COMMENT COMPILER OU INTERPRÉTER UNE EXPRESSION ARITHMÉTIQUE

(suite)

Raymond SEROUL

8. Dérivation formelle

Dans cette partie, nous allons apprendre comment *dériver* une expression arithmétique. Ce que nous voulons, c'est que le programme nous réponde que la dérivée par rapport à x de la fonction

$$(a * x * x + b * x + c) * (d * x + e)$$

est la fonction

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

On notera qu'il s'agit de manipuler des chaînes de caractères.

8.1 — Les constantes suivantes rendront plus facile la lecture du programme :

```
const  zero = '0' ;
       un = '1' ;
```

8.2 — Quand on programme naïvement, on s'aperçoit très vite que les dérivées obtenues sont très maladroites : la dérivée de $a * x * x$ est

$$(a * 1 + 0 * x) * (x) + (a * x) * (1) \ !$$

Ce n'est pas faux, mais il y a trop de parenthèses et trop de facteurs inutiles. Pour remédier à cela, nous allons créer un type :

```
type fonction = record
                  text : str255 ;
                  plus : boolean ;
                end ;
```

Le champ *text* contient la fonction elle-même. Le champ *plus* indique si la chaîne *text* contient un signe '+' non *caché* par des parenthèses. Précisons ce que nous entendons par là : si *text* est l'une des chaînes

$$a, \quad a * b, \quad (a + b) * c,$$

le champ *.plus* est faux. Par contre, il est vrai pour les chaînes

$$a+b, \quad a+(\cdots), \quad (\cdots)+c.$$

8.3 — Nous aurons besoin d'*ajouter* deux expressions. Mais comme ces expressions-là sont des chaînes de caractères, il faut faire attention : les chaînes '$a+0$', '$0+a$' et 'a' sont distinctes! Il est donc nécessaire de prévoir une procédure qui *ajoute* intelligemment deux chaînes :

```
procedure somme(var u : fonction; v1, v2 : fonction) ;
begin
   if (v1.text = zero) and (v2.text = zero) then begin
      u.text := zero ; u.plus := false ; exit(somme) ;
      end ;
   if (v1.text <> zero) and (v2.text = zero) then
      begin u := v1 ; exit(somme) end ;
   if (v1.text = zero) and (v2.text <> zero) then
      begin u := v2 ; exit(somme) end ;
   if (v1.text <> zero) and (v2.text <> zero) then begin
      u.text := concat(v1.text, '+', v2.text) ; u.plus := true ;
      exit(somme) ;
      end ;
end ;
```

8.4 — Il est parfois nécessaire de parenthéser un facteur quand on l'insère dans un produit. Par exemple, il est inutile de parenthéser a et $(b+c)$ si on les multiplie. Par contre, il est indispensable de parenthéser $a*x+b$ s'il figure dans un produit : $a*(b+c)*(a*x+b)$. Pour éviter une multiplication excessive de parenthèses inutiles, il est prudent de prévoir une procédure capable de parenthéser une expression uniquement lorsque c'est nécessaire :

```
procedure parentheser(var u : fonction) ;
begin
   if (length(u.text) > 1) and u.plus
   then  begin
            u.text := concat('(', u.text, ')') ; u.plus := false ;
            end ;
end ;
```

8.5 — Lorsqu'on *multiplie* deux expressions représentées par des chaînes de caractères, il faut encore faire plus attention que dans le cas d'une somme. En effet, la situation est plus riche : les chaînes '$a*1$' et '$a*0$' sont différentes des

chaînes 'a' et '0'.

```
procedure produit(var u : fonction ; v1, v2 : fonction) ;
begin
   if (v1.text = zero) or (v2.text = zero) then begin
      u.text := zero ; u.plus := false ; exit(produit) ;
      end ;
   if (v1.text = un) and (v2.text = un) then begin
      u.text := un ; u.plus := false ; exit(produit) ;
      end ;
   if (v1.text <> un) and (v2.text = un) then
       begin u := v1 ; exit(produit) end ;
   if (v1.text = un) and (v2.text <> un) then
       begin u := v2 ; exit(produit) end ;
   if (v1.text <> un) and (v2.text <> un) then begin
      parentheser(v1) ; parentheser(v2) ;
      u.text := concat(v1.text, '*', v2.text) ; u.plus := false ;
      exit(produit) ;
      end ;
end ;
```

8.6 — La formules de dérivation $(uv)' = u'v+uv'$ nous apprend qu'il est nécessaire de connaître simultanément les quatre fonctions u, v, u' et v' pour calculer la dérivée de uv. C'est pourquoi la déclaration des procédures E, T et F est ici :

```
procedure E(var u, du : fonction) ; forward ;
procedure T(var u, du : fonction) ; forward ;
procedure F(var u, du : fonction) ; forward ;
```

La procédure E renvoie dans la chaîne $u.text$ une version simplifiée de l'expression qu'elle a reconnu et dans $du.text$ la dérivée de $u.text$. Nous avons des énoncés analogues pour T et F.

