcal{F}$ (et nous appellerons *facteur*) une *variable* (c'est-à-dire une des lettres 'a', 'b',...,'z') ou une expression arithmétique parenthésée (c'est-à-dire la chaîne de caractères que l'on obtient en concaténant une parenthèse ouvrante, une expression arithmétique et une parenthèse fermante).

De manière imagée, nous pouvons exprimer les définitions précédentes par les "équations" :

$$\begin{array}{lll} \mathcal{E} = T & \text{ou} & \mathcal{E} = T + \cdots + T, \\ \mathcal{T} = F & \text{ou} & \mathcal{T} = F * \cdots * F, \\ \mathcal{F} = a & \text{ou} & \mathcal{F} = (E). \end{array}$$

Dans toute la suite, "expression" sera synonyme de "expression arithmétique".

REMARQUE : la définition que nous venons de donner laisse un certain malaise quand on la lit pour la première fois. Mais c'est le prix à payer avec les définitions récursives... Il est possible de donner une définition plus agréable d'une expression arithmétique, mais pour cela, il faut apprendre la théorie des grammaires. (La définition qui vient d'être donnée est d'ailleurs directement inspirée de cette théorie.)

EXEMPLE : la chaîne de caractères

$$a + (b + c * x) * u * v + e + f$$

est-elle une bonne expression arithmétique au sens de notre définition? Procédons petit à petit :

a) Si les chaînes a, $(b + c * x) * u * v$, e et f sont des termes, alors la chaîne donnée est une bonne expresssion arithmétique en vertu de (2.2.1).

b) En combinant (2.2.2) et (2.2.3), nous voyons tout de suite que les chaînes a, e et f sont des termes, car ce sont des facteurs.

c) La définition (2.2.2) nous indique que $(b + c * x) * u * v$ est un terme si nous savons prouver que $(b + c * x)$, u et v sont trois facteurs.

d) (Accélérons un peu.) Si la chaîne $b + c * x$ est une bonne expression, nous saurons que $(b + c * x)$ est un facteur en vertu de (2.2.3). Pour savoir si $b + c * x$ est une expression, il faut recommencer : si b et $c * x$ sont deux termes, alors $b + c * x$ est une expression. Mais b est un facteur, donc un terme, etc.

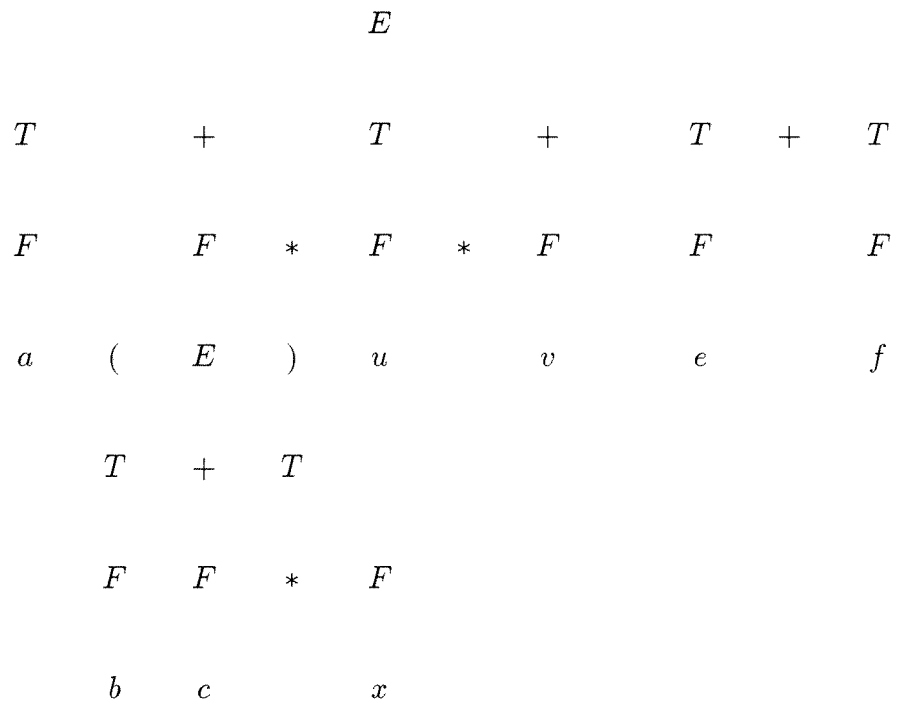


Figure 1

e) Enfin u et v sont deux facteurs.

Nous pouvons illustrer et résumer la démarche que nous venons de suivre sous la forme d'une arborescence (voir Figure 1).

REMARQUE : il importe de bien comprendre la portée des définitions (2.2.1), (2.2.2) et (2.2.3). Une chaîne de caractères est une bonne expression arithmétique s'il est possible de la décomposer en une somme de termes; une chaîne est un terme s'il est possible de la décomposer en un produit de facteurs, etc. Nous n'avons aucune protection contre les manières farfelues de fabriquer une expression, un terme ou un facteur. Par exemple, la concaténation des chaînes $a*($ et $b+c)$ est une expression. Mais nous savons que $a*(b+c)$ est une expression parce qu'il existe une décomposition "légale" en un produit de facteurs.

