

L'Axe Parser

Par kindermoumoute ,
Matrefeytontias
et nikitouzz



www.openclassrooms.com

*Licence Creative Commons 4.0
Dernière mise à jour le 28/03/2013*

Sommaire

Sommaire	2
Lire aussi	2
L'Axe Parser	4
Partie 1 : Les débuts en Axe	5
Information et installation	5
Présentation	5
La naissance d'un langage	5
Avantages et désavantages	5
Les outils	6
Notion de compilation	7
Mon premier programme	7
Créer le programme source	8
Afficher du texte	8
Faire une pause	9
Variables et calculs	9
Les variables et quelques notions de mathématiques	10
Calculs booléens	11
Exercice	13
Résultat	14
Les conditions	15
Explication de la condition	15
Les structures conditionnelles classiques	15
? ?? une autre structure !	17
Les conditions décrémentées	18
Les boucles	18
La boucle Repeat	19
La boucle While	19
La boucle For	20
EndIf et End!If	20
Le getKey	21
Explication	22
Le getKey en action	23
Exercice	24
Résultat	24
TP n°1 : la pluie de numéro	26
Quelques consignes	26
Les commandes à connaître	26
Correction	27
Les ajouts possibles	29
Partie 2 : Notions de programmation avancées	32
Les pointeurs	32
La théorie	32
La pratique	32
Évolution possible	33
Les fonctions	34
Les labels	35
Appeler une fonction	35
Les variables temporaires	37
Exercice d'application	38
Dessiner c'est gagner !	39
Plusieurs écrans en un seul	39
Pixels et Géométrie	40
Le troisième écran	42
Les Datas 1/2	43
Le binaire	43
Apprenons à compter	43
Le binaire	43
L'hexadécimal	44
L'hexadécimal par rapport au décimal	44
L'hexadécimal par rapport au binaire	45
Une histoire d'octets	46
Les Datas : de simples données	46
Les Datas 2/2	48
Mon premier sprite	48
Le tilemapping	50
création de la map	50
Affichage de la tilemap	52
Encore plus de Data !	53
Les listes	56
présentation des listes	56
Les tableaux	57
D'autres commandes utiles	57

TP n°2 : en quête de l'échec	59
Présentation du jeu	59
Quelques commandes et conseils utiles	60
Correction	61
Les ajouts possibles	72
Partie 3 : Repoussons les limites !	81
Les caractères ASCII et les tokens	81
Les caractères ASCII	81
Les tokens	82
La commande input	83
Les niveaux de gris	85
3 niveaux de gris	85
4 niveaux de gris	86
Le tilemapping avec grayscales	87
Création de la map	87
Optimiser son code	91
Généralités	92
Le registre HL et ses usages	94
Le dilemme entre la taille et la vitesse du programme	96
Autres conseils	97
Quelques commandes d'optimisation	97
Spécificités du compilateur	98
Manipuler les variables de la calculatrices	99
Appvars et programmes	100
Manipuler une appvar	100
Manipuler un programme	102
Les "vraies" variables et plus encore	102
Les variables cachées	103
Maîtriser les fonctions	106
Un peu plus sur les arguments	106
L'adresse des fonctions	107
Lambda	108
La récursivité	111
Partie 4 : Annexes	113
Utilisation de TI-Connect	113
Installer TI-Connect	113
Brancher sa calculatrice	114
Brancher sa TI-83+	114
brancher sa TI-84+	117
Transférer ses programmes	118
Autres outils	120
Créer une image pour sa TI	120
Tableau des erreurs et émulation	122
Les erreurs	122
Utiliser un émulateur	122
Liste des commandes	124
Système	125
Ecran et mémoire tampon	126
Blocs de contrôle	128
Labels et fonctions	130
Math (base)	131
Math (avancé)	133
Dessin	134
Sprites	136
Texte	137
Data et stockage	140
Variables externes	143
Interruptions	144
Port de liaison	145
Autres	145



L'Axe Parser



Par

kindermoumoute et



Matrefeytontias et



nikitouzz

Mise à jour : 28/03/2013

Difficulté : Intermédiaire

Durée d'étude : 1 mois

Vous possédez une calculatrice TI-83+ ou TI-84+ (Silver Edition comprise) ? Vous voulez apprendre à programmer dessus ?

Ne partez pas, vous êtes au bon endroit 😊, dans ce cours je vais vous présenter comment faire toutes sortes de programmes sur votre calculatrice, grâce à l'Axe Parser (prononcez axe parseur).



Mais de toute façon les programmes sur calculatrices sont lents et moches.

FAUX, vous découvrirez ici qu'on peut les faire non seulement rapides, mais également beaux.
Quelques exemples de programmes faits en Axe :



A gauche le jeu pokemon TI (de finale TI) et à droite le jeu axe snake (de ztrumpet)

Où est l'utilité de faire cela ? Sans compter que vous pourrez faire de beaux programmes pour votre calculatrice 😊, vous aurez des bases solides pour programmer dans d'autres langages par la suite (sur ordinateur par exemple 😊).

Donc, l'Axe Parser se résume à deux choses : facilité d'utilisation et rapidité.

A la fin de ce tuto, vous serez en mesure de :

- Manipuler des sprites 8*8 et faire des cartes.
- Manipuler les variables propres à la calculatrice (comme les appvars, les Pic, les String, etc.).
- Faire des jeux multijoueurs via un câble.
- Faire apparaitre 3 niveaux de gris ou 4 niveaux de gris.
- Diffuser du son sur des écouteurs.
- Gérer les multi-touches.
- Gérer le contraste.
- Dessiner à l'aide de quelques fonctions de géométries.
- et bien plus encore...

Le tout à la même vitesse que n'importe quel programme fait en ASM.



Le tutoriel est régulièrement mis à jour. Dernière version : 1.1.2.

Partie 1 : Les débuts en Axe

Bon, oui j'ai dit que l'Axe Parser est simple d'utilisation, cependant, pour que cela paraisse ainsi, il faut des bases... et je n'ai jamais dit que ces bases étaient simples à apprendre 😊.

Dans cette partie vous apprendrez à faire des calculs, afficher du texte, repérer si une touche a été pressée, etc. Rien n'est très compliqué, mais tout est nécessaire à connaître pour la suite.

Information et installation

Avant de commencer à programmer en Axe, il faut savoir dans quoi vous vous lancez. Tout d'abord, ce cours s'adresse :

- Aux personnes qui possèdent leur première calculatrice programmable (généralement les secondes).
- A tous les passionnés de calculatrices cherchant à les exploiter au mieux.
- Aux professeurs qui cherchent un moyen d'enseigner des notions poussées d'algorithmie dans leurs cours.

De plus, je tiens à rappeler que l'on peut coder en Axe uniquement sur les calculatrices z80 munies de mémoire flash (ROM), qui sont :

- Ti 83+(SE)
- Ti 84+(SE)

 Mais si ma calculatrice n'est pas un de ces modèles, ça veut dire que je ne peux vraiment rien faire avec l'Axe Parser ?

Pas tout à fait, il est encore possible de développer ses programmes à partir d'émulateurs (voir [annexe](#)).

 Il est également possible d'émuler une TI-84+ sur une TI-Nspire (non-CAS uniquement), on pourra donc programmer en Axe dessus.

