Lse-83

Jacques Arsac

LSE 83

Jacques ARSAC

1. LA COMMUNAUTÉ LSE

Nous avons passé notre temps à dire et répéter que le langage de programmation n'est jamais l'essentiel. En particulier, en aucun cas, un cours de programmation ne devrait être la description d'un langage. Ce qui compte, ce n'est pas la façon de dire les choses, c'est d'avoir quelque chose à dire.

"Avant donc que d'écrire, apprenez à penser.
Ce que l'on conçoit bien s'énonce clairement,
Et les mots pour le dire arrivent aisément".

Ce que Boileau a dit du français vaut entièrement en programmation. Cela ne veut pas dire que pour autant le langage soit méprisable. Il véhicule une culture. Il est le lien d'une communauté. De même que les Québécois se sont battus pour leur droit à parler Français, de même il est normal que ceux qui ont appris un langage de programmation défendent ce qui est pour eux un moyen de communication entre hommes : un programme est d'abord écrit pour être lu par un homme, non par une machine (H. Ledgard). Le LSE est le lien de la communauté des professeurs de lycée ayant fait de l'informatique.

Mais ce langage est attaqué sur divers fronts. Il y a d'abord toutes les opérations de vulgarisation de l'informatique, qui jettent précipitamment dans les écoles des nano-ordinateurs ne connaissant qu'un petit BASIC. Pas le temps de faire mieux. Il vous faudra parler BASIC... Comme les Québécois obligés de parler anglais ont craint pour leur langue maternelle, on peut douter de l'avenir de LSE.

D'autant que les constructeurs sont là pour pousser à la roue. L'implémentation de LSE sur un micro-ordinateur est une grosse opération, onéreuse. Vive BASIC !

Dans le même temps, les journaux vont répétant l'éternelle rengaine : l'informatique s'apprend à raison d'une heure par jour en une ou deux semaines (Laurence Bacman et Alain Ehrenberg, du club Méditerranée, dans Esprit, p. 27, février 85).

"Les quelques mots du BASIC qu'on aura appris forceront ceux qui savent à se dessaisir d'une propriété exclusive (M. Lacroix, dans Esprit p.12, février 1985). L'apprentissage de quelques mots de BASIC est présenté comme l'arme contre la technocratie informatique.

Après cela, allez donc faire comprendre à un parent d'élève que sa fille va apprendre LSE : elle sera la victime des technocrates, elle ne connaîtra pas BASIC!

A l'opposé, les universitaires qui ont partie liée avec les professeurs de lycée ne jurent que par PASCAL : il est mieux structuré, plus riche ... Je ne suis pas tout à fait sûr que certains (bon nombre ?) d'entre eux ne soient retombés dans le piège du langage. Il est certain que l'enseignement du langage PASCAL est moins désagréable que celui du langage LSE (et a fortiori de BASIC). Mais l'expérience maintenant importante de l'enseignement de l'informatique dans les options des lycées a montré que l'on pouvait vivre avec LSE. Certes, certains font un enseignement déguisé de PASCAL et donnent des schémas de traduction de WHILE ou CASE en LSE. La plupart utilisent directement LSE, sans trop de complexes.

Ainsi coincé entre BASIC et PASCAL, quelles sont les chances de LSE ? Or, je ne suis pas prêt à y renoncer. Chacun est attaché à sa langue. C'est du reste ce qui explique l'aspect passionnel que revêt tout débat sur les langages de programmation.

On a parfois l'impression, en mettant en doute quelque qualité d'un langage, de commettre un sacrilège! d'autant qu'il y a des raisons objectives pour vouloir préserver LSE. Outre qu'il est l'œuvre de Français (Hebenstreit, Noye11e et Berche, 1970), ses mots clefs sont en Français, ce qui est important pour nos écoles. Il est vrai que l'on peut traduire les mots clefs de BASIC ou PASCAL en Français. (Un journaliste de la télévision a présenté un jour la grande nouveauté d'un SICOB : on peut maintenant programmer en français. Cela ne voulait pas dire que le français était devenu langage de programmation, mais plus simplement que l'on écrivait SOIT au lieu de LET en BASIC!). Mais un BASIC français n'est plus BASIC, et l'on perd l'avantage d'appartenir à la communauté BASIC. On n'a pas facilité l'intégration à une communauté, on a créé une nouvelle secte.

LSE a un système très évolué de traitement de chaînes de caractères. Depuis 1973, tous mes travaux de programmation ont été faits en LSE, à une exception près. Etant à !Rio de Janeiro en 1977, j'ai écrit un système sophistiqué de manipulation de programmes en SNOBOL, le langage le plus évolué en matière de traitement de texte. J'ai retranscrit ce programme en LSE : je suis passé de 500 lignes SNOBOL à 500 lignes LSE. Selon mon expérience, on a le classement suivant entre les grands langages traitant des chaînes de caractères

PASCAL < APL < LSE < SNOBOL.

Je n'ai pas mis LISP dans cette liste : c'est un langage applicatif, et sa comparaison à des langages impératifs (munis de l'instruction d'affectation) n'a guère de sens.