8.7 — Il n'y a rien de spécial concernant la procédure E. On écrit toujours le même code, avec quelques variantes :

```
procedure E ;
var v, dv : fonction ;
begin
   T(u, uprime) ;
   while token = '+' do begin
      next_token(token) ;
      T(v, dv) ;
      somme(u, u, v) ; somme(du, du, dv)
      end ;
   if not (token in [')', '$']) then erreur ;
end ;
```

8.8 — Mêmes commentaires concernant la procédure T. Un point technique : cette procédure contient les affectations $u := u * v$ et $u' := u'v + uv'$. Il faut donc calculer u' d'abord (sinon, la fonction u qui figurera dans le calcul de u' ne sera pas la bonne) :

```
procedure T ;
var v, dv, alpha, beta : fonction ;
begin
   F(u, du) ;
   while token = '*' do begin
      next_token(token) ;
      F(v, dv) ;
      produit(alpha, du, v) ; produit(beta, u, dv) ;
      somme(du, alpha, beta) ;
      produit(u, u, v) ;
      end ;
end ;
```

8.9 — La procédure F est — pour une fois — plus intéressante à écrire. La variable x est traitée à part. Lorsque la procédure F traite une expression parenthésée, elle renvoie l'expression et sa dérivée *sans* les parenthèses. Celles-ci seront rétablies lorsque le besoin s'en fera sentir (c'est pour cela que le champ *.plus* existe). Par conséquent, après appel de $F(u, du)$, l'expression $u.text$ renvoyée par F n'est pas tout à fait l'expression que F a reconnue, mais une version simplifée :

```
procedure F ;
begin
   if token = 'x' then begin
      u.text := 'x' ; u.plus := false ;
      du.text := un ; du.plus := false ;
      next_token(token) ; exit(F) ;
      end ;
   if token in ['a'..'z'] then begin
      u.text := token ; u.plus := false ;
      du.text := zero ; du.plus := false ;
      next_token(token) ; exit(F) ;
      end ;
   next_token('(') ; E(u, du) ; next_token(')') ;
end ;
```

8.10 — Le corps du programme principal est, à quelques variantes près, toujours le même. Il consiste à initialiser correctement les variables globales k, *expression* et *token*. On appelle ensuite $E(u, du)$ (ce qui exige d'avoir déclaré u et du comme variables globales du programme), puis on demande l'affichage de $du.text$ (et aussi celui de $u.text$ si l'on est curieux). Une remarque : pourquoi ne pas avoir

lancé $E(expression, du)$? Rappelons-nous que la fonction *next_token* se réfère à *expression*. On ne peut donc pas communiquer *expression* à la procédure E, car celle-ci modifie ses arguments!

```
begin
   write('expression à dériver = '); readln(expression) ;
   expression := concat(expression, '$') ;
   k := 1 ; token := expression[k] ;
   E(u, du) ;
   if token <> '$' then erreur
   else writeln('dérivée = ', du.text) ;
   999 :
   end ;
end .
```

9. Extentions possibles

Une expression arithmétique ne contient pas que des additions ou des multiplications. On rencontre aussi des soustractions, des divisions et des appels de fonctions. (Les signes '$-$' unaires seront traités dans la partie suivante.)

9.1 — La définition d'une expression arithmétique se généralise facilement. Continuons à noter les noms de variables et les constantes par les minuscules de 'a' à 'z'. Notons les noms de fonctions par des majuscules, de 'A' jusqu'à 'Z'. Ceci posé :

- une *expression* est une chaîne de caractères $\mathcal{E}$ qui est un terme $\mathcal{T}$ ou de la forme $\mathcal{T}\theta\mathcal{T}\theta\cdots\theta\mathcal{T}$, le signe θ étant à prendre parmi les signes '$+$' ou '$-$';
- un *terme* $\mathcal{T}$ est une chaîne de caractères qui est un facteur $\mathcal{F}$ ou de la forme $\mathcal{F}\varphi\mathcal{F}\varphi\cdots\varphi\mathcal{F}$, le signe φ étant à prendre parmi les signes '$*$' ou '$/$';
- un *facteur* $\mathcal{F}$ est une chaîne de caractères qui est un nom de variable (de a à z), une expression parenthésée ($\mathcal{E}$) ou une expression parenthésée précédée d'un nom de fonction (de $A(\mathcal{E})$ à $Z(\mathcal{E})$ donc).

9.2 — La procédure E doit tenir compte de la présence des signes '$-$' :

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

9.3 — La procédure T doit tenir compte de la présence des signes '$/$' :

```
procedure T ;
begin
   F ;
   while token in ['*', '/'] do
   begin next_token ; F end ;
end ;
```

9.4 — La procédure F examine si le caractère courant de la chaîne d'entrée est un nom de variable, un nom de fonction ou une expression parenthésée :

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

10. L'analyse lexicale

Il s'agit dans cette partie de décrire l'interface indispensable entre une "vraie" expression arithmétique (par exemple $\exp(3.14 * x + 0.135)$) et la vision idéale avec laquelle nous avons travaillé jusqu'ici (les variables, les constantes et les noms de fonctions sont représentées par un seul caractère).