Présentation La naissance d'un langage

Pour programmer sur calculatrice à processeur z80, il existe 2 langages officiels :

- **Le TI-Basic** : programmable sur la calculatrice (ou utilitaires PC). Ce langage est très simple à apprendre, mais la vitesse d'exécution des programmes TI-Basic est excessivement lente (pour les jeux en tous cas).
- **L'Asm z80** : programmable sur calculatrice (depuis peu) et surtout à partir d'utilitaires PC. Ce langage est très compliqué à apprendre, pour un résultat beaucoup plus puissant qu'un programme TI-Basic.

La nécessité d'un troisième langage s'imposait. Plusieurs tentatives plus ou moins abouties existent (voir tableau plus bas), mais celle qui ressort de plus en plus du lot est l'**Axe Parser**.

 Le mot **axe** signifie *hache* (l'arme), et le mot **parser** est un terme pour désigner le compilateur.

 Lorsque l'on parle de l'Axe Parser, on parle de l'application (du compilateur), mais lorsque l'on parle du langage, on parle de l'**Axe**

A seulement 19 ans, Kevin Horowitz (alias Quigibo), étudiant en génie électrique et informatique, a eu l'idée de créer un langage qui allait mettre tout le monde d'accord. Le premier février 2010 sort une première version de l'Axe Parser. C'est un succès sur le forum [omnimaga](#) qui va en faire un de ses intérêts principaux.

Dès lors, les versions du compilateur se succèdent et se perfectionnent (encore aujourd'hui). Les possibilités de ce langage sont effarantes pour un résultat indiscutables.

Avantages et désavantages

	TI-Basic	xLIB/Celtic	BBC Basic	Grammer	Asm z80	Axe
Difficulté du langage	Facile	Facile	Moyen	Moyen	Difficile	Moyen
Vitesse	Lent	Moyen	Rapide	Très Rapide	Très rapide	Très rapide
Éditable sur la calculatrice?	Oui	Oui	Avec un éditeur spécial	Oui	Avec un éditeur spécial	Oui
Exécution	Interprété	Interprété	Interprété	Interprété	Compilé	Compilé
Support des sprites ?	Non	Oui	Oui	Oui	Oui	Oui
Variable nécessaire pour être exécuté	Pic, Lists, Strings,...etc	Pareil qu'en Basic avec 16ko d'application en plus	16Ko d'application	49Ko d'application	Aucun	Aucun
Compatible avec les shells ?	Oui	Quelques	Aucun	Aucun	Oui	Oui
Spécialité	Math	Jeux	Varié	Varié	Tout	Jeux principalement
Voir le code source	Toujours	Toujours	Toujours	Toujours	Optionnel	Optionnel

Il n'y a pas de langage de programmation parfait. Chacun a ses avantages et inconvénients. C'est à vous de décider ce qui convient le mieux à vos besoins. Si vos priorités sont la rapidité, la facilité d'utilisation, et la capacité à faire beaucoup de choses, alors l'Axe Parser est fait pour vous.

Seulement, à chaque bon côté en Axe, il y a un mauvais côté (bon ok, il y a quand même plus de bons cotés 😊). Il arrive de temps en temps que votre code contienne une erreur où vous risquez le *Ram Cleared* ou le *freeze de la calculatrice*. Il est même

possible de corrompre la mémoire flash de la calculatrice (mais là, faut le chercher 😊).

Image utilisateur

Ici on peut admirer un magnifique ram cleared sur une TI 84+SE (tout ce qu'il y a de plus classique).

```
RAM FREE 21694
ARC FREE 1573K
ABCDEF 67
FEDCBA 877
▶ Pic1 767
▶ Pic1 778
L1 12
L2 12
```

Ici on a affaire à deux variables Pic1, un prodige que seul l'Axe Parser peut expliquer.

Heureusement sur TI-83+ et TI-84+ vous disposez d'une mémoire flash vous permettant d'archiver vos programmes :



Pendant tout le tutoriel, j'utiliserai des touches franco-anglaises des calculatrices TI-83+/84+, créées par critor (il y a passé des heures et des journées croyez moi💡). Retrouvez les différents claviers en intégralité : [ici](#).

Il est également recommandé de bien sauvegarder vos codes sources sur un ordinateur en cas de problèmes. De plus, vous pouvez utiliser un émulateur (voir [annexe](#)).

i Autre chose encore : un programme écrit en Axe sera environ 1.5 à 2 fois plus grand qu'un même programme écrit en Asm z80.

Les outils

Tout d'abord il faut télécharger la dernière mise à jour : [ici](#). Dézippez le fichier zip.

Puis on va un peu décortiquer de quoi il est composé :

- Un dossier *Developers*
- Un dossier *Examples*
- Un dossier *Tools*
- Un fichier **ACTUALLY READ ME.txt**
- Un fichier *Auto Opts.txt*
- Un fichier *Axe.8xk*
- Un fichier *ChangeLog.txt*
- Un fichier *Commands.html*
- Un fichier *Documentation.pdf*
- Un fichier *keycodes.png*

Ce dont on va avoir besoin pour l'instant est le fichier **Axe.8xk**. Mettez-le sur votre calculatrice.

? Euh.. comment je peux mettre un fichier de mon ordi sur ma calculatrice ? 😊

Une annexe est prévue pour ça en fin de tutoriel : [ici](#).

i Si vous n'y arrivez pas, n'hésitez pas à poster vos questions sur les forums indiqués en fin de tutoriel.

Maintenant il vous suffit d'aller dans le menu des applications avec la touche **angle B apps**, puis vous verrez Axe rajouté dans la liste. Démarez-le, et vous devriez voir ça :



Pour se déplacer dans les menus de l'application, on utilisera les flèches , pour sélectionner

i on utilisera **2nde 2ND** ou **précéder résol entrer** et pour quitter ou revenir au menu précédent on peut utiliser **annul CLEAR** ou **quitter QUIT mode**

Maintenant nous allons nous intéresser aux réglages. Allez dans *option*, et là un sous-menu apparaît :



Shell : Permet de choisir pour quel shell on compilera le programme source, on peut soit mettre :

- **No shell** : Compilera pour aucun shell, le programme s'exécutera via la commande *Asm(prgmMONPROG)*.
- **Ion** : compilera le programme pour Ion.
- **Mirage OS** : compilera le programme pour Mirage OS.
- **Doors CS** : compilera le programme pour Doors CS.
- **Application** : compilera le programme sous forme d'application.

Alpha : Permet d'activer ou non les minuscules (lowercase en anglais).

Safety : Permet de sauver le code source lors de la compilation en cas d'éventuels bugs du programme (très utile !).

Back : Retour au menu principal



Un shell est un programme ou une application ayant une interface améliorée pour démarrer des programmes ASM prévus pour ce shell. Ici les programmes seront compilés à peu près comme pour l'ASM, donc on laisse le choix du type de shell.

Si vous préférez utiliser un shell, je vous conseille l'application **noshell** qui permet d'exécuter les programmes ASM qui ne nécessitent pas de shell, ainsi que certains nécessitant un shell (MirageOS et Ion notamment). Il s'installe via le menu de l'application et les programmes s'exécutent comme normalement dans le menu programme (archivé ou non).