La pratique de LSE, et son étonnante facilité à manipuler les chaînes de caractères m'a obligé à réviser beaucoup d'idées reçues. Une pile, c'est un vecteur et un pointeur. En LSE, on réalise beaucoup plus facilement une pile en concaténant en tête d'une chaîne, puis en extrayant pas SCH ou GRL... Un arbre, c'est un système de pointeurs fils-frère. En LSE, il vaut beaucoup mieux représenter un arbre par une chaîne parenthèsée. Les pointeurs deviennent inutiles et, en tant que professeur, j'en suis bien content.

Mais il est certain que LSE souffre d'avoir été conçu vers 1968-1969, en dehors du courant de pensée de la programmation structurée. Il offre des inconvénients pédagogiques manifestes qui gênent les professeurs :

- le ALLER EN oblige à gérer des numéros de ligne, et c'est malheureux en même temps que fastidieux.
- la confusion entre fonction et sous-programme, regroupés sous le vocable commun PROCEDURE, est source d'obscurité.
- la confusion entre paramètre formel-donnée et variable locale est une autre source de difficulté.
- l'absence de la notion de corps de procédure ne facilite pas les choses (on peut aller d'une procédure dans une autre par ALLER EN).
- le SI... ALORS... SINON sur une ligne restreint beaucoup la possibilité d'une bonne structuration des programmes, et impose l'usage de ALLER EN.

Le langage Fortran est né en 1955. Il est toujours extrêmement vivant en 1985. Il a résisté à tous les assauts Algol 60, PL/1, Pascal,

ADA. Mais cela vient essentiellement qu'il a réussi des aggiornamento successifs. Il y a loin du Fortran 0 de Backhus en 1955 au Fortran 4, H, 77…

Il me semble que si LSE ne se met pas au goût du jour, il sera difficile de le faire survivre. Si l'on ne veut pas le laisser devenir, langue morte (Langage Sans Espoir m'a-t-on dit au dernier SICOB) il faut le faire vivre en le faisant évoluer. Je propose ici une version fortement structurée de LSE, conservant la physionomie du langage qui nous est cher, mais agréable à utiliser et ne mettant pas l'enseignant devant d'insurmontables contradictions. Je décrirai d'abord le langage, puis l'implémentation que j'en ai faite et qui est aisément portable. Ce n'est pas un objet figé. Je suis ouvert à toute bonne suggestion.

2. ASPECT GÉNÉRAL DU LANGAGE

Le plus profond changement de LSE à LSE83, est la disparition totale de l'instruction ALLER EN. Ceci entraîne la disparition de tout numéro de ligne dans le texte d'un programme.

La numérotation des lignes est pourtant maintenue, pour bénéficier des facilités de traitement de texte que cela apporte :

- possibilité de lister de la ligne p à la ligne q ,
- possibilité de changer une ligne en la retapant sous le même numéro,
- possibilité d'introduire un programme suivant la logique de la pensée (on écrit l'initialisation d'une boucle après le corps de la boucle), l'ordre des lignes étant celui des numéros, non celui de la frappe des lignes,
- possibilité d'insérer des lignes)
- possibilité de modifier une ligne en la listant, puis en utilisant l'éditeur de ligne (signes %)
- accessoirement (mais pour moi c'est important) possibilité de structurer un programme, et de manifester cette structure, par une bonne gestion des numéros de ligne : on change de centaine à chaque paragraphe, de millier à chaque chapitre...

Les expressions LSE demeurent inchangées, ainsi que le mécanisme des types et déclarations de types du langage. Tout ce qui touche les fichiers demeure inchangé.

Dans la version actuelle de LSE 83, on a ajouté de nouvelles facilités à l'éditeur de texte qui est partie intégrante de LSE :

SElectionner lignes avec < identificateur >

En réponse, on obtient la liste de toutes les lignes comportant cet identificateur. Ceci permet de retrouver les lignes où l'on a utilisé un certain identificateur (par exemple, quand on a utilisé le même nom pour une variable simple et un tableau), mais aussi de savoir si un certain identificateur est utilisé dans le programme (rien n'est affiché s'il ne l'est pas).

SAuvegarder &<nom> range sur disquette le texte d'une procédure (fonction ou sous-programme, en ramenant ses numéros de ligne à commencer à la ligne 0).

ATtacher &<nom> <numéro> copie à la suite du programme le texte d'une procédure précédemment sauvegardé, en numérotant les lignes à partir du numéro indiqué. Ceci permet de constituer une bibliothèque de procédures dans lequel on puise librement.

Les commentaires dans un programme sont enfermés entre accolades afin que toute parenthèse ouverte soit fermée, et que l'on puisse insérer des commentaires même en cours de ligne.

3. L'INSTRUCTION SI

Elle a deux formes possibles :

```
SI <expression booléenne> ALORS <suite d'instructions> IS
```

IS est la parenthèse fermante associée à la parenthèse ouvrante SI.