10.1 — Lorsque vous lisez, votre cerveau assemble certaines lettres pour en faire des mots. Un ordinateur fait exactement la même chose. Quand je tape

$$\sin(3 * x + 5) + x * x - 12.35$$

j'introduit en mémoire une chaîne de caractères (stockée d'habitude sous la forme d'un tableau), ce qu'on peut symboliser par

s	i	n	(	3	*	x	+	5	)	+	x	*	x	−	1	2	.	3	5

L'ordinateur commence par lire les trois lettres 's', 'i' et 'n' et reconnaît le mot 'sin'. La parenthèse ouvrante forme à elle seule un mot, etc. Dans notre exemple, l'ordinateur reconnaîtrait les mots :

sin	(	3	*	x	+	5	)	+	x	*	x	−	12.35

Au lieu de 'mot', les informaticiens préfèrent utiliser 'lexème'.
Les lexèmes ont des formats et des significations différentes ('sin', 'x', '+', etc.). Pour fonctionner de manière efficace, l'ordinateur ne se contente pas de découper la chaîne d'entrée en lexèmes. Il *code* ceux-ci sous une forme convenable, de format fixe. Traditionnellement, un lexème codé s'appelle un *token*.
L'information portée par un token est à deux niveaux (par exemple sous la forme d'un record). Un premier niveau nous apprend s'il s'agit d'un identificateur,

d'un nom de fonction, d'une parenthèse, d'une opération binaire. Les autres informations utiles (le nom d'une variable, la valeur d'une constante, le type, etc.) sont mémorisées dans une *table des symboles*. Ces informations sont accessibles à partir du token grâce à un pointeur vers la table des symboles.

10.2 — Comme nous nous occcupons uniquement d'expressions arithmétiques, nous pouvons employer une forme très simple de codage :

- Chaque lexème est codé par un entier différent de 0.
- Les codes des lexèmes sont stockés dans le tableau $tokens[1..50]$.
- L'entier nul marque la fin de la suite des tokens (en remplacement du caractère '$' que nous avons utilisé).
- Un entier strictement positif représente *l'adresse*, dans le tableau $valeurs[1..50]$, d'une variable ou d'une constante. Par convention, la valeur de la variable x est à l'adresse 1 dans ce tableau.
- Un entier strictement négatif représente un nom de fonction, une parenthèse ou une opération. Pour ne pas nous surcharger la mémoire, nous utiliserons une facilité que nous offre PASCAL. Au début de notre programme, nous déclarerons ces entiers négatifs comme des *constantes* :

const
$\_sin = -1$; $\_cos = -2$; $\_exp = -3$;
$\_parg = -4$; $\_pard = -5$;
$\_plus = -6$; $\_moins = -7$; $\_mult = -8$;

La valeur de ces entiers importe peu car nous ne ferons références à ces entiers qu'à travers leur nom.

10.3 — Après codage de l'expression précédente, le tableau *valeurs* se présenterait comme ceci :

?	3	5	12.35	?	?	...
1	2	3	4	5	6	

et le tableau *tokens* comme cela :

_sin	_parg	2	_mult	1	_plus	2	_pard
1	2	3	4	5	6	7	8

1	_mult	1	_moins	4	0	?	...
9	10	11	12	13	14	15	16

Les points d'interrogation symbolisent des variables non initialisées. Il est intéressant de noter que la valeur de x n'a pas besoin d'être connue au moment de l'analyse lexicale ; cette valeur ne sert qu'au moment de l'évaluation de l'expression.

10.4. — Si nous adoptons cette suggestion :

• La variable *token* devient une variable de type entier qui *pointe* sur le tableau *tokens*.

• La procédure *next_token* (qui s'appelle traditionnellement *l'analyseur lexical*) déchiffre la chaîne d'entrée *expression*, code le lexème reconnu (en entrant l'entier convenable dans le tableau *tokens* et en entrant, si nécessaire, la valeur d'une constante dans le tableau *variables*) et renvoie ses informations dans *token*.

10.5 — Un dernier mot, comme promis, sur les signes '−' unaires. Comment décider si un signe '−' est un signe unaire ou binaire? Pour cela, il suffit de regarder le token qui précède. Si le signe '−' est le premier caractère de l'expression, s'il est précédé d'une parenthèse ouvrante ou d'un signe opératoire (unaire ou binaire), alors c'est un signe unaire. Dans tous les autres cas, le signe '−' est un signe binaire. Bien entendu, un signe '−' binaire ne sera pas codé de la même manière qu'un signe '−' unaire dans le tableau *tokens*.

Toujours pour éviter les phénomènes de "bord de chaîne", il est plus recommandé d'entourer l'expression de départ de parenthèses (et de la faire suivre du symbole dollar '$'). Cela simplifie la détection des signes '−' unaires qui peuvent débuter une expression.

Il n'y a presque rien à changer aux procédures E, T et F. Il suffit d'introduire un nouveau type de facteur : un facteur précédé d'un signe moins unaire (en symboles : $\dot{-}\mathcal{F}$) est encore un facteur. (Ici, le signe '−' unaire est typographié $\dot{-}$.)

A vos claviers!