Si néanmoins vous préférez ne pas utiliser de shell, il vous faudra exécuter vos programmes avec la commande *Asm* suivie de votre programme. La commande *Asm* se trouve ici :

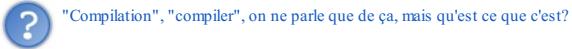
puis **entrée**

Maintenant retournez dans le menu principal, puis allez dans *compile*. Pour l'instant vous n'avez peut-être aucun programme affiché, mais il peut y en avoir de 2 sortes :

- Le premier est un source prêt à être compilé.
- Le deuxième est une sauvegarde de source près à être restaurée (toujours précédé par le symbole dièse # devant lui).



Notion de compilation



On a vu plus haut que les programmes sur la TI peuvent être soit "interprétés", soit "compilés".

Seulement, pour que les actions comme "allumer un pixel à tel endroit" ou encore "stocker la valeur 4" soit réalisée, il faut que la calculatrice puisse lire un langage qui est propre à elle, le langage machine.

Un programme est interprété quand la calculatrice lit, ligne par ligne, le code source du programme, l'interprète en langage machine et l'exécute en même temps. C'est le cas du Ti-Basic.

Autres exemples de langages interprétés : le Javascript, le PHP, le Python, etc.

En revanche, un programme est compilé lorsque le code source est **préalablement transformé en langage machine**, afin d'être directement exécuté par la TI. La calculatrice n'a plus besoin de lire ligne par ligne votre code pour l'interpréter et ensuite l'exécuter ; un programme compilé est donc en général bien plus **rapide** ! ☺

Autres exemples de langages compilés : le C, le java, etc.

En Axe, c'est l'application *Axe.8xk* qui va transformer le code source en langage machine ; on l'appelle le compilateur.

Il faut savoir que lorsque vous compilez un code source contenant une erreur, une erreur de compilation apparaît.



Nous verrons l'utilité du contenu du reste du fichier zip plus tard (mais je ne vous interdis pas de le regarder ☺).

Bon, je ne vous retiens pas plus, rendez-vous au prochain chapitre !

Mon premier programme

Maintenant que vous avez bien réglé vos options préférées, on va pouvoir commencer à programmer en Axe (enfin ! 😊).

Dans ce chapitre vous apprendrez à afficher du texte à l'écran. C'est un début, mais c'est de fil en aiguille que vous allez apprendre à programmer en Axe, il est donc nécessaire de commencer par le commencement. 😊

Créer le programme source

Créez un nouveau programme  , puis indiquez le nom de votre programme source.

Ensuite sur la première ligne de ce programme, mettez un point suivi du nom de votre programme exécutable (8 caractères maximums) :

Code : Axe

```
PROGRAM:AXESOURC  
:.AXEEXEC
```

Ici j'ai créé le programme AXESOURC et lorsqu'il compilera il créera AXEEXEC.

A partir de maintenant, votre programme est visible et compilable dans le menu de compilation de l'application vu précédemment. Il faut aussi savoir que certaines commandes de l'Axe Parser sont des commandes de TI-Basic dont le nom a été modifié, par exemple si vous appuyez sur la touche  , vous verrez ►Char à la place de ►Frac

Autre chose avant de commencer à programmer : quand vous allez compiler votre source non archivé et que vous avez une erreur de compilation, vous pouvez voir à quelle ligne se trouve précisément l'erreur en appuyant sur la touche .

Afficher du texte

Commençons par ce qu'il y a de plus simple, écrire du texte.

Il existe 3 commandes pour afficher du texte en Axe, celles-ci sont très similaires au TI-Basic mais ne vous fiez pas aux apparences :

- La commande *Text* (   0) et la commande *Text* (   6)

Code : Axe

```
:Text (X, Y, "MON TEXT  
:ou  
:Text (X, Y  
:Text "MON TEXT
```

Image utilisateur

La commande *Text*(affichera directement à un point (X,Y) le texte, alors que la commande *Text* nécessite qu'on ait préalablement mis un curseur pour savoir où placer le texte, cela peut être utile dans certains cas.

 Le X et le Y seront ici des nombres entre 0 et 95 pour X, et 0 et 63 pour Y. Cette commande est l'équivalent du Text(en TI-Basic, le texte sera placé en fonction des pixels de la calculatrice.

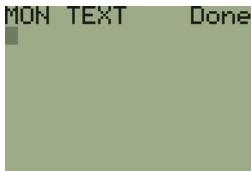


Vous remarquerez que le " et la) disparaissent à la fin de la ligne, on peut donc écrire aussi bien *Text(X,Y,"MONTEXT")* que *Text(X,Y,"MONTEXTE")* ; c'est en fonction de vos préférences.

- La commande *Disp* (   3)

Code : Axe

```
:Disp "MON TEXT
```

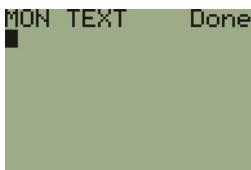


La commande *Disp* va afficher le texte en haut à gauche de l'écran. Quand le texte est trop long, il saute la ligne. Quand on enchaîne deux *Disp* dans un code, le deuxième s'affichera à la suite du premier.

- La commande *Output* (   6)

Code : Axe

```
:Output (X, Y, "MON TEXT
```



Ici on peut placer à un point donné notre texte.



Le X et le Y seront ici des nombres entre 0 et 15 pour X, et 0 et 7 pour Y. Cette commande est l'équivalent du *Output(* en TI-Basic.



J'ai compilé et démarré mon programme, mais il n'a rien affiché.

Oui, c'est possible, surtout si vous utilisez un shell (comme MirageOS ou Ion), cela vient du fait que le programme a affiché le texte, mais qu'il s'est ensuite éteint, il faut donc mettre une pause.

Faire une pause



J'ai fait une pause, mon programme a fait apparaître le texte comme il faut, mais la calculatrice s'est freezée et il a fallu que j'enlève les piles

Rah.. vous pouviez pas attendre que j'explique? 😤

Donc, je disais qu'il fallait faire une pause pour nous laisser le temps de voir, seulement la commande *Pause* utilisée en TI-Basic n'a pas la même utilisation en Axe!

La commande *Pause* se trouve ici :



La pause en Axe est une durée pendant laquelle le programme va être arrêté. Cependant le nombre indiqué doit être au minimum de 1, donc les codes suivants ne sont pas bons :

Code : Pas bien

```
:Pause
```

Code : Pas bien

```
:Pause 0
```

Une seconde correspond environ à un *Pause 1800*. Donc deux secondes de pause reviendrait au code suivant :

Code : Axe

```
:Pause 3600  
:.Car 1800 fois 2 = 3600 = 2 seconde
```



💡 Pourquoi ce commentaire apparaît dans le code ?

Comme dans tout langage qui se respecte, l'Axe Parser possède une commande pour inclure des commentaires dans le source. Pour commenter sur une seule ligne, on placera un point "." en début de ligne, sauf la première réservée au nom de l'exécutable. On peut également faire des commentaires sur plusieurs lignes. Pour cela il faut commencer la première ligne par "..." et terminer la dernière de la même façon :

Code : Axe

```
...  
:Ici un commentaire  
:sur plusieurs lignes  
:...
```

Ce qu'il faut retenir de ce chapitre, c'est surtout qu'il y a plusieurs commandes pour afficher du texte, et qu'il ne faut pas oublier qu'elles ont chacun des avantages qui fait qu'on les utilisera plus dans certaines situations que dans d'autres.