Sa présence a un double intérêt :

- du point de vue grammatical : il faut fermer les parenthèses que l'on a ouvertes,
- son emploi évite les délimiteurs DEBUT FIN de LSE, et donc allège l'écriture.

```
SI <exp.bool.> ALORS <suite instr.> SINON <suite instr.> IS
```

Même remarque sur le fait que les délimiteurs DEBUT ... FIN sont inutiles. Exemples :

```
SI I>8 ALORS I ← I - 8 IS
SI I>N ALORS R ←.FAUX. ; FINI IS
SI X=Y ALORS
   X ← 0
```

```
   SINON Z ← X ; X ← Y ; Y ← Z
IS
```

4. LA STRUCTURE CAS

Le traitement d'un nombre de cas supérieur à 2 peut entraîner une imbrication d'instructions SI rapidement illisible ; On évite ceci par le recours à la structure CAS. Elle a la structure suivante :

```
CAS (suite d'instructions) $
```

Le signe $ est la parenthèse fermante associée aux délimiteurs d'ouverture de bloc.

L'instruction FINI, rencontrée dans la suite entre CAS et $ signifie que l'objectif visé dans cette structure est atteint que les calculs correspondants sont finis, et donc que l'on peut passer à la suite. C'est équivalent à un ALLER EN qui renverrait à la fin du bloc.

Exemple : soit j,m,a une date donnée par le jour j, le mois m et l'année a. Soit l[m] la longueur du mois m. La date du lendemain est donnée par

```
CAS
   SI j < l [m] ALORS j ← j+l ; FINI IS
   SI m < 12 ALORS j ← l ; m←m+l ; FINI IS
   j←l ; m←l ; a←a+l
$
```

Cette structure peut être utilisée pour rendre plus claire la structure du programme. Si par exemple il faut calculer une valeur f pour une variable x, nulle quand x est nul, résultant d'un calcul compliqué si x est non nul, on écrira.

```
CAS
   SI x=0 ALORS f ← 0 ; FINI IS
   ....<calcul de f> …
$
```

5. L'ITÉRATION

Il y a 3 formes de boucles possibles.

5.1. La boucle POUR

```
POUR <var> ← <exp> PAS <exp> JUSQUA <exp> FAIRE
    <suite d'instructions sans FINI à niveau 0 >
BOUCLER
```

Comme en LSE, la partie PAS est optionnelle. Si elle est absente, le pas est pris égal à 1.

On appelle "instruction FINI au niveau 0" une instruction FINI qui n'est pas englobée dans une autre boucle, ou dans une structure CAS. Autrement dit, on ne peut sortir de la boucle autrement que par épuisement de la liste de valeurs décrites dans l'intitulé. Ceci est une restriction faite de propos délibéré : la suite décrite dans l'intitulé sera toujours intégralement traitée. A la différence de LSE, les expressions figurant dans PAS et JUSQUA sont des constantes sur la boucle. Leur valeur est calculée à l'entrée, affectée aux variables PAS et LIMITE respectivement, puis utilisée pour le contrôle de la boucle.

Exemple :

```
POUR I ← 1 JUSQUA N FAIRE A [I]← 0 BOUCLER
POUR I ← 1 PAS 20 JUSQUA 100 FAIRE
  AFFICHER SCH(C,1,15)
BOUCLER
```

L'instruction SUIVANT rencontrée dans le corps de boucle a la signification suivante : les calculs correspondant au pas de boucle en cours sont finis. Passer au pas suivant.

Exemple :

```
POUR I ← 1 JUSQUA N-1 FAIRE
  SI A[I] > A[I+1]
     ALORS X ← A[I] ; A[I]← A[I+1] ; A[I+1]← X
  IS
BOUCLER
```

est équivalent à :

```
POUR I←1 JUSQUA N-1 FAIRE
  SI A[I] <= A[I+1] ALORS SUIVANT IS
  X ← A[I] ; A[I] ← A[I+1] ; A[I+1] ← X
BOUCLER
```

L'instruction SUIVANT n'ajoute aucune puissance d'expression. Elle permet seulement d'éviter de trop fortes imbrications de tests.

5.2. La boucle TANT QUE

```
TANT QUE <exp.booléenne> FAIRE
    <suite d'instructions sans FINI à niveau 0>
BOUCLER
```

Les conditions d'écriture sont les mêmes ; pas de sortie extraordinaire. Cette boucle est effectuée tant que l'expression conditionnelle est vraie.

L'instruction SUIVANT a la même signification.