3. Reconnaître une expression arithmétique

3.1 — Nous allons écrire un programme PASCAL qui nous dira si une chaîne de caractères donnée est une bonne expression arithmétique. Nous aurons besoin des déclarations :

```
label
  999;
type
  str255 = string[255];
```

```
var
  expression : str255;
  token      : char;
  k          : integer;
```

La variable *expression* contient la chaîne à examiner et *token* contiendra un des caractères de cette chaîne. Pour éviter "d'arriver au bord" de *expression*, nous concatènerons la chaîne d'entrée avec le caractère '$'. Ce caractère sert d'indicateur de fin de chaîne; d'un point de vue technique, il nous assure de l'existence d'un caractère *après* l'expression qu'on cherche à reconnaître.

3.2 — La chaîne d'entrée n'est pas obligée d'être une bonne expression. Aussi avons-nous besoin d'une procédure *erreur*. La première erreur sera fatale, il n'y aura aucune tentative de traitement d'erreur. Pour simplifier, cette procédure n'envoie qu'un message lapidaire (sans diagnostic) :

```
procedure erreur ;
begin ;
    writeln('expression incorrecte') ;
    goto 999 ;
end ;
```

3.3 — La variable *token* sera successivement égale à tous les caractères de la chaîne *expression* (y compris le caractère '$' qui borde désormais cette chaîne). C'est la procédure *next_token* qui modifie *token*. Pour simplifier l'écriture des procédures que nous verrons plus loin, il est utile de demander à la procédure *next_token* de vérifier d'abord si la valeur courante de *token* est égale à un caractère qu'on lui transmet. Si c'est le cas, *token* change de valeur. Sinon, *next_token* tire la sonnette d'alarme :

```
procedure next_token(tok : char) ;
begin ;
    if tok <> token then erreur ;
    k := k + 1 ;
    token := expression[k] ;
end ;
```

Cas particulier : *next_token*(*token*) change la valeur de *token* sans jamais déclencher d'erreur.

3.4 — Avant de commencer à programmer, demandons-nous quels caractères peuvent commencer ou terminer une expression, un terme ou un facteur :

- une expression, un terme ou un facteur ne peuvent débuter que par une lettre (i.e. un nom de variable) ou une parenthèse ouvrante;
- une expression ne peut être suivie que par l'un des deux caractères ')' ou '$';
- un terme ne peut être suivi que par l'un des trois caractères '+', ')' ou '$';
- un facteur ne peut être suivi que par l'un des quatre caractères '+', '*', ')' ou '$'.

3.5 — Nous allons maintenant écrire trois procédures E, T et F qui ne font que traduire très fidèlement les définitions (2.2.1), (2.2.2) et (2.2.3). Le cahier des charges est le suivant :

- Les procédures E, T et F modifient la valeur de la variable *token*.
- La procédure E "lit" (c'est-à-dire passe successivement en revue) les caractères de la chaîne *expression* en partant du caractère *token* et avance dans la chaîne donnée jusqu'à la fin de la plus grande expression arithmétique qui commence avec *token*. Quand la procédure E s'arrête, la variable *token* doit être égale au premier caractère non examiné. De manière imagée, nous dirons que la procédure E *reconnaît* les expressions arithmétiques.
- De manière analogue, la procédure T reconnaît le plus grand terme de *expression* qui débute par *token*.
- Enfin, la procédure F reconnaît le plus grand facteur de *expression* qui débute par *token*.

3.6 — Comme chacune des procédures E, T ou F pourra appeler l'une des trois autres, nous devons utiliser la déclaration **forward** :

```
procedure E ; forward ;
procedure T ; forward ;
procedure F ; forward ;
```

3.7 — La procédure E commence par "consommer" un terme, puis consomme autant qu'elle le peut des chaînes de la forme $+\,T$. On examine alors la valeur de *token* : si cette variable est différente d'une parenthèse fermante (cas d'une sous-expression) ou d'un signe dollar (lorsque la chaîne à traiter a été entièrement lue), on déclenche une alarme :

```
procedure E ;
begin
   T ;
   while token = '+' do
   begin next_token(token) ; T end ;
   if not (token in [')', '$']) then erreur ;
end ;
```

3.8 — On procède de manière analogue pour la procédure T (l'alarme est donnée si le caractère qui suit le terme n'est pas le signe '+', une parenthèse fermante ou la fin de la chaîne) :

```
procedure T ;
begin
   F ;
   while token = '*' do
   begin next_token(token) ; F end ;
   if not (token in ['+', ')', '$']) then erreur ;
end ;
```

3.9 — La procédure F examine si le caractère courant de la chaîne d'entrée est un nom de variable. Si c'est le cas, elle le consomme (procédure *next_token*). Dans le cas contraire, elle vérifie qu'elle a affaire à une expression parenthésée. Enfin, si *token* n'est pas une lettre ou une parenthèse ouvrante, l'alarme est déclenchée :

```
procedure F ;
begin
   if token in ['a'..'z']
   then next_token(token)
   else begin next_token('(') ; E ; next_token(')') end ;
   if not (token in ['+', '*', ')', '$']) then erreur ;
end ;
```

3.10 — REMARQUES :

1) Une lecture superficielle suggère que seule la procédure F s'assure que le premier caractère de la sous-chaîne à traiter est correct. Mais le premier travail de la procédure E est d'appeler T, qui appelle F. Par conséquent, E et T s'assurent bien que la sous-chaîne à traiter commence par un caractère légal (lettre ou parenthèse ouvrante).