Rien de difficile encore, mais je vous préviens... le prochain chapitre relève un peu la barre. 😊

Variables et calculs

Que ce soit pour le nombre de vies dans un jeu, les nombres des coordonnées X et Y sur un plan, les points gagnés, etc. Tous ces nombres sont des valeurs que l'on manipule grâce aux variables. Dans tout langage de programmation il y a des variables, on ne peut pas y échapper.

Les variables et quelques notions de mathématiques

Des maths ? je peux m'en aller ?

Ahhh je vous ai enfin piégés !

Mais non ne vous inquiétez pas, ce ne sera pas bien compliqué. Il est nécessaire de comprendre quelques notions pour la suite des choses, donc autant s'y attaquer dès le début !

Les opérations de base et les variables

Une variable est une valeur que l'on peut modifier durant un programme. Celle-ci est enregistrée dans un endroit très précis dans la calculatrice que l'on vera plus loin dans le cours.

Pour définir une variable, rien de plus simple :

Code : Axe

```
: 0→A
```

Ici on a donné 0 comme valeur à la variable A.

→ apparaît en appuyant sur cette touche :



Maintenant je veux donner 0 aux variables A et B :

Code : Axe

```
: 0→A→B
```

Vous avez remarqué qu'il n'est pas nécessaire de réécrire à chaque fois la valeur 0, mais on peut faire mieux encore. Si je veux donner comme valeur 4 à A et 8 à B :

Code : Axe

```
: 4→A+4→B  
:.Ou encore  
:4→A*2→B  
:.Dans le même genre  
:8→B/2→A  
:8→B-4→A
```

Jusque là ça va, non ?

Maintenant si je veux faire respectivement une addition, une soustraction, une multiplication, une division et un modulo :

Code : Axe

```
: 1+1→A      : .A=2  
: 2-1→A      : .A=1  
: 2*2→A      : .A=4  
: 5/5→A      : .A=1  
: 4^3→A      : .A=1
```

Modulo? 🤔 kesako?

Le modulo est une opération mathématique qui permet d'obtenir le reste d'une division euclidienne, donc ici $4/3 = 3*1+1$, reste 1.

On ne peut stocker dans une variable qu'un nombre entre 0 et 65 535, donc en faisant l'opération $0\text{-}1\rightarrow A$, vous obtiendrez 65 535. Nous étudierons pourquoi plus loin dans le cours.

D'autres opérations

Maintenant on va voir quelques opérations plus complexes, et toujours utiles à savoir :

La valeur absolue :

Code : Axe

```
:abs(A)→A  
:.Si A est négatif, il devient positif
```

Vous trouverez l'opération `abs` ici :



Il faut savoir que c'est utile que pour les nombres compris entre -32768 et 32767, donc 65535 sera considéré comme -1 :

Code : Axe

```
:abs(65535)→A
:..A=1
:abs(A-101)→A
:..A=100
```

Le maximum des deux expressions :
Code : Axe

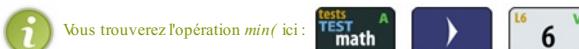
```
:12→A+6→B
:max(A,B)→C
:.Ici C=B car B=18
```



Le minimum des deux expressions :

Code : Axe

```
:12→A+6→B
:min(A,B)→C
:.Ici C=A car C=12
```



Le carré de l'expression :

Code : Axe

```
:3→A
:A^2→A
:.A est égale à 9
```



La racine de l'expression :

Code : Axe

```
:9→A
:√(A)
:.A=3
```



⚠ ATTENTION : les calculs en Axe n'ont aucune règle de priorité !

N'ignorez pas cette ligne car cela pourrait vous causer plein de problème par la suite. 🍪

En effet, lorsque l'on fait le calcul $1+1*2$ sur sa calculatrice, on obtient 3, mais en Axe on obtiendra 4. D'où vient le problème ? En fait l'Axe Parser lit le calcul comme il vient, sans prendre compte des règles de priorités... SAUF les parenthèses quand il y en a.

Donc pour avoir le bon résultat de $1+1*2$ en Axe on écrira $1+(1*2)$, mais le plus simple reste d'**organiser son calcul** comme cela : $1*2+1$.

Calculs booléens



Oui encore du calcul, mais celui ci est plus simple à comprendre, car il n'y a que 2 résultats possibles dans un calcul booléen (prononcer "boule et un") :
0 (on dira que c'est FAUX)
1 (on dira que c'est VRAI)

Cette étape est **très importante** pour comprendre les chapitres suivants.

Les symboles tests

Un symbole test va en fait tester soit l'infériorité, soit la supériorité, soit l'infériorité/égale, soit la supériorité/égale, soit l'inégalité ou encore l'égalité de 2 expressions. Si le test de ces deux expressions est VRAI, il renverra 1. Si il est FAUX, il renverra 0. (**tout compris ? 🤓**)

Je suis sûr que vous comprenez mieux avec ce code, en commençant avec le symbole égale :
Code : Axe

```
:2→A+1→B
:.A vaut 2 et B vaut 3
:A=B→C
```

```
:.C vaut 0 car A n'est pas égal à B
```



Note : le menu des symboles test " $=, <, >, \neq, \leq, \geq$ " se trouve ici :



Maintenant avec d'autres symboles :

- Strictement inférieur
Code : Axe

```
:2->A+8->B
:.A vaut 2 et B vaut 10
:A<B->C
:.C est égal à 1 car A est plus petit que B
```

- Strictement supérieur
Code : Axe

```
:2->A+8->B
:A>B->C
:.C est égal à 0 car A n'est pas plus grand que B
```

- Inférieur ou égale
Code : Axe

```
:5->A->B
:.A et B valent 5
:4->D
:.D vaut 4
:A\leq B->C
:.C est égal à 1 car A est égal à B
:D\leq B->C
:.C est égal à 1 car D est plus petit que B
```

- Supérieur ou égal
Code : Axe

```
:5->A->B
:4->D
:A\geq B->C
:.C est égal à 1 car A est égal à B
:D\geq B->C
:.C est égal à 0 car D n'est pas plus grand que B
```

Maintenant, petit test, trouvez quelle est la valeur de X dans ce code :

Code : Axe

```
:2->A*4->B^3->C
:A<C->D\geq 0->X+B->X
```

Vous ne trouvez pas ? 🤔

Voici la correction :

Secret (cliquez pour afficher)

D'abord il faut comprendre quelles sont les valeurs que l'on a données à A, B et C :

Code : Axe

```
:2->A*4->B^3->C
:.A vaut 2
:.B est égal à  $2 \times 4$  soit 8
:.8 modulo 3 vaut 2, donc C vaut 2
:.(8 divisé par 3 a pour quotient 2 car il y a 2 fois 3 dans 8, mais il a pour
(2*3)) )
```

Ensuite on calcul avec attention :

Code : Axe

```
:A<C->D\geq 0->X+B->X
:.A vaut B (=2), A n'est pas plus petit que 0 (c'est FAUX) donc D vaudra 0
:.D égal 0 (c'est VRAI) donc X est égal à 1
:.Seulement on rajoute la valeur de B à X (X+B) comme B vaut 8, 8 plus 1 est é
```

Les opérateurs logiques

Un opérateur logique est un moyen de comparer plusieurs expressions testées, par exemple :

- and compare si les deux expressions sont vraies : EXP1 and (EXP2).