5.3. La boucle FAIRE

```
FAIRE
  < suite d'instructions avec FINI à niveau 0 >
BOUCLER
```

L'instruction FINI signifie que l'objectif de la boucle est atteint et donc que l'on peut et doit sortir de la boucle. Exemples : consultation linéaire de table

```
I ← 1 ;
FAIRE
  SI I>N ALORS FINI IS
  SI A[I]=X ALORS FINI IS
  I←I+1
BOUCLER.
SI I>N ALORS
   traitement du cas pas dans la table
   SINON cas où x est dans la table
IS
```

Recherche du dernier blanc dans une ligne :

```
i←0
FAIRE
   {il Y a un blanc en i dans c}
   ip←pos(c,i+1,' ')
   SI ip=0 ALORS FINI IS
   i←ip
BOUCLER
```

L'instruction SUIVANT est utilisable, avec la même signification ; nota : les instructions FINI et SUIVANT peuvent être considérées comme des ALLER EN. Mais il y a deux différences fondamentales :

- il n'y a pas d'étiquette, ce qui délivre de la gestion des numéros de ligne, et supprime des risques d'erreur.
- dans l'esprit des programmeurs, l'instruction ALLER EN répond en général à la question : que vais-je faire maintenant ? Elle met l'accent sur l'action. Le mot FINI répond à la question : où en suis-je ? Quelle est la situation actuelle ? Si c'est l'assertion de fin de boucle alors c'est fini, et l'on écrit FINI. Il en va de même pour le mot SUIVANT : si la situation actuelle est l'invariant de boucle, alors on peut passer au cas suivant. Toutefois, le mot SUIVANT ne me parait pas encore l'idéal, et je me réserve de le remplacer par un participe passé indiquant clairement que l'assertion invariante de boucle est à nouveau vraie.

6. LA PROGRAMMATION DESCENDANTE

Ceci est inspiré d'Aladin. Il faut pouvoir rédiger un programme de façon descendante, c'est-à-dire y faire figurer des instructions symboliques qui seront détaillées plus tard. Une telle instruction est représentée par un identificateur entre doubles apostrophes ; Pour améliorer la présentation, l'identificateur peut être suivi d'un blanc, puis d'un texte quelconque ayant valeur de commentaire.

L'action ainsi symbolisée sera plus tard détaillée dans un bloc. En voici un exemple :

```
M←N
FAIRE
  SI M = 1 ALORS FINI IS
  "CHER cher la position k du max sur 1..m "
  " ECH anger les éléments k et m "
  M←M - 1
BOUCLER
…
BLOC "CHER"
   K←1
   POUR I 2 JUSQUA M FAIRE
     SI A [I] > A[K] ALORS K ← I IS
   BOUCLER
$"CHER"
BLOC "ECH"
   X←A[I] ; A[I]←A[K] ; A[K]← X
```

```
$"ECH"
```

Le bloc n'est pas une procédure sans paramètre. Dans la compilation de LSE83 (pour le moment en LSE) le corps du bloc est copié à la place de l'appel du bloc. Ainsi, aucun supplément de temps d'exécution n'est dû au mécanisme de bloc.

Le mot FINI dans le corps d'un bloc dit que l'on a atteint l'objectif du bloc, et envoie donc en séquence après le bloc. Un bloc peut en appeler un autre, mais il ne saurait y avoir de récursivité sur les blocs, à cause du mécanisme de recopie : on entrerait dans un cycle infini de recopies.

7. FONCTIONS

Une fonction (procédure produisant un résultat) est définie par :

```
FONCTION &nom(<liste de paramètre formels>) | LOCAL <liste var> |
```

La liste de paramètres formels peut être vide. La partie LOCAL peut être omise. Tout paramètre formel d'une fonction est automatiquement utilisé par sa valeur. Il ne peut y avoir de paramètre résultat dans une fonction. Ceci ne suffit pas à empêcher les effets de bord dus aux fonctions, mais en limite l'importance ou le risque. Le mot RESULTAT définit la valeur obtenue pour la fonction et dit du même coup que le calcul de la fonction est fini.

Le corps de la fonction est délimité par l'en-tête et le signe $ suivi du nom de la fonction. Il n'y a aucune raison d'emboîter les déclarations de fonction.

Exemple :

```
FONCTION &SGN(X)
  SI X>0 ALORS RESULTAT 1 IS
  SI X=0 ALORS RESULTAT 0 IS
  RESULTAT -1
$&SGN

FONCTION &DERPO(C,D) LOCAL I,IP
{la dernière occurrence de D dans c}
IP←0
FAIRE
  {dernière occurrence rencontrée de 0 dans C en IP}
  I←POS(C,IP+1,D) ;SI I=0 ALORS FINI IS
  IP←1
BOUCLER
```

```
RESULTAT IP
$&DERPO
```

Les déclarations de fonction ne sont pas hiérarchisées : elles sont toutes faites au même niveau. Il n'y a pas de déclaration de fonction dans le corps d'une autre fonction ou d'un sous-programme. Ceci veut dire qu'il ne peut *y* avoir de variable globale à un groupe de fonctions.

8. SOUS-PROGRAMMES

Un sous-programme a pour effet le plus courant de modifier des variables du programme principal, soit en les définissant, soit en leur donnant de nouvelles valeurs à partir de celles qu'elles possèdent. En ce sens, ils élargissent l'affectation :

I←0 ou I←I+2

On appelle "paramètre résultat" un paramètre défini par le sous-programme, paramètre "mixte" ceux qui sont modifiés mais dont la valeur à l'appel de la procédure sert de donnée à la procédure. Les autres paramètres sont des données (non modifiées).