2) Chacune des procédures E, T et F teste si le caractère qui suit la sous-chaîne qu'elle traite est correct. On peut alléger ces tests en cascade. En effet, le test qui figure à la fin de la procédure T

if not (*token* **in** [')','$']) **then** *erreur*

est inutile, car il fait double emploi avec le test correspondant qu'effectue la procédure E appelant T. De même, le test

if not (*token* **in** ['+','*',')','$']) **then** *erreur*

qui figure à la fin de la procédure F est inutile (car T ou E s'en chargent). Ces tests seront donc supprimés dans les versions ultérieures des procédures T et F.

3.11 — Le corps du programme est minuscule : il consiste à entrer la chaîne *expression* qu'on borde ensuite avec le symbole '$', puis à initialiser les variables k et *token*. Ceci fait, la procédure E traite la chaîne. Par construction, E reconnaît la plus grande chaîne qui commence par *expression*[1]. Par exemple, si la chaîne d'entrée est $a + bc$, la procédure E reconnaît $a + b$ et *token* vaut c. Il est donc indispensable de s'assurer que la chaîne *expression* a été entièrement lue (d'où le test **if** *token* <>' $'**then** *erreur*) :

```
begin
   write('expression = ') ; read(expression) ;
   expression := concat(expression, '$') ;
   k := 1 ; token := expression[k] ;
   E ;
   if token <> '$' then erreur ;
   999 :
end.
```

3.12 — Nous comprenons maintenant beaucoup mieux la Figure 1 de la section précédente. L'arborescence associée à la chaîne $\omega = a + (b + c * x) * u * v + e + f$ n'est pas autre chose que *l'arbre des appels* du programme principal lorsqu'on lui fournit la chaîne ω. (Une branche telle que "F — a" ou "E — $+$" correspondant à l'appel de *next_token*.)

3.13 — EXERCICE : enrichir le programme précédent pour qu'il émette un diagnostic raisonnable en cas d'erreur.

4. Interpréter ou évaluer une expression arithmétique

Nous supposons maintenant que les variables a à z possèdent des valeurs mémorisées dans le tableau $variables['a'..'z']$.

4.1 — Pour *calculer* la valeur d'une expression, il suffit d'introduire une variable globale *valeur* que les procédure E, T, F mettent à jour. Plus précisément, après appel de $E(valeur)$, la variable *valeur* contiendra la valeur de l'expression reconnue par E. De même, après l'appel de $T(valeur)$ (resp. $F(valeur)$), la variable *valeur* contiendra la valeur du terme (resp. facteur) reconnu par T (resp. F). La déclaration de ces trois procédures est donc :

```
procedure E(var valeur : real) ; forward ;
procedure T(var valeur : real) ; forward ;
procedure F(var valeur : real) ; forward ;
```

4.2 — Si une expression se réduit à un terme, sa valeur est celle du terme. Si une expression est une somme de termes, sa valeur est la somme des valeurs des différents termes qu'on ajoute au fur et à mesure :

```
procedure E ;
var new_val : real ;
begin
   T(valeur) ;
   while token = '+' do begin
      next_token(token) ;
      T(new_val) ;
      valeur := valeur + new_val ;
      end ;
   if not (token in [')', '$']) then erreur ;
end ;
```

4.3 — La procédure T est calquée sur la procédure E, à ceci près qu'il s'agit de multiplications et pas d'additions :

```
procedure T ;
var new_val : real ;
begin
   F(valeur) ;
   while token = '*' do begin
      next_token(token) ;
      F(new_val) ;
      valeur := valeur * new_val ;
      end ;
end ;
```

4.4 — Si un facteur est un nom de variable, sa valeur est celle de la variable et si un facteur est une expression parenthésée, sa valeur est celle de l'expression elle-même :

```
procedure F ;
begin
   if token in ['a'..'z'] then
   begin valeur := variables[token] ; next_token(token) end
   else
   begin next_token('(') ; E(valeur) ; next_token(')') end ;
   end ;
```

4.5 — Le corps du programme principal est le même que dans la section précédente, à ceci près qu'il faut penser à donner aux variables 'a', 'b', etc. une valeur avant de lancer l'appel de $E(valeur)$.

5. Compiler une expression arithmétique

Dans la partie précédente, nous avons vu comment interpréter (c'est-à-dire évaluer) une expression arithmétique entrée au clavier. Nous connaissons tous l'inconvénient de cette démarche : si l'expression donnée doit être calculée 1000 fois de suite, le programme de la partie précédente *reconnaîtra et évaluera* la même chaîne 1000 fois de suite.
Dans le programme d'interprétation, le *calcul* de la valeur se fait en même temps que la *reconnaissance*. Ne serait-il pas possible de reconnaître la chaîne une fois pour toutes et de séparer ainsi la reconnaissance du calcul de la valeur? Dans l'exemple de la boucle, nous remplacerions 1000 reconnaissances et 1000 calculs de valeur par une seule reconnaissance et 1000 calculs de valeur.

5.1 — Mais quelle forme donner au résultat de la reconnaissance ? Nous choisirons ici la notation *polonaise suffixée*. Rappelons rapidement les propriétés de cette notation. L'expression arithmétique

$$\omega = a + \big((b + c * d) + (e + f)\big)$$

s'écrit en polonais suffixé

$$\pi = abcd * + ef + + + .$$

Si nous disposons de la chaîne π, *l'évaluation* de l'expression arithmétique de l'expression ω est très facile.