On n'est pas obligé de mettre des parenthèses dans la première expression, mais les suivantes nécessitent obligatoirement celles-ci.

Code : Axe

: 0->B
: (A>6) and (B=A) ->C
: (A≤0) and (B=A) ->D

EXP1 est égal à A>6 et EXP2 à B=A
 A>6 est FAUX (0 n'est pas plus grand que 6)
 B=A est vrai (0 est égal à 0)
 Les deux expressions ne sont pas vraies, C=0.
 A≤0 est vrai (0 est plus petit OU égale à 0)
 B=A est vrai (0 est égale à 0)
 Les deux expressions sont vraies donc D=1



Comme les expressions renvoient 0 ou 1, on peut faire un tableau de toutes les réponses possibles !

Ce tableau, appelé tableau de vérité, est le suivant pour l'opérateur *and* :

0	and	0	=	0
1	and	0	=	0
0	and	1	=	0
1	and	1	=	1

Un petit schéma pour expliquer le *and* :



En remplaçant les 2 cercles par 2 expressions on peut mieux visualiser le *and* dans la zone rouge.

- *or* cherche à savoir si au moins une expression est vraie : EXP1 or EXP2
Code : Axe

: 0->B
: (A>6) or (B=A) ->C
: (A≤0) or (B=A) ->D

C est égal à 1 car la deuxième expression est vraie, B=A (à partir du moment où on remarque que l'une des expressions est vraie, on n'a même pas besoin de vérifier la deuxième car il en faut au moins une de VRAIE).
 D est égal à 1 car les 2 expressions sont vraies.

Tableau de vérité :

0	or	0	=	0
1	or	0	=	1
0	or	1	=	1
1	or	1	=	1

Petit schéma :



- *xor* cherche à savoir si et seulement si une seule des conditions est vraie :
Code : Axe

: 0->B
: (A>6) xor (B=A) ->C
: (A≤0) xor (B=A) ->D

C est égal à 1 car la deuxième expression est vraie (B=A) et la première est fausse.
 D est égal à 0 car ici les deux expressions sont vraies !

Tableau de vérité :

0	xor	0	=	0
1	xor	0	=	1
0	xor	1	=	1
1	xor	1	=	0

Et le schéma :



Tout cela paraît compliqué, mais en réalité c'est de la logique appliquée avec des 0 et des 1.

Exercice

Ce n'est pas le tout de lire, mais pour comprendre il n'y a pas meilleur moyen que de pratiquer.

Ici, l'exercice va être d'afficher 0 quand A est égal à 1 (quand A est VRAI), et 1 quand A est égal à 0 (quand A est FAUX) avec un seul calcul qui aura le rôle d'inverseur. Vos deux résultats seront affichés une seconde chacun au centre de l'écran à peu près.



Euh.. afficher une variable ?

Oui, je ne vous ai pas encore donné la commande pour afficher une variable. Il suffit de mettre la variable sans guillemets suivie de la commande ►Dec, par exemple pour afficher la variable A avec la commande Disp :

Code : Axe

```
:Disp A►Dec
```

donc je récapitule :

Code : Autre

```
:On initialise A à 0
:On fait le fameux calcul qui donnera la valeur 1 à A
:On affiche A vers le milieu de l'écran
:On attend une seconde
:On refait le fameux calcul qui cette fois donnera 0 à A
:On affiche A une ligne en dessous du premier résultat (toujours vers le centre)
:On attend une seconde encore
```

Résultat

Maintenant que vous avez bien cherché et bien trouvé le résultat 🎉, je vais vous montrer les possibilités qui s'offrent à vous :

Secret (cliquez pour afficher)

Tout d'abord le principal code sans le calcul :

Code : Axe

```
:0→A
:.Calcule→A
:Output(6,3,A►Dec
:Pause 1800
:.Même calcul→A
:Output(6,3,A►Dec
:Pause 1800
```

La Pause 1800 pour une seconde et la commande *Output*(car elle sera plus facile à utiliser que *Text*(dans ce cas (il suffit de rajouter 1 à l'axe Y pour sauter une ligne).

Pour le calcul, on pouvait soit faire avec un symbole :

Code : Axe

```
:A<1→A
:...
:Quand A vaut 1, il n'est pas plus petit que 1, donc il vaut 0
:Quand A vaut 0, il est plus petit que 1 donc il vaut 1
:...
:A=0→A
:.Et c'est le même principe ici aussi
:A≠1→A
:.ainsi que là
:A≤0→A
```

On pouvait également faire par une astuce de calcul :

Code : Axe

```
:1-A→A
```

Quand A est égale à 1, le calcul fait 1-1=0. Quand A est égale à 0, le calcul fait 1-0=1.

La solution retenue

La façon la plus optimisée de faire ce calcul est d'utiliser la commande *xor*. En effet on a vu que *0 xor 1* vaut un et *1 xor 1* vaut zéro :

Code : Axe

```
:A xor 1→A
```

Donc voici le code final :

Secret (cliquez pour afficher)

Code : Axe

```
:0→A
:A xor 1→A
:Output(8,4,A►Dec
:Pause 1800
:A xor 1→A
:Output(8,5,A►Dec
:Pause 1800
```

Relisez bien ce chapitre, c'est LE chapitre important pour cette première partie. Une fois que vous aurez compris l'intérêt des variables et des booléens, vous aurez compris beaucoup de choses. 🎉

Les conditions

On retrouve les conditions dans beaucoup de langages, mais que sont exactement les conditions ? C'est ce que nous allons essayer de comprendre durant ce chapitre.

Explication de la condition

Une condition sert à tester une variable. Dans votre vie de tous les jours vous êtes confrontés à de nombreuses conditions, plus ou moins sympathiques, par exemple :

Citation

Si j'ai donné deux euros à la boulangère, alors j'ai du pain.

Ou encore :

Citation

Si je n'ai pas donné deux euros à la boulangère, alors je n'ai pas de pain.

Cette deuxième condition va être une conséquence de la première et vice versa.

Donc dans un programme de jeu ça pourrait donner :

Code : Algorithme

```
Si je meurs
Alors j'ai perdu la partie
```

Ici en Axe avec V qui est égal à 1 s'il reste une vie, ou V est égal à 0 s'il ne reste aucune vie :

Code : Axe

```
:If V=0          ::La condition commence ici ou on teste si V=0
:Disp "Game Over"   ::Si cette expression est VRAIE alors on affiche "Game Over"
:End            ::N'oubliez pas le End pour marquer la fin
```

Toute structure conditionnelle se termine par un End. Celui ci est présent pour signaler à notre programme que notre condition est terminée, et que le code reprend son cours normal. Si il n'y a pas de End, ou qu'il y a un End en trop, vous aurez une erreur **ERR:BLOCK** lors de la compilation.