Les paramètres résultats ou mixtes sont nommés en premier et suivis du signe ←. S'il y en a, les paramètres données viennent après. Un sous-programme est déclaré par SSP. Comme pour les fonctions, il peut *y* avoir une partie LOCAL. Les paramètres figurant après le signe ← , s'il *y* en a, sont pris par valeur. Exemples :

```
SSP &AFLI(A,N) LOCAL I
AFFICHER ' '
  POUR I←1 JUSQUA N FAIRE
    AFFICHER [U] A [I]
  BOUCLER
FINI
$&AFLI
```

Cette procédure d'affichage n'a que des paramètres don- nées.

```
SSP &ECH(A,B← ) LOCAL X
  X←A ;A←B ;B←X
  FINI
$&ECH
```

Note : FINI marque que l'action du SSP est finie (et renvoie par conséquent au programme principal. La situation a priorité sur l'action : FINI et non RETOUR).

Dans le sous-programme &ECH il n'y a que des paramètres mixtes :

```
SSP &DAF(A,B←C,D)
  A←C ;B←D
  FINI
$&DAF
```

Des appels de ce sous-programme sont :

```
&DAF(A,B← B,A)
```

qui échange les valeurs de A et B pourvu qu'elles soient de même type :

```
&DAF(X,Y←X+Y,X-Y)
```

qui met en X la valeur de la somme de X et Y et en Y celle de leur différence.

Le signe ← est utilisé pour bien marquer la nature des sous-programmes.

9. EXCEPTIONS

C'est un point très délicat. On munit LSE83 du plus petit dispositif possible. La politique sur ce point pourrait évoluer, notamment lors de l'écriture du système complet en LSE83.

Le problème est le suivant : certaines fonctions ou sous-programmes ne peuvent être rêa1isês parce que les données sont des cas d'exception. Il faut alors abandonner l'exécution en cours, et reprendre en un point particulier du programme.

Rencontrée dans un SSP l'instruction ABANDON provoque :

- l'arrêt des calculs en cours,
- la sortie de toutes les procédures en cours,
- l'exécution d'un bloc particulier du programme principal délimité par les parenthèses ERREUR $.

Dans l'exécution normale du programme principal, le bloc ERREUR est sauté. Il ne peut être exécuté que par ABANDON.

Exemple : on lit une expression. On en extrait les sous-expressions entre parenthèses en les remplaçant par de nouvelles variables (dont le

nom est formé de Z et d'un numéro). Si l'expression est mal parenthésée, on l'indique et on demande une nouvelle expression.

```
CHAINE C, EXC
FAIRE
  AFFICHER 'expression : ' ;LIRE C
  SI C=' ' ALORS FINI IS
  N←1
  FAIRE
    &BAL(I,J←C)
    SI J>LGR(C) ALORS FINI IS
    AFFICHER 'Z' ,N,' = ',SCH(C,I+1,J-I-1)
    C←MCH(C,I,J+1-I,'Z'!CCA(N))
    N←N+1
  BOUCLER
  AFFICHER C
  ERREUR
     AFFICHER 'MANQUE ', EXC
  $
BOUCLER
TERMINER
SSP &BAL(P,Q←C)
{donne les positions P et Q dans () les plus intérieures de C}

   P←0
   FAIRE
    {SI P #  0, pointe une ( }
     Q←PTR(C,P+1,'()')
     SI Q>LGR(C) ALORS &RR(P#0,')') ; FINI IS
     SI SCH(C,Q,1) - ')' ALORS &RR(P=0,'(') ; FINI IS
     P ← Q
   BOUCLER
   FINI
$&BAL
SSP &RR(C,D)
   SI NON C ALORS FINI IS ;
   EXC←D ;ABANDON
$&RR
```

Si l'expression est correctement parenthèsée, la boucle du programme principal s'achève normalement, on affiche C, le bloc ERREUR est sauté et on passe à une nouvelle expression.

Dans le cas contraire, on entre dans &RR, venant d'une boucle du programme principal, dans laquelle est appelé BAL, puis d'une boucle de BAL. On sort de la procédure BAL (ce qui ferme la boucle en cours dans BAL), puis on sort de RR. On revient dans le programme en ERREUR pour afficher la nature de l'erreur et passer au cas suivant.

10. TYPES DE DONNÉES

Dans un premier temps, je propose de garder les types LSE. C'est une solution de moindre transformation. A titre personnel, ils me conviennent raisonnablement. L'expérience d'enseignement montre qu'ils sont acceptables au niveau du lycée. Les points contestables sont lès suivants :

10.1. *Déclaration implicite des réels.* Non distinction entre entiers et réels

Ce qui milite en faveur de la déclaration de toute variable, c'est la nécessité de préciser toujours de quoi l'on parle Mais il me semble qu'alors il faudrait aller plus loin : les déclarations devraient être le lexique commenté des variables du programme.

Ainsi les déclarations du programme précédent pourraient être quelque chose comme :

```
CHAINE C expression traitée,
       EXC caractère manquant,
NOMBRE N numéro de variable disponible,
       I pointeur de ( ,
       J pointeur de ).
```

C'est très bavard ...