L'algorithme est le suivant. Il faut disposer d'une pile (qu'on appelle la *pile d'évaluation*). Au départ, la pile est vide. On lit successivement les caractères de la chaîne "polonaise" :

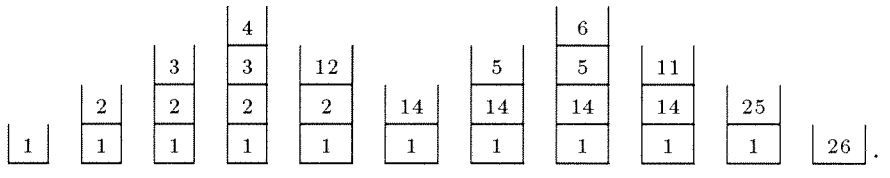
- si le caractère est une lettre (c'est-à-dire un nom de variable), on empile la *valeur* de cette variable dans la pile d'évaluation ;
- si le caractère est un signe '+', on dépile les deux dernières valeurs empilées dans la pile d'évaluation, on fait la somme des deux nombres ainsi obtenus, puis on redépose le résultat au sommet de la pile ;
- les opérations sont analogues si le caractère est le signe '$*$' (en remplaçant naturellement addition par multiplication).

Lorsque la chaîne a été entièrement lue, la pile d'évaluation ne contient plus qu'un seul nombre : c'est la valeur de l'expression.

EXEMPLE : évaluons l'expression $abcd * + ef + + +$ sachant que l'on a

$$a = 1, \quad b = 2, \quad c = 3, \quad d = 4, \quad e = 5 \quad \text{et} \quad f = 6.$$

Les états successifs de la pile d'évaluation sont



5.2 — Reprenons nos trois procédures E, T et F. Le résultat de leur action — c'est-à-dire leur "valeur" — ne sera pas un nombre, mais une *chaîne de caractères*. Plus exactement le résultat de l'appel $E(polonais)$ sera la traduction "en polonais" de l'expression que E a reconnu. De même, le résultat de l'appel $T(polonais)$ (resp. $F(polonais)$) sera la traduction "en polonais" du terme (resp. facteur) que T (resp. F) a reconnu. Au niveau des déclarations, cela donne :

procedure E(**var** *polonais* : *str*255) ; **forward** ;
procedure T(**var** *polonais* : *str*255) ; **forward** ;
procedure F(**var** *polonais* : *str*255) ; **forward** ;

5.3 — Lorsqu'une expression arithmétique est réduite à un terme ($\mathcal{E} = \mathcal{T}$), la procédure E n'a rien à faire; elle se contente de transmettre le résultat qu'a calculé la procédure T. Par contre, si $\mathcal{E} = \mathcal{T}_1 + \mathcal{T}_2$ et si π_1 et π_2 sont les traductions en polonais des termes $\mathcal{T}_1$ et $\mathcal{T}_2$, la procédure E renvoie la chaîne $\pi_1\pi_2+$.

```
procedure E ;
var new_pol : str255 ;
begin
   T(polonais) ;
   while token = '+' do begin
      next_token ;
      T(new_pol) ;
      polonais := concat(polonais, new_pol, '+') ;
      end ;
   if not (token in [')', '$']) then erreur ;
end ;
```

5.4 — La procédure T se traite de manière analogue, à ceci près qu'on y remplace les additions par des multiplications :

```
procedure T ;
var new_pol : str255 ;
begin
   F(polonais) ;
   while token = '*' do begin
      next_token ;
      F(new_pol) ;
      polonais := concat(polonais, new_pol, '*') ;
      end ;
end ;
```

5.5 — La traduction en polonais du facteur a est bien évidemment la chaîne 'a' et la traduction de $(\mathcal{E})$ est celle de $\mathcal{E}$. La procédure F est, une fois de plus, la plus facile à écrire :

```
procedure F ;
begin
   if token in ['a'..'z'] then
      begin polonais := token ; next_token(token) end
   else
      begin next_token('(') ; E(polonais) ; next_token(')') end ;
end ;
```

5.6 — Question : que sont les parenthèses devenues? Considérons par exemple l'expression $a + b + c$. La procédure E la traite comme une somme de trois termes consécutifs, ce qui donne successivement a, $ab+$ et $ab + c+$. Introduisons des parenthèses : $a + (b + c)$. La procédure E traite cette chaîne comme la somme des termes $\mathcal{T}_1 = a$ et $\mathcal{T}_2 = (b + c)$. La traduction en polonais de $\mathcal{T}_2$ est $bc+$. Par conséquent, la traduction de $a + (b + c)$ est $abc++$. Les parenthèses servent donc uniquement à rythmer les appels de procédures.

6. Production de code intermédiaire

Voici une variante de la partie précédente. Il s'agit de remplacer une expression arithmétique par du *code intermédiaire*. On entend par là une suite d'ordres très simples (*résultat = opérande* ou *résultat = opérande1 opérateur opérande2*) qui permet l'évaluation d'une expression arithmétique donnée. Par exemple, l'expression arithmétique

$$a + \big((b + c * d) + (e + f)\big) * \big(x + (y + z) * (u + v)\big)$$

sera remplacée par le code intermédiaire

```
100 : t1=c*d      |  105 : t6=u+v
101 : t2=b+t1     |  106 : t7=t5*t6
102 : t3=e+f      |  107 : t8=x+t7
103 : t4=t2+t3    |  108 : t9=t4*t8
104 : t5=y+z      |  109 : t10=a+t9.
```

Comme nous ne nous autorisons qu'une seule opération à la fois, il est nécessaire d'introduire des *variables temporaires* (qui sont ici les variables $t1,\ldots,t10$). Ces variables mémorisent les calculs intermédiaires. La valeur de l'expression se trouve dans la dernière mémoire temporaire créée (ici $t10$).