On trouvera la commande *If* ici : 
 Et la commande *End* là : 

? "V=0" ? ça ressemble drôlement à des booléens !

C'est là que je voulais en venir, le principal intérêt sera d'utiliser des booléens dans des conditions, et croyez moi ça facilite vraiment la vie ! 

Les structures conditionnelles classiques

On a vu précédemment les conditions les plus courtes :

Code : Axe

```
:If (Ma Condition)
:le code à exécuter
:End
:..toujours terminer par un End!
```

i Note : *Ma Condition* sera donc une opération booléenne!

If...Else...End

Maintenant, si l'on veut exécuter un code quand la condition est vraie, et un autre quand elle est fausse, on utilisera Else de la façon suivante :

Code : Axe

```
:If (maCondition)
:..le code à exécuter si la condition est VRAI
:Else
:..sinon on exécute le code suivant
:.."code suivant"
:End
:..(on oublie pas le End)
```

i On trouvera Else ici : 

? Je vais devoir mettre des conditions partout alors ? 

Oui ! C'est un programme !
 Mais je vous rassure, on peut organiser ces conditions, grâce à une autre commande : *Elself*.

If...ElseIf...Else...End

Elseif sera utilisé comme ceci :

Code : Axe

```
:If (maPremièreCondition)
:.le code à exécuter si la condition est vraie
:Elseif (maDeuxièmeCondition)
:.sinon si la deuxième condition est vraie, on exécute le code suivant
:."code suivant"
:Else
:.sinon ce code est exécuté.
:End
```

Par exemple je reprends notre système de vie :

Code : Axe

```
:If V=0
:Disp "Game over"
:Elseif V=1
:Disp "Il ne vous reste plus qu'une vie !"
:Else
:Disp "Vous êtes en parfaite santé"
:End
```

En fait les conditions vérifient si les tests booléens sont vrais, donc en reprenant nos symboles tests :

Code : Axe

```
:If A=0
:.Mon code
:End

:If A#0
:.Mon code
:End

:If A<0
:.Mon code
:End

:If A>0
:.Mon code
:End

:If A≥0
:.Mon code
:End

:If A≤0
:.Mon code
:End
```

Puis les comparateurs logiques :

Code : Axe

```
:If (A=0) and (B#8)
:.Mon code
:End

:If (A<0) or (B>8)
:.Mon code
:End

:If (A≥0) xor (B≤8)
:.Mon code
:End
```

Une autre particularité des booléens, on peut simplifier l'écriture suivante :

Code : Axe

```
:If A=1
:.le code s'exécutera si A est VRAI
:End
```

En celle ci :

Code : Axe

```
:If A
:.le code s'exécutera si A est VRAI
:End
```



Ici, A s'exécutera si il est plus grand ou égal à 1



Et si A est négatif, il n'y a pas un moyen plus simple que *If A#0* ?

Si, il existe les conditions inversées ! 🍪

Les conditions inversées

En fait, les conditions inversées sont exactement pareilles que les conditions normales. On rajoutera juste le point d'exclamation !



Code : Axe

```
:!If A
:.Si A est FAUX, on exécute ce code
:End
```

Il marchera donc pour toutes les variantes de la structure conditionnelle vue précédemment :

Secret (cliquez pour afficher)

Code : Axe

```
:!If (maCondition)
:.le code à exécuter si la condition est FAUX
:Else
:.sinon on exécute ce code
:End
```

Code : Axe

```
:!If V
:Disp "Game over"
:Else;if V>1
:Disp "Il ne vous reste plus qu'une vie !"
:Else
:Disp "Vous êtes en parfaite santé"
:End
```

Code : Axe

```
:!If (A=0) and (B≠8)
:.Mon code
:End

:!If (A<0) or (B>8)
:.Mon code
:End

:!If (A≥0) xor (B≤8)
:.Mon code
:End
```

?? une autre structure !



Ouch, mon code devient illisible, il y a des *If* et des *End* partout ! 😞

Sachez qu'il existe une autre structure pour diminuer la taille de vos conditions dans votre code source. Tout d'abords regardons une condition simple comme on vient de le voir :

Code : Axe

```
:If A<2
:6→A
:End
```

Ici, il y a deux choses intéressantes : $A < 2$ et $6 \rightarrow A$. On utilisera le point d'interrogation ? (



) pour réduire cette condition à :

Code : Axe

```
:A<2?6→A
```

i Concrètement le résultat sera exactement le même, c'est juste une notation différente qui est utile pour des conditions contenant qu'une ligne.

Maintenant si on veut l'équivalent avec la commande *Else* :

Code : Axe

```
:If A<2
:6→A
:Else
:3→B
:End
```

On ajoutera une virgule après la première expression :

Code : Axe

```
:A<2?6→A, 3→B
```



Est ce qu'il existe un moyen de réduire les conditions inversées ?

Oui, et c'est tout aussi simple : il suffit de mettre deux points d'interrogation :

Code : Axe

```
:A<2??3→B
```

De même avec le Else :

Code : Axe

```
:A<2??3→B, 6→A
```



Rien ne vous oblige à utiliser cette notation, elle est juste différente de la structure conditionnelle classique et permet une lecture du code plus claire lorsqu'il n'y a qu'une expression de testée.

Les conditions décrémentées

Ce type de condition est plus dur à comprendre car, en plus de tester une condition, elle va agir sur la variable testée en la décrémentant de 1 à chaque fois qu'elle est testée. Quand cette variable atteint 0, elle est restituée à sa valeur maximum que l'on aura définie.

Code : Axe

```
:DS<(VARIABLE, VALEURMAXIMUM)  
:.code1  
:End
```



La commande décrémentée DS< se trouve ici :



Un exemple de condition décrémentée :

Code : Axe

```
:DS<(X, 3)  
:.code1  
:End
```

La première fois que la condition sera testée, X=3, comme X n'est pas égal à 0 on enlève 1 à X et le code1 est ignoré, la deuxième fois X=2, pareil on enlève 1 et on ignore code1, la troisième fois X=1, on enlève 1 et on ignore toujours le code1. La quatrième fois X=0, le code1 est exécuté, et X=3.



Mais comment cette condition peut être testée plusieurs fois dans le même code? 🤔

Cela tombe bien, c'est le sujet de notre prochain chapitre : *Les boucles*.

Normalement beaucoup de choses devraient s'éclaircir dans votre tête, mais je suis sûr qu'en lisant le chapitre suivant, vous aurez la tête... encore plus clair ! 🤓

Les boucles

Une boucle consiste à répéter une partie de code en fonction d'une expression.

Omniprésentes dans tout programme, les boucles se ressemblent à travers les langages, et l'on retrouve en Axe les 3 principales boucles utilisées dans la plupart des langages :

- La boucle *Repeat*
- La boucle *While*
- La boucle *For*

La boucle Repeat

La boucle *Repeat* se traduit par : "répéter jusqu'à ce que telle condition soit vraie"

Syntaxe :

Code : Axe

```
:Repeat condition
:..code à exécuter
:End
:.On n'oublie pas le End !
```



Le code exécuté sera répété tant que la boucle n'est pas terminée.