Dans un tel système, on peut supprimer la déclaration TABLEAU. Toute variable donnée avec une liste de valeurs entre crochets est déclarée comme un tableau du type correspondant, les bornes supérieures des indices étant données par les valeurs de la liste.

10.2. Absence de types construits

On peut en LSE ou LSE83 représenter aisément de très nombreux types de données. Mais c'est à la charge du programmeur, et dispersé dans le programme. L'emploi de procédures permet de limiter cet inconvénient.

Mais une bonne utilisation des types est une affaire complexe, et je ne suis pas convaincu qu'elle relève du lycée.

S'il s'agit d'augmenter les contrôles effectués sur les programmes, je plaiderai contre cette solution. D'une part, il faut habituer les programmeurs à être responsables de leurs actes. S'ils écrivent des bêtises, il leur faudra les retrouver, sans attendre qu'un système omniprésent les détecte pour eux. Par ailleurs, un tel contrôle pénalise les bons programmeurs. On vérifie sans arrêt qu'ils n'ont pas écrit de stupidités.

La seule vraie question est pédagogique : que perd-on à ne pas avoir de tels types en LSE ? C'est un débat ouvert. Au demeurant, le langage n'est pas figé. Appel aux suggestions!

11. RÉALISATION ACTUELLE

Le système LSE83 est écrit en LSE. Il est donc immédiatement utilisable sur tout ordinateur muni du LSE, à des adaptations près. Il a été implémenté sur THEMIS de EFCIS-THOMSON (64K, microprocesseur 8 bits Motorola 6800), mais avec un système très compact qui laisse un espace important pour le programme LSE d'une part, l'espace de travail de l'autre. Il faudra voir sur d'autres machines s'il n'apparaît pas de contraintes venues d'un espace insuffisant pour le programme ou les chaînes.

Le système est en 3 modules :

11.1. L'éditeur

Ce module assure l'entrée du programme, ainsi que ses modifications. Le programme est lu ligne à ligne. Chaque ligne est analysée pour vérifier une partie de sa syntaxe (structure des instructions, validité des expressions. Mais, par exemple, on ne contrôle pas les types, ni la conformité des arguments d'une fonction LSE. Simple affaire de place. Le programme est déjà volumineux). Une ligne incorrecte est refusée, avec message d'erreur en clair (ERREUR MANQUE). Elle peut être corrigée par l'éditeur de ligne LSE. Aucun contrôle de contexte n'est fait (vérification

de la structure d'une instruction SI ou FAIRE qui peut être éclatée sur plusieurs lignes). Une ligne entrée sous le numéro d'une ligne déjà présente se substitue à celle-ci. Une ligne correcte est insérée à sa place dans le programme déjà entré.

Le programme est stocké en mémoire comme une unique chaîne de caractères, partiellement codée : tous les mots clefs du LSE83 sont codés (SI, ALORS, FAIRE, BLOC...), les autres restant inchangés. Les blancs en tête de ligne sont compactés sur un seul caractère.

L'éditeur gère les commandes, qui sont actuellement :

LIster * (tout, comme en LSE)
n (une seule ligne)
n A p (lignes n à p, comme en LSE)

Les lignes sont listées en clair (restitution des blancs en tête de ligne et des mots clefs). La liste s'arrête à la fin de chaque page. En tapant Return, on relance la liste. Mais on peut alors taper une commande quelconque qui arrête la liste, puis est effectuée.

DEcoder <nom> décode le programme, et le stocke en fichier ASCII sous le nom donné. Il peut alors être traité par tout autre programme LSE ou même système (en particulier, éditeur de texte extérieur à LSE).

ENcoder <nom> prend un texte dans le fichier ASCII défini par son nom, et le prend comme entrée pour l'éditeur. Le texte est lu sur disquette ligne à ligne, et codé en mémoire. Il peut être travaillé alors par l'éditeur LSE83.

TRanslater |* |n| n1 A n2 | DE p

Suivant les cas, tout le programme, la ligne n, ou les lignes de n1 à n2 ont leur numéro translaté de p, positif ou négatif. Si une ligne translatée reçoit le numéro d'une ligne existante, elle la remplace.

EFfacer |* |n| n1 A n2 | comme en LSE

SElectionner lignes avec < suite de caractères >

Cette commande cherche dans le programme toutes les lignes où figure une occurrence de la chaîne de caractères qui soit ni précédée ni suivie d'un caractère alphanumérique (lettre ou chiffre). C'est un peu plus général que la recherche d'un identificateur. On peut chercher l'identificateur I (non précédé ou suivi d'une lettre ou d'un chiffre, ce qui

élimine les I de SI ou AFFICHER). Mais on peut aussi chercher un nom de fonction ou sous-programme, ou A I …

SAuvegarder & <nom> range le texte de la fonction ou du SSP &nom (connu parce qu'il y a le signe fermant $&nom) sur disquette, en translatant ses lignes pour ramener la première au numéro 0.