Rappelons que nous choisissons d'écrire le code intermédiaire sur l'écran. Toujours dans un but de simplification, nous confondrons un nom de variable avec son emplacement en mémoire.

6.1 — Il est nécessaire de prévoir une procédure *new_temp* qui engendrera les temporaires nécessaires. Puisque nous envoyons tous nos résultats sur écran, la variable globale *temp*, modifiée par la procédure *new_temp*, sera une variable de type chaîne. Nous utiliserons aussi la variable *nb_temp* qui est une variable globale de type entier. Comme son nom l'indique, elle dénombre les temporaires déjà créés :

```
procedure new_temp(var temp : str255) ;
begin
   nb_temp := nb_temp + 1 ;
   NumToString(nb_temp, temp) ;
   temp := concat('t', temp) ;
end ;
```

6.2 — La "valeur" d'une expression sera ici le *code intermédiaire* correspondant à l'évaluation de l'expression reconnue. Plus précisémment, l'appel de $E(place)$ engendrera le code intermédiaire nécessaire pour évaluer l'expression reconnue par E. En outre, la variable *place* contiendra l'emplacement mémoire où sera déposé la valeur de l'expression reconnue par E quand on exécute le code intermédiaire correspondant. Par exemple, lorsque la procédure E reconnaît la chaîne $a + b * c$, elle engendre le code

```
100 : t1=b*c ; 101 : t2=a+t1
```

et retourne l'emplacement `t1` dans *place*. Énoncés analogues pour T et F.
Vu notre choix (tout écrire sur l'écran), la variable *place* est une chaîne de caractères qui contient le nom d'un temporaire ou d'une variable. C'est une variable globale du programme. Au niveau des déclarations, cela nous donne :

procedure E(**var** *place* : $str255$) ; **forward** ;
procedure T(**var** *place* : $str255$) ; **forward** ;
procedure F(**var** *place* : $str255$) ; **forward** ;

6.3 — Lorsque nous avons une somme de termes $\mathcal{T}_1 + \mathcal{T}_2$ ou un produit de facteurs $\mathcal{F}_1 * \mathcal{F}_2$, soient $p1$ et $p2$ les emplacements où seront déposés les valeurs des termes $\mathcal{T}_1$, $\mathcal{T}_2$ ou celui des facteurs $\mathcal{F}_1$, $\mathcal{F}_2$.
Nous devons produire du code pour une addition ou une multiplication. Pour cela, on réserve d'abord un emplacement en mémoire (appel de *new_temp* qui mémorise l'emplacement dans *place*). Cet emplacement accueillera le résultat de l'addition ou de la multiplication. On produit ensuite le code correspondant :

procedure *gen_code*(**var** *place* : $str255$;
$p1 : str255$; $op : char$; $p2 : str255$) ;
begin
new_temp(*place*) ;
writeln(*next_quad* : 0, ' : ', *place*, ' = ', $p1$, op, $p2$) ;
next_quad := *next_quad* + 1 ;
end ;

Remarque : la variable *next_quad* est une variable globale du programme. Elle est de type entier et sert à numéroter les lignes écrites sur l'écran.

6.4 — Lorsqu'une expression arithmétique est réduite à un terme, la procédure E n'a rien à faire; elle se contente de transmettre l'emplacement calculé par la procédure T. Lorsque l'expression est une somme de termes, elle appelle la procédure *gen_code* pour calculer l'emplacement réservé à la somme (remarquer

les modifications successives de la variable *place*) :

```
procedure E ;
var new_place : str255 ;
begin
   T(place) ;
   while token = '+' do begin
      next_token ;
      T(new_place) ;
      gen_code(place, place, '+', new_place) ;
      end ;
   if not (token in [')', '$']) then erreur ;
end ;
```

6.5 — La procédure T se traite de manière analogue :

```
procedure T ;
var new_place : str255 ;
begin
   F(place) ;
   while token = '*' do begin
      next_token ;
      F(new_place) ;
      gen_code(place, place, '*', new_place) ;
      end ;
end ;
```

6.6 — L'emplacement en mémoire de la variable a est, vu nos conventions, le caractère 'a'. Quant à celui de $(\mathcal{E})$, c'est celui de $\mathcal{E}$:

```
procedure F ;
begin
   if token in ['a'..'z'] then
   begin place := token ; next_token(token) end
   else
   begin next_token('(') ; E(place) ; next_token(')') end
end ;
```

6.7 — Le corps du programme consiste à initialiser correctement les différentes variables globales, puis de demander à la procédure E de calculer l'emplacement

de la valeur de l'expression proposée :

```
begin
   write('expression arithmétique = ') ; readln(expression) ;
   expression := concat(expression, '$') ;
   k := 1 ; token := expression[k] ;
   nb_temp := 0 ; next_quad := 100 ;
   E(place) ;
   if token <> '$' then erreur ;
   999 :
end .
```