Par exemple, je veux répéter jusqu'à ce que le nombre de vies soit égal à 0 :

Code : Axe

```
:Repeat V=0
:..code à exécuter
:End
:Disp "PERDU!"
```

Maintenant, imaginez le code pour faire un compteur jusqu'à ce que V=0, avec un départ de 5 vies.

Non, pas trouvé ?

Bon OK je vous le donne :

Secret (cliquez pour afficher)

Code : Axe

```
:5→V
:Repeat V=0
:V-1→V
:End
:Disp "PERDU!"
```

Mieux encore, on peut mettre la ligne : $V \leftarrow V - 1$ directement au début de la boucle :

Secret (cliquez pour afficher)

Code : Axe

```
:5→V
:Repeat V-1→V=0
:End
:Disp "PERDU!"
```



Et il n'y a pas, comme pour les conditions, un moyen plus simple d'exprimer V=0 (V est FAUX)?

Si, grâce à la boucle *While*.

La boucle While

La boucle *While* se traduit par : "répéter tant que telle condition est vraie".

Syntaxe :

Code : Axe

```
:While condition
:..code à exécuter
:End
:.Toujours le End !
```



Contrairement au *Repeat*, le code s'exécutera tant que la condition sera vraie :

Code : Axe

```
:While V
:..code à exécuter
:End
:Disp "PERDU!"
```

On reprend notre compteur de tout à l'heure, mais en simplifié bien sûr :

[Secret \(cliquez pour afficher\)](#)

Code : Axe

```
:5→V
:While V<1→V
:..Tant que V est différent de 0
:End
:Disp "PERDU!"
```

La boucle For

Bon, certes les boucles *Repeat* et *While* sont utiles, mais pour faire un compteur, rien de plus utile que la boucle *For*.



On trouvera la boucle *For* ici :



La syntaxe est la suivante :

Code : Axe

```
:For (Variable,ValeurInitiale,ValeurDeFinDeBoucle)
:..code à exécuter
:End
:..Et bien sûr, le End.
```

Une variable choisie va être initialisée à la valeur initiale, et la boucle va être exécutée en incrémentant 1 à cette variable à chaque fois que le code est exécuté. Lorsque la boucle est terminée, la variable utilisée sera supprimée.

On peut facilement recréer une boucle *For* avec une boucle *Repeat* :

[Secret \(cliquez pour afficher\)](#)

Code : Autre

```
:ValeurInitiale→Variable
:Repeat Variable=ValeurDeFinDeBoucle
:..code à exécuter
:Variable+1→Variable
:End
:Delvar Variable
```

Dans un exemple concret ; je veux écrire le même texte mais en sautant une ligne à chaque fois, à l'aide de la commande *Output* :

Code : Axe

```
:For (A,0,7)
:Output(0,A,"Même texte
:End
```

Maintenant avec la variable derrière, cela pourrait donner :

Code : Axe

```
:For (A,0,7)
:Output(0,A,"Choix")
:Output(6,A,A+1▶Dec)
:End
```



On aurait pu mettre le code suivant :*Output(0,A,"Choix",A+1▶Dec)*, cependant la variable va être décalée de quelques colonnes par rapport au texte, et il est donc plus utile ici de détacher les commandes pour une meilleure présentation.

EndIf et EndIf

Voici une autre manière d'utiliser les boucles *Repeat* et *While*, en mettant la condition à la fin de la boucle. Pour cela il y a les commandes *EndIf* et *EndIf* qui testent exactement de la même manière qu'une condition afin de quitter le programme ou non.

Par exemple, étudions le code suivant avec la boucle *While* :

Code : Axe

```
:While 1
:..Code
:EndIf V
```

While 1 sera toujours vrai car l'expression est égale à 1, donc la seule façon de quitter la boucle est de n'avoir plus aucun vie (ici la variable V).

Et on peut faire de même avec la boucle *Repeat* :

Code : Axe

```
:Repeat 0
:..Code
:EndIf A>50
```

Repeat 0 est toujours faux car l'expression vaut zéro, cette fois il faudra attendre que A soit supérieur à 50 pour quitter la boucle.

 Je ne comprend vraiment pas ce qu'il y a de différent à utiliser une boucle ainsi. 😕

En fait lorsque vous faites ce code :

Code : Axe

```
:1→A  
:Repeat A=1  
.Code A  
.End  
.Fin
```

Seulement la partie surlignée (en jaune) sera exécutée, et le *Code A* sera ignoré. Or si vous mettez la condition à la fin :

Code : Axe

```
:1→A  
:Repeat 0  
.Code A  
.Endif A=1  
.Fin
```

Tout le code sera exécuté, même le *Code A* qui se trouve à l'intérieur de la boucle !

Cela est du au fait que la condition teste seulement à la fin de la boucle, donc le Code A est exécuté **une fois** avant que la boucle s'arrête. 😊

Maintenant on peut faire encore mieux, par exemple en combinant des conditions au début et à la fin :

Code : Axe

```
:Repeat A<3 and (B<5  
.Code A  
.End!If A
```

D'abord on teste si A est plus petit que 3 et qu'en même temps B est plus petit que 5. Si ce n'est pas le cas on exécute le *Code A* et à la fin de la boucle on test si A est égale à zéro. Si ce n'est pas le cas la boucle recommence, et ainsi de suite. 😊

Arrivés ici, vous pensez avoir fait le tour des boucles. Pourtant, ce n'est pas fini ! Vous verrez que l'on peut faire encore plus avec les boucles : dans le prochain chapitre bien évidemment. 🎉

Le getKey

C'est bien beau, vous pouvez calculer pendant un programme, mettre des conditions, le faire tourner dans une boucle, afficher plein de choses à l'écran, etc.. Mais ça ne ressemble en rien à un jeu, pour l'instant on se contente de regarder notre programme. Le but de ce chapitre est de vous apprendre à manipuler le getKey, fameuse commande qui repère si une touche est pressée, et l'assimiler à toutes les choses que l'on a vues jusque là.

Explication

La commande `getKey`, comme en TI-Basic, repère si une touche a été pressée. On peut donc utiliser la même syntaxe qu'en TI-Basic :

Code : Axe

```
:getKey→K
```



La commande `getKey` se trouve ici :



Donc la variable K va contenir la valeur d'une touche pressée.



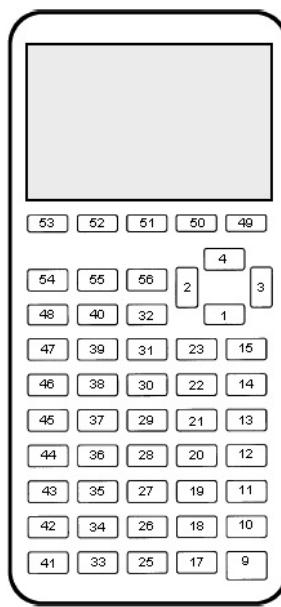
Mais comment savoir quelle valeur appartient à quelle touche ?

Souvenez-vous ! Dans le dossier zip que vous avez téléchargé au début :

Citation

- Un fichier `keycodes.png`

Je vous enlève la tâche de double cliquer dessus pour voir l'image :



Cela veut dire qu'à chaque fois que j'aurai besoin de la commande `getKey`, je devrai avoir sous la main cette image ?

