

Rpl : leçon n°2 *Hp 49*

Maintenant que vous savez à quoi ressemble un programme, regardons ce qu'il y a dedans avec un large aperçu des structures du Rpl...

Pour résumer très brièvement ce qu'on a raconté dans le numéro précédent, un programme correspond à une suite d'instructions qui aura pour unique but de réaliser une action (et en particulier des applications). Aussi, chaque programme est construit entre les délimiteurs « et » sachant que les instructions devront se trouver au milieu. Et puis, comme nous travaillons dans un environnement Rpn, il ne faudra pas oublier d'utiliser au maximum les possibilités que nous offre la pile...

• les variables •

C'est dans les variables qu'on stocke des objets temporaires pendant l'exécution d'un programme. Dans ces variables, on peut y stocker ce que bon nous semble. Les variables peuvent être globales (utilisables par tous les programmes) ou locales (utilisables uniquement dans un programme).

Pour créer une variable globale, on procède comme pour un programme : on pose le contenu de la variable sur la pile, puis son nom (qu'on appelle aussi nom global) et on se sert de la touche [STO] pour la stocker. Et ensuite, on peut en faire ce qu'on en veut. A titre d'exemple, créons la variable TST dans laquelle on va stocker l'entier 25, la séquence est la suivante : 25 'TST' [ENTER] suivi de [STO].



La variable avant le [STO]...



... et juste après : la voilà dans le menu !

Après un appui sur [VAR], on s'aperçoit que notre variable fait partie des objets déjà stockés. Maintenant, on peut par exemple lui infliger une équation du type $TST = TST / (1 + TST^2)$, et le programme devient tout simplement :

```
« TST 1 TST SQ + / ' TST' STO »
```

On aurait pu aussi faire ce programme de manière à utiliser la pile un peu plus :

```
« TST 1 OVER SQ + / ' TST' STO »
```

On peut stocker l'un ou l'autre programme sous le nom ESSAI (ou le nom de votre choix), et après une première exécution, on peut rappeler la valeur de TST pour voir que la variable a effectivement évolué.

Passons aux variables locales. Comme elles ne sont utilisées qu'à l'intérieur d'un programme, elles ont une structure particulière :

```
« objet_1 objet_2 ... objet_N '' variable_1 variable_2 ... variableN  
« (corps du programme) » »
```

Dans cette structure, l'objet n°1 sera stocké dans la variable n°1, l'objet n°2 dans la variable n°2, et ainsi de suite. Prenons un exemple où il faut calculer $z = x^3 y - x^2 y^3$, le programme pourrait être :

```
« '' X Y « X 3 ^ Y * X Y 3 ^ * - "Z" ''TAG » »
```

Ici, l'objet posé sur le niveau 2 de la pile ira dans X et celui du niveau 1 dans Y. Et le résultat sera posé sur le niveau 1 après exécution.

• les tests •

Les tests permettent de choisir la réalisation d'une action en fonction d'une condition. Et le résultat de cette condition est nécessairement un booléen, c'est à dire 0 ou 1. Exemple, si un nombre entier est divisible par 2, alors il est pair, sinon il est impair : voilà un test concret. Avant d'en donner le programme, voici les différentes structures qu'on peut rencontrer :

```
• I F (condition)  
  THEN (suite de commandes)  
  END
```

Cette structure correspond à un test de type : SI (condition) ALORS (on exécute une suite de commandes) FIN. C'est le test le plus simple qu'on puisse faire : selon le résultat, on réalise ou non la suite de commandes.

```
• I F (condition)  
  THEN (suite de commandes)  
  ELSE (autre suite de commandes)  
  END
```

Cette structure est la plus complète puisqu'elle permet de choisir entre deux types d'action : si le résultat de la condition est vrai on exécute le premier bloc de commandes, sinon on lance le second bloc. C'est plutôt une structure du type : SI (condition) ALORS (on exécute une suite de commandes) SINON (on exécute une autre suite de commandes) FIN.

```
• (condition) « (suite de commandes) » I FT
```

Cette structure est la forme réduite de la structure I F THEN END.

```
• (condition) « (suite de commandes) » « (autre suite de commandes) » I FTE
```

Cette structure est la forme réduite de la structure I F THEN ELSE END.

```
• CASE  
  (condition 1) THEN (suite de commandes 1) END  
  (condition 2) THEN (suite de commandes 2) END [...]  
  (condition N) THEN (suite de commandes N) END  
  (autre suite de commandes, ceci est optionnel)  
  END
```

C'est une structure qui permet de réaliser différentes actions en fonction des tests demandés. C'est bien pratique dans des cas particuliers (voir le programme en fin d'article), mais c'est assez peu utilisé.

Les tests utilisent les opérateurs logiques suivants : == < < % > \$ SAME NOT AND OR XOR. D'autres tests assez spécifiques sont aussi disponibles dans le manuel de l'utilisateur.