Comment utiliser moins de temporaires

6.8 — Nous venons d'apprendre à produire du code intermédiaire. Mais la solution proposée souffre d'un grave défaut : les temporaires ne sont pas recyclés. Or il est souvent possible d'évaluer une expression arithmétique compliquée avec peu de temporaires. Pour évaluer l'expression

$$a + \big((b + c * d) * (e * f + g)\big) * \big((x + (y + z)) * (u + v + w)\big),$$

trois temporaires suffisent :

```
100 : t1=c*d      |   106 : t2=x+t2
101 : t1=b+t1     |   107 : t3=u+v
102 : t2=e*f      |   108 : t3=t3+w
103 : t2=t2+g     |   109 : t2=t2*t3
104 : t1=t1*t2    |   110 : t1=t1*t2
105 : t2=y+z      |   111 : t1=a+t1.
```

Avec le programme précédent, il nous en faudrait 12!

6.9 — Voyons comment nous pouvons minimiser le nombre total des temporaires. Pour des raisons qui deviendront claires un peu plus loin, commençons par écrire une fonction booléenne capable de reconnaître si un emplacement mémoire est un temporaire ou celui d'une variable :

```
function is_temp(place : str255) : boolean ;
begin
   if length(place) > 1
   then is_temp := true
   else is_temp := false ;
end ;
```

6.10 — La procédure qui recycle les temporaires est des plus sommaires :

```
procedure release(temp : str255) ;
begin
   nb_temp := nb_temp − 1 ;
end ;
```

6.11 — Lorsque la procédure *gen_code* est activée, celle-ci examine les paramètres $p1$ et $p2$. Il y a quatre cas possibles :

- $p1$ et $p2$ sont deux temporaires. Comme $p1$ a été créé avant $p2$, on produit le code

$$*p1 = *p1 \ * op \ * p2 \ ;$$

et on recycle la variable $p2$ qui ne sert plus à rien (l'étoile indique qu'il faut prendre le *contenu* de la variable correspondante) ;

- $p1$ est un temporaire et $p2$ n'est pas un temporaire. On produit alors le code

$$*p1 = *p1 \ * op \ * p2 \ ;$$

- $p1$ n'est pas un temporaire et $p2$ est un temporaire. On produit alors le code

$$*p2 = *p1 \ * op \ *p2 \ ;$$

- $p1$ et $p2$ ne sont pas des temporaires (ce sont donc deux noms de variables). Dans ce cas, il est nécessaire de d'engendrer un temporaire qui recevra le résultat de l'opération.

$$new_temp(place) \ ; \ *place = *p1 \ * op \ * p2 \ ;$$

Il est crucial ici d'utiliser un temporaire. En effet, si nous utilisions un nom de variable pour y déposer le résultat d'un calcul intermédiaire, la traduction de $x + y + x * z$ risquerait d'être quelque chose comme $x := x + y$; $x := x * z$; $x := x * x$!

On remarquera qu'en procédant ainsi on confère aux temporaires une structure de *pile* (le dernier créé est le premier détruit).

Si on prend la précaution de donner à la variable *place* la bonne valeur dans chaque cas de figure, nous pourrons nous borner à n'engendrer qu'un seul type de

message :

```
procedure gen_code( var place : str255 ; p1 : str255 ;
                        op : char ; p2 : str255) ;
begin
   if is_temp(p1) then begin
      place := p1 ;
      if is_temp(p2) then release(p2)
      end
   else if is_temp(p2)
         then place := p2
         else new_temp(place) ;
   writeln(next_quad : 0, ' : ', place, ' = ', p1, op, p2) ;
   next_quad := next_quad + 1 ;
end ;
```

6.12 — Il n'y a rien à changer au reste du programme.

7. Un préprocesseur : le TRRUmpilateur (*)

Supposons que nous voulions travailler en virgule fixe sur une machine. Si le langage de programmation implanté sur cette machine est assez riche, il existe probablement un type *virgule fixe* qui autorise des additions, des multiplications et des divisions. Cette solution — très souvent utilisée en graphisme — conduit à des calculs beaucoup plus rapides qu'en virgule flottante.

Nous allons ici explorer une autre voie. Au lieu d'utiliser un type virgule fixe et de laisser gérer les retenues par la machine, nous allons tout faire nous-mêmes. Les calculs seront aussi rapides qu'en virgule fixe (puisque ce seront uniquement des calculs portant sur des entiers). Mais nous aurons la maîtrise complète des opérations.

7.1 — Mettons en place notre arsenal théorique. Soit $N > 1$ un nombre entier. Nous lui associons deux ensembles :

- l'ensemble VF (pour *virgule fixe*) des nombres rationnels x de la forme

$$x = x_0 + \frac{x_1}{N},$$

où $x_0, x_1 \in \mathbb{Z}$ sont deux entiers avec $0 \leq x_1 < N$;

- l'ensemble RM (pour *représentation en machine*) l'ensemble des couples (x_0, x_1) appartenant à $\mathbb{Z} \times \mathbb{N}$ tels que $0 \leq x_1 < N$.

(*) Se reporter à L'OUVERT **48** pages 25–30.