ATtacher &<nom> <numéro> prend sur disquette la fonction ou le SSP &nom, et l'ajoute au texte du programme en translatant ses numéros de ligne (commençant en 0) de la valeur donnée par le deuxième argument.

EXécuter. Cette commande lance la compilation (enchaînée avec l'exécution) du programme LSE83. La chaîne codée est garée en fichier temporaire sur disquette, et le module compilateur est lancé.

11.2 Le compilateur

Il travaille sur la chaîne codée (qu'il reprend sur disquette). Il remplace les structures LSE83 par des structures LSE équivalentes. Ceci implique une gestion des numéros de lignes LSE, cette fois significatifs. Ainsi :

```
POUR … FAIRE
```
est traduite en
```
FAIRE n POUR
```

Or le numéro de la ligne fermant cette boucle est inconnu au moment de la compilation de la ligne. Il n'a pas paru opportun de garder en mémoire le texte codé et le texte compilé. On met donc des numéros symboliques. Chaque ligne compilée est aussitôt sortie dans un fichier ASCII.

Lorsque un numéro symbolique se trouve défini, il est entré avec son équivalent dans une chaîne de caractères. Celle-ci servira au troisième module pour la production du texte définitif.

Lorsque le compilateur détecte un appel de bloc, il suspend la compilation pour entreprendre celle du texte du bloc, puis à la reconnaissance de la fin de bloc ($") reprend où il l'avait laissée la compilation du programme. Ceci rompt le caractère séquentiel de la compilation et interdit de lire le texte source ligne à ligne et de le traduire aussitôt.

Le plus délicat est la compilation de l'instruction LSE83 SI … Elle peut en effet s'étendre sur plusieurs lignes, et n'être alors traduisible que par les ALLER EN

```
SI t ALORS a SINON b IS
```

avec pour a et b de longues suites d'instructions devrait être traduit en :

```
      SI NON t ALORS ALLER EN n1
      a
      ALLER EN n2
n1    b
n2    ......
```

La négation de t peut poser des difficultés (ce qui avait été proposé en épreuve libre en 1984 à Paris ne couvrait pas tous les cas possibles. Cela fait beaucoup d'instructions LSE, et il y a ,un problème de place). Nous avons utilisé la forme équivalente, mais moins élégante.

```
      SI t ALORS ALLER EN n1 ; ALLER EN n2
n1    a
      ALLER EN n3
n2    b
n3 .......
```

Mais il ne faut pas faire ceci si la traduction est possible en une ligne LSE (de 256 caractères, sans blancs. Il n'y a pas ici à soigner la présentation). Le système détecte donc les SI courts qui peuvent être traduits en

```
      SI t ALORS DEBUT a FIN SINON DE BUT b FIN
```

ou

```
      SI t ALORS DE BUT a ; ALLER EN n FIN
      b
n ...
```

Pour les procédures, le compilateur sait que tout paramètre formel d'une fonction doit être ajouté à la liste LOCAL si elle existe, ou mis dans une liste ajoutée à la déclaration : Dans le cas d'un SSP, il rajoute en LOCAL tout ce qui est à droite du signe s'il existe, tout si ce signe n'existe pas. Rien de bien difficile.

La boucle FAIRE la plus générale est traduite par des ALLER EN. Les boucles POUR et TANT QUE sont traduites en boucle similaires de LSE, même s'il y a une différence fondamentale dans le cas de POUR LSE considère les quantités derrière PAS et JUSQUA comme des variables

recalculées à chaque pas de la boucle, alors que LSE83 les prend comme constantes. Dans le meilleur des cas, c'est une perte de temps. Dans le pire des cas (une variable figurant dans PAS ou JUSQUA est modifiée dans la boucle) c'est une erreur. On pourrait l'éviter (préaffectation de PAS et JUSQUA à de nouvelles variables avant d'entrer dans la boucle). Ceci n'a pas paru urgent, le cas pathologique relevant d'une stylistique à déconseiller : tant pis pour l'imprudent qui prend une file d'autoroute à contresens!

Le parenthésage des structures LSE (FAIRE... BOUCLER, SI . . .IS CAS $ etc.) est vérifié par le compilateur. Si une erreur est détectée, elle est affichée. ERREUR MANQUE BOUCLER LIGNE 280

Le contrôle est rendu à l'éditeur, qui affiche la ligne incriminée après avoir ramené en mémoire le programme codé, sur lequel on pourra travailler pour le corriger, puis relancer EXécuter. C'est un mécanisme lourd, lié à la nécessité de structurer le système LSE83 en modules (taille mémoire insuffisante). Ceci se traduit concrètement par un temps d'attente (plusieurs dizaines de secondes entre la détection de l'erreur et la possibilité d'intervenir). C'est un moindre mal. La solution est bien évidemment dans l'écriture d'un système intégré en langage d'assemblage. C'est un assez gros travail (bien que l'on puisse récupérer de grands morceaux de LSE usuel). Mais cela ne s'impose pas pour le moment. On peut fort bien travailler avec le système écrit en LSE : je n'utilise plus que LSE83 depuis un an et demi.