Revenons à notre exemple qui va déterminer la parité d'un entier. Le programme (qu'on nommera PARITE) pourrait être construit comme ceci :

```
« I F 2 MOD  
  THEN " Impai r"  
  ELSE " Pai r"  
  END »
```

N'oublions pas de poser un entier sur la pile avant de lancer le programme, sinon une erreur surgira ! Une méthode plus élégante consiste à écrire ceci :

```
« 2 MOD " Impai r" " Pai r" I FTE »
```

Voici un autre petit exemple (57 octets) qui permet de connaître le maximum de deux nombres sans utiliser la fonction MAX intégrée à la Hp 49 :

```
« '' A B  
  « I F A B > THEN A ELSE B END » »
```

Simplifions les choses au maximum (27,5 octets) pour obtenir ceci :

```
« I F DUP2 < THEN SWAP END DROP »
```

Voyons ce que ça peut donner avec une structure à la IFT (35 octets) :

```
« DUP2 < « SWAP » IFT DROP »
```

Modifions un peu ce programme pour obtenir ceci (27,5 octets) :

```
« DUP2 < { SWAP } IFT DROP »
```

• du concret •

Pour conclure ce chapitre, passons à une application concrète qu'on rencontre très souvent quand on est lycéen : la résolution d'un polynôme du second degré. Nous n'allons pas utiliser les commandes internes, mais nous allons plutôt créer un petit programme qui n'utilisent que les tests et qui affiche le résultat de manière fort sympathique. Comme chacun sait, pour déterminer les racines d'un polynôme, il suffit de calculer le discriminant (qu'on appellera D) avant d'en extraire les racines (réelles ou complexes). Si la forme d'un polynôme du second degré est $ax^2+bx+c=0$, alors la formule de D est b^2-4ac . Si D est nul, alors il n'y aura qu'une racine qui sera égale à $-b/2a$. Si D est positif, alors il y aura deux racines réelles qui seront $x1=(-b-\text{SQRT}(D))/2a$ et $x2=(-b+\text{SQRT}(D))/2a$, avec SQRT (square root) qui représente la racine carrée. Si D est négatif, alors nous avons deux racines complexes qui seront $z1=(-b-i*\text{SQRT}(-D))/2a$ et $z2=(-b+i*\text{SQRT}(-D))/2a$. Le programme devient tout naturellement :

```
« "aX²+bX+c=0" MSGBOX
  "Entrer a" "" INPUT STR
  "Entrer b" "" INPUT STR
  "Entrer c" "" INPUT STR
  `` A B C
  « B SQ 4 A C * * - `` D
  « D "D" ``TAG
  CASE
  D O == THEN
    B NEG 2 A * / "X" ``TAG END
  D O > THEN
    B NEG D f + 2 A * / EVAL "X1" ``TAG
    B NEG D f - 2 A * / EVAL "X2" ``TAG END
  D O < THEN
    B NEG i D NEG f * + 2 A * / EVAL
    "Z1" ``TAG DUP CONJ EVAL "Z2" ``TAG END
  END »
»
»
```

En regardant ce programme, on se rend compte ici qu'on pourrait gagner un peu de place en simplifiant ce qui est répétitif. Le plus flagrant est $B \text{ NEG } 2 A *$ qui sont utilisés quel que soit le test réalisé. Une solution serait de les calculer avant et de s'en servir ensuite dans les différents tests. On ne réécrit pas tout le programme, voici juste l'essentiel de ce qui a changé :

```
[ . . . ]
« D "D" ``TAG
  2 A * B NEG
  CASE
  D O == THEN
```

```
SWAP / "X" ``TAG END
D O > THEN
  D f DUP2 4 PICK / EVAL "X1" ``TAG
  UNROT - ROT / EVAL "X2" ``TAG END
D O < THEN
  i D NEG f * + SWAP / EVAL
  "Z1" ``TAG DUP CONJ EVAL "Z2" ``TAG END
```

[. . .]

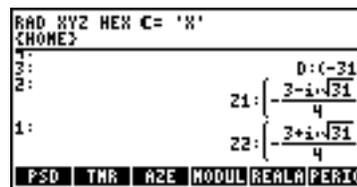
Cette petite opération, qui permet d'utiliser un peu plus la pile, nous fait gagner la bagatelle de 39 octets, ce qui n'est pas négligeable. A titre d'information, I NPUT est une aide à la saisie, MSGBOX affiche une chaîne de caractères dans une boîte à message, NEG est la fonction qui se cache sous la touche [+/-] et qui inverse le signe d'un nombre, ``TAG permet de coller une étiquette à un objet, CONJ donne la forme conjuguée d'un nombre complexe, et EVAL permet ici de simplifier les expressions. Voyons maintenant comment se présente notre programme dans les différents cas :



Résolution de $x^2+2x+1=0$



Résolution de $x^2-7x+3=0$



Résolution de $2x^2+3x+5=0$

Cette application mathématique permet de comprendre assez facilement le fonctionnement des tests. A titre d'exercice, vous pouvez toujours essayer de transformer ce petit programme à la sauce I FT ou I FTE.

• fin de la leçon •

Le mois prochain, nous attaquerons les boucles, partie du Rpl sans doute la plus intéressante. Une bonne brochette d'applications est également prévue...