Le nombre rationnel $x \in \mathrm{VF}$, s'il est de taille raisonnable, sera représenté en machine par le couple $(x_0, x_1) \in \mathrm{RM}$. Pécisons ce que nous entendons par "représenter" x par le couple (x_0, x_1). On dispose de deux applications naturelles

$$\mathrm{code} : \mathrm{VF} \to \mathrm{RM} \quad \text{et} \quad \mathrm{valeur} : \mathrm{RM} \to \mathrm{VF}$$

définies par les formules

$$\mathrm{code}(x) = (x_0, x_1) \quad \text{avec} \quad \begin{cases} x_1 = (N * x) \ \mathbf{mod}\ N, \\ x_0 = (N * x - x_1) \ \mathbf{div}\ N; \end{cases}$$
$$\mathrm{valeur}(x_0, x_1) = x_0 + \frac{x_1}{N}\cdot$$

L'application de *normalisation* $[\![\ ]\!] : \mathbb{Z} \times \mathbb{Z} \to \mathrm{RM}$ définie par

$$[\![(x_0, x_1)]\!] = \big(x_0 + (x_1 \ \mathbf{div}\ N), x_1 \ \mathbf{mod}\ N\big).$$

permet de préciser les liens qui relient les applications code et valeur :

$$\mathrm{valeur}(\mathrm{code}(x)) = x \quad \text{et} \quad \mathrm{code}(\mathrm{valeur}(x_0, x_1)) = [\![(x_0, x_1)]\!].$$

7.2 — Pour trouver le code (dans RM) associé à la somme de deux nombres de VF, nous devons écrire, avec $t = x_1 + y_1$,

$$(1) \qquad \begin{aligned} x + y &= x_0 + y_0 + \Big(\frac{x_1 + y_1}{N}\Big) \\ &= x_0 + y_0 + (t \ \mathbf{mod}\ N) + \frac{t \ \mathbf{div}\ N}{N}\cdot \end{aligned}$$

Comme on s'y attend, trouver le code du produit de deux éléments de VF est beaucoup plus compliqué car il y a deux retenues. En posant $u = x_1 * y_1$ et $t = x_1 * y_0 + x_0 * y_1 + (u \ \mathbf{div}\ N)$, on a

$$(2) \qquad \begin{aligned} x * y &= \Big(x_0 + \frac{x_1}{N}\Big) * \Big(y_0 + \frac{y_1}{N}\Big) \\ &= x_0 * y_0 + \frac{x_0 y_1 + x_1 y_0}{N} + \frac{x_1 * y_1}{N^2} \\ &= x_0 * y_0 + (t \ \mathbf{div}\ N) + \frac{t \ \mathbf{mod}\ N}{N} + \frac{u \ \mathbf{mod}\ N}{N^2}\cdot \end{aligned}$$

7.3 — Les formules (1) et (2) suggèrent de munir l'ensemble RM d'une addition $\oplus$ et d'une multiplication $\otimes$ telles que :

$$(\mathrm{TRRU}) \qquad \begin{cases} (x \oplus y)_0 = x_0 + y_0 + (x_1 + y_1) \ \mathbf{div}\ N, \\ (x \oplus y)_1 = (x_1 + y_1) \ \mathbf{mod}\ N; \\ (x \otimes y)_0 = x_0 * y_0 + (t \ \mathbf{div}\ N), \\ (x \otimes y)_1 = t \ \mathbf{mod}\ N, \end{cases}$$

où, pour simplifier la définition de la multiplication, on a posé

$$u = x_1 y_1 \quad \text{et} \quad t = x_0 y_1 + x_1 y_0 + (u \ \mathbf{div}\ N).$$

La loi $\oplus$ est "exacte" en ce sens que l'on a

$$x + y = \text{valeur}\big(\text{code}(x) \oplus \text{code}(y)\big)$$

pour tous $x, y \in \text{VF}$. Par contre, la loi $\otimes$ n'est qu'une approximation de la multiplication des nombres rationnels :

$$x * y = \text{valeur}\big(\text{code}(x) \otimes \text{code}(y)\big) + \frac{u \ \mathbf{mod}\ N}{N^2}.$$

L'erreur commise en remplaçant $x * y$ par $\text{valeur}(\text{code}(x) \otimes \text{code}(y))$ est donc majorée par N^{-1}.

Il résulte de cela que RM, muni de la loi $\oplus$, est un groupe abélien. Remarquons au passage que l'opposé de $(x_0, x_1) \in \text{RM}$ est

$$-x = (-x_0 - 1, N - x_1).$$

Par contre, la loi $\otimes$ n'est pas associative; RM n'est donc pas un anneau.

7.4 — La loi $\oplus$ gère automatiquement la retenue. Introduisons donc une variante de cette loi qui ne le fait pas :

$$x \oplus' y = (x_0 + y_0, x_1 + y_1).$$

Cette loi est définie sur $\mathbb{Z} \times \mathbb{N}$ et l'ensemble RM n'est pas stable pour cette addition. Les additions $\oplus$ et $\oplus'$ sont liées par la relation

$$x \oplus y = [\![x \oplus' y]\!].$$

7.5 — Donnons-nous maintenant une expression arithmétique

$$\omega = (a * x + b) * (c * x * x + d * x + e).$$

Si nous connaissons les codes (dans RM) des nombres a, b, etc, nous pouvons calculer la valeur de l'expression

$$\omega' = (a \otimes x \oplus b) \otimes (c \otimes x \otimes x \oplus d \otimes x \oplus e).$$

ou, si nous désirons gérer "à la main" les retenues, calculer la valeur de l'expression

$$\omega'' = [\![a \otimes x \oplus' b]\!] \otimes [\![c \otimes x \otimes x \oplus' d \otimes x \oplus' e]\!].$$

Et il y a encore beaucoup d'autres variantes!