11.3. Le numéroteur

Quand le compilateur a terminé son travail, il gare sur fichier temporaire la chaîne de définition des numéros de ligne symboliques, puis appelle le numéroteur, qui fait revenir cette chaîne, lit ligne par ligne le texte compilé provisoire, y remplace les numéros symboliques par les numéros réels, remet tout sur fichier ASCII. Enfin, il rend le contrôle au système LSE pour lui faire ENTRER puis EXECUTER le texte qu'il vient de produire. Ceci est très rapide.

12. EXEMPLES

12.1.. Anagramme itérative

Ce programme dit si la chaîne B est anagramme de A (mêmes caractères mais dans un autre ordre).

```
1 CHAINE A,B,BP
```

```
10 FAIRE
15    "LEC ture des données A et B"
20    CAS
22 SI LGR(A)#LGR(B) ALORS ANAG←.FAUX. ; FINI IS
25     I←1
30 FAIRE {tous les caractères de A de 1 à I-1 retirés de B}
32  SI I>LGR(A) ALORS ANAG←LGR(B)=0 ; FINI IS
34       J←POS(B,1,SCH(A,I,1))
36    SI J=0 ALORS ANAG← .FAUX. ;FINI IS
38    B←MCH(B,J,1,")
39   I←I+1
40     BOUCLER
49 $
50
60 "AFFichage du résultat"
65
70 AFFICHER 'AUTRE COUPLE ?' ; LIRE A
71 SI A #'OUI' ALORS FINI IS
72 BOUCLER
79 TERMINER
80 BLOC "LEC"1
81 AFFICHER 'PREMIERE CHAINE ' ; LIRE A
82 AFFICHER 'DEUXIEME CHAINE ' ; LIRE B ; BP B
83 $"LEC"
89
90  BLOC "AFF"
91   SI ANAG ALORS AFFICHER A, 'EST ANAGRAMME DE ',BP
92    SINON AFFICHER A, 'N"EST PAS ANAGRAMME DE ',BP
93  IS
94  $ "AFF"
99 $
```

On notera la puissance du mécanisme FAIRE FINI.

La boucle intérieure possède deux sorties, et ne peut être mise sous la forme d'un TANT QUE de PASCAL que par un artifice. Il n'est pas possible de commander ce TANT QUE par le Booléen ANAG dont la valeur peut être aussi bien VRAI que FAUX à la sortie de la boucle. Une forme possible est :

```
incertain :=VRAI
TANT QUE incertain FAIRE
```

```
SI i>lgr(a)ALORS DEBUT incertain :=FAUX ; anag    :=lgr(b)=0
 FIN
 SINON DEBUT
   j :=pos(b,l,sch( a, i ,l))
   SI j=0 ALORS DEBUT incertain := FAUX
                      anag :=FAUX
         FIN
     SINON DEBUT
        b :=mch(b,j,l,")
        i :=i+l
        FIN
  FIN
```

L'utilisation de FINI d'une part, des parenthèses fermantes IS de l'autre permet une écriture beaucoup plus simple et plus claire. Je ne suis pas PASCALIEN !

12.2. Anagramme récursive

```
1 CHAINE A,B,BP
5 FAIRE
10 AFFICHER 'A = ' ;LIRE A ; SI A='' ALORS FINI IS "
11 AFFICHER 'B = ' ;LIRE B ; BP←B
12
15 R SI LGR(A) # LGR(B) ALORS .FAUX. SINON &ANAG(A,B) IS 18
20 SI R ALORS AFFICHER A, 'EST ANAGRAMME DE ',BP
21 SINON AFFICHER A, 'N"EST PAS ANAGRAMME DE 1 ,BP
22 IS
25 BOUCLER
29
30 TERMINER
31
50 FONCTION &ANAG(U,V) LOCAL J {lgr(u)=lgr(v)}
51    SI U=' ' ALORS RESULTAT .VRAI. IS
52    SI J = 0 ALORS RESULTAT .FAUX. IS
54 RESULTAT &ANAG(SCH(U,2, ' '),MCH(V,J,1, ' '))
55 $&ANAG
99 $
```

13. CONCLUSION

J'ai d'abord écrit ce système pour moi. J'aime bien LSE, mais j'en avais assez de certaines de ses contraintes. Je programme plus vite en LSE83.

Ce qui a été fait est presque immédiatement disponible sur tout micro-ordinateur possédant LSE. Reste à savoir s'il y a des amateurs. Dans la mesure où la rédaction du système LSE83 en LSE ne relève pas de l'acrobatie, et ne représente que 360 lignes LSE, il est possible de modifier le langage, l'éditeur, ou le système.

Je suis ouvert à toutes les suggestions, au moins pour les discuter : le meilleur langage n'est pas le plus gros. On peut changer, retrancher. Les additions doivent être fortement discutées avant d'être acceptées. Si vous désirez LSE83, ou si vous le souhaitez avec modifications, écrivez-moi.

Jacques ARSAC
École Normale Supérieure
45 rue d'Ulm 75005 PARIS