Non pas forcément, car si K est une simple valeur, on peut l'afficher sur la calculatrice, avec un programme !
Je vous laisse chercher comment faire :

Après quelques secondes vous aurez trouvé ce code :

Secret (cliquez pour afficher)

Code : Axe

```
:.GETKEY
:0→K
:Repeat K
:getKey→K
:End
:Text(0,0,K►Dec
:Pause 2000
```



L'erreur ici serait d'afficher K dans la boucle, car quand aucune touche n'est pressée, K vaut 0, et il affichera toujours 0 si l'on ne s'en rend pas compte.

Mais on peut l'optimiser en utilisant dès maintenant la commande `getKey` :

Secret (cliquez pour afficher)

Code : Axe

```
:.GETKEY
:0→K
:Repeat K=15
:getKey-K
:If K
:Text (0,0,K▶Dec
:End
:End
```

Le programme affichera la dernière touche pressée, et s'éteindra quand la touche 15 aura été pressée :

annul
CLEAR

Mieux encore, `getKey→K` peut être mis directement au début de la boucle !

Secret (cliquez pour afficher)

Code : Axe

```
:.GETKEY
:0→K
:Repeat getKey→K=15
:If K
:Text (0,0,K▶Dec
:End
:End
```



Le getKey en action

Hormis les codes touches différents, c'est exactement la même commande qu'en TI-Basic?

En fait il y a 2 commandes `getKey` en Axe, la première va retourner un nombre pour une touche qui aurait été pressée (voir plus haut). Et la deuxième va retourner un booléen si une touche précise a été pressée.

Reprendons notre code pour les vies. Je veux que l'on quitte quand la touche **CLEAR** a été pressée :

Code : Axe

```
:Repeat getKey(15)
:.mon code de jeu.
:End
```

L'avantage ? Grâce à cette commande, on peut repérer si plusieurs touches ont été pressées en même temps !

Code : Axe

```
:Repeat getKey(15)
:If getKey(9) or getKey(54)
:.code
:End
:If getKey(33) and getKey(25)
:.code
:End
:End
```

 La commande `getKey()` renverra 0 si la touche indiquée n'est pas pressée (FAUX), et 1 si elle l'est (VRAI)

Mieux encore, on peut mettre le numéro de la touche dans une variable :

Code : Axe

```
:15→X
:If getKey(X)
:End
```

 Et si on fait `getKey(0)`, il se passe quoi ?

En fait, il va repérer si une touche (n'importe laquelle) a été pressée, mais il est surtout utilisé pour savoir si aucune touche n'a été pressée :

Code : Axe

```
:!If getKey(0)
:.Code à exécuter quand aucune touche n'a été pressée.
:End
```

Quoi ? vous pensiez que c'était plus compliqué que cela ? 😊 Que nenni !

Exercice

Énoncé

Pour vous échauffer pour le prochain chapitre, je vais donner un petit exercice sur la commande `getKey`.

Donc le but de l'exercice va être de faire bouger la lettre 'O' à travers l'écran de la calculatrice en fonction des touches :



Le 'O' ne doit pas sortir de l'écran (96*64 pixels je le rappelle).

Le programme se ferme en appuyant sur **annul CLEAR**.

Astuce

Avant de vous lancer, je vais vous donner une commande pour incrémenter et décrémenter vos variables de 1. En gros, $X+1 \rightarrow X$ et $X-1 \rightarrow X$ deviennent respectivement :

Code : Axe

```
:X++  
:X--
```

J'en dis pas plus, à vos caltos ! 😎

Résultat

Vous êtes sûrs d'avoir bien cherché ? 😊

D'abord, il fallait choisir la commande pour afficher la lettre '**O**'. La plus adaptée ici est la commande `Text()` car vue la rapidité du programme, vous n'auriez pas le temps de voir le '**O**' se balader avec la commande `output`. Seulement, il y a une difficulté qui se pose ici ; lorsque l'on déplace le '**O**', celui-ci va laisser des **traces** derrière lui en allant de gauche à droite et de bas en haut. Il fallait donc chercher une méthode plus ou moins efficace. Je vous propose de mettre un espace avant le '**O**' et d'en mettre 5 autres, 6 pixels en dessous des coordonnées normales (X,Y).

Donc en ajoutant la boucle qui se termine en appuyant sur **annul CLEAR**, on aura ce code :

Secret (cliquez pour afficher)

Code : Axe

```
:0→X→Y  
.On initialise les variable X et Y à 0 (X = largeur et Y = hauteur)  
.Repeat getKey(15)  
.ici la boucle repeat sera préférable  
.Text(X,Y+6," "  
.Text(X,Y," O  
.code pour modifier X et Y  
.End
```

Bon, c'est un peu tiré par les cheveux je vous l'accorde. Mais si vous avez abandonné votre programme en rencontrant ce problème, retournez à votre calculatrice et revenez quand vous aurez vraiment fini. 😎

Maintenant, on doit modifier X et Y en fonction des touches directionnelles, sans oublier les "murs" virtuels :

Secret (cliquez pour afficher)

Code : Axe

```
:If getKey(2) and (X≠0  
.X--  
.ElseIf getKey(3) and (X≠91  
.X++  
.End  
.If getKey(4) and (Y≠0  
.Y--  
.ElseIf getKey(1) and (Y≠58  
.Y++  
.End
```

En tâtonnant vous aurez sûrement trouvé les valeurs 91 (pour la largeur) et 58 (pour la hauteur), qui correspondent à la place que prend la lettre 'O'.

Donc le code final :

Secret (cliquez pour afficher)

Code : Axe

```
:0→X→Y  
.On initialise les variable X et Y à 0 (X = largeur et Y = hauteur)  
.Repeat getKey(15)  
.ici la boucle repeat sera préférable  
.Text(X,Y+6," "  
.Text(X,Y," O  
.If getKey(2) and (X≠0  
.X--  
.ElseIf getKey(3) and (X≠91  
.X++  
.End
```

```
:X++  
:End  
:If getKey(4) and (Y#0  
:Y--  
:ElseIf getKey(1) and (Y#58  
:Y++  
:End  
:End
```

Et voilà, vous avez terminé la première partie de ce cours.. ah non j'oubliais, un petit TP vous attend au prochain chapitre 😊,
relisez bien ce cours (comme d'habitude), vérifiez que vous avez du temps libre, et allez-y ! 😊

TP n°1 : la pluie de numéro

Pour ce dernier chapitre, je fais une pause, et je vous laisse travailler un peu 😊. Ce TP aura pour but de créer un jeu complet : la pluie de numéro.

Quelques consignes

Menu principale

Tout d'abord, le jeu doit démarrer sur un menu avec 3 choix :

- Jouer ==> voir fonction de jeu.
- Crédits ==> Crédits donne des informations sur le créateur, année de création et les personnes remerciées pour la création du jeu. Ces renseignements s'effaceront après avoir appuyé sur la touche 
- Quitter ==> Quitter quitte le jeu.

fonction de jeu

Le principe est le suivant : un chiffre entre 0 à 9 (choisi au hasard) tombe de l'écran (du haut vers le bas) et le joueur doit appuyer sur la touche correspondant à ce chiffre :



Lorsqu'un numéro a atteint le