Précisons ce que nous entendons par *calculer* la valeur de ω' ou de ω'' : on fait comme si $\oplus$ et $\otimes$ étaient une "vraie" addition et une "vraie" multiplication. Autrement dit, les règles usuelles d'évaluation de gauche à droite, de priorité, de parenthésage, etc, sont respectées.

7.6 — Considérons l'expression très simple

$$\omega = (a * x + b) * (c * y + d).$$

Pour calculer la valeur de $\omega' = (a \otimes x \oplus b) \otimes (c \otimes y \oplus d)$, nous devrons générer à partir de ω un source PASCAL qui peut ressembler à ceci :

$$\begin{array}{l}
temp := a0 * x1 + a1 * x0 + (a1 * x1) \textbf{ div } N ; \\
TA0 := a0 * x0 + (temp \textbf{ div } N) ; \\
TA1 := temp \textbf{ mod } N ; \\
temp := TA1 + b1 ; \\
TB0 := TA0 + b0 + (temp \textbf{ div } N) ; \\
TB1 := temp \textbf{ mod } N ; \\
temp := c0 * y1 + c1 * y0 + (c1 * y1) \textbf{ div } N ; \\
TC0 := c0 * y0 + (temp \textbf{ div } N) ; \\
TC1 := temp \textbf{ mod } N ; \\
temp := TC1 + d1 ; \\
TD0 := TC0 + d0 + (temp \textbf{ div } N) ; \\
TD1 := temp \textbf{ mod } N ; \\
temp := TB0 * TD1 + TB1 * TD0 + (TB1 * TD1) \textbf{ div } N ; \\
TE0 := TB0 * TD0 + (temp \textbf{ div } N) ; \\
TE1 := temp \textbf{ mod } N ;
\end{array}$$

Explications : la variable temporaire $TA = (TA_0, TA_1)$ recueille le résultat du produit $a * x$ puis TB recueille la valeur de $a * x + b$, etc. Le code de la valeur de ω (ou plutôt le code d'une valeur approchée de ω) se trouve dans la dernière variable temporaire TH.

Générer un tel code est ennuyeux et mécanique, ce qui est source d'erreurs. Pourquoi ne pas écrire un programme PASCAL qui lirait l'expression ω et qui écrirait le code à notre place?

Pour cela, commençons par modifier la procédure *new_temp*. En effet, les temporaires s'appellent maintenant TA, TB, etc. :

```
procedure new_temp(var temp : str255) ;
begin
    temp := chr(ord('A') + nb_temp) ;
    temp := concat('T', t) ;
    nb_temp := nb_temp + 1 ;
end ;
```

7.7 — La procédure *gen_add* se charge de d'écrire sur l'écran le code correspondant

à une addition $\oplus$ et de réserver un emplacement pour le résultat :

```
procedure gen_add(var place : str255 ; p, q : str255) ;
var s : str255 ;
begin
    s := concat(p, '1 + ', q, '1') ;
    new_temp(place) ;
    writeln('temp := ', s, ' ; ') ;
    writeln(place, '1 := temp mod N ; ') ;
    writeln(place, '0 := ', p, '0 +', q, '0 + (temp div N ) ; ') ;
end ;
```

7.8 — Nous allons demander à la procédure E de :

- d'écrire sur l'écran un code PASCAL correspondant à l'expression (ou la sous-expression) qu'elle a reconnu ;
- renvoyer le nom de la variable qui contient le résultat de l'évaluation du code PASCAL

```
procedure E ;
var new_place : str255 ;
begin
    T(place) ;
    while token = '+' do begin
        next_token(token) ;
        T(new_place) ;
        gen_add(place, place, new_place) ;
        end ;
    if not (token in [')', '$']) then erreur ;
end ;
```

7.9 — La procédure *gen_mult* se charge d'écrire sur l'écran le code correspondant à une multiplication $\otimes$ et de réserver un emplacement pour le résultat :

```
procedure gen_mult(var place : str255 ; p, q : str255) ;
var u : str255 ;
begin
    u := concat(p, '1 * ', p, '1') ;
    writeln('temp := ',  p, '0 * ', q, '1 + ',
                         p, '1 * ', q, '0 + (', u, 'div N ) ; ') ;
    new_temp(place) ;
    writeln(place, '1 := temp mod N ; ') ;
    writeln(place, '0 := ', p, '0 *', q, '0 + (temp div N ) ; ') ;
end ;
```

7.10 — La procédure T est analogue à la procédure E : on remplace simplement les additions par des multiplications.

```
procedure T ;
var new_place : str255 ;
begin
    F(place) ;
    while token = '*' do begin
        next_token(token) ;
        F(new_place) ;
        gen_mult(place, place, new_place) ;
        end ;
end ;
```

7.11 — Enfin, la procédure F se contente d'inspecter les lieux et de prendre note, comme d'habitude :

```
procedure F ;
begin
    if token in ['a'..'z'] then
    begin place := token ; next_token(place) end
    else if token = '(' then
    begin next_token('(') ; E(place) ; next_token(')') end ;
end ;
```

7.12 — Il n'y a rien à changer au corps du programme principal.

7.13 — EXERCICE : modifier le programme pour qu'il engendre le moins possible de variables temporaires.

(à suivre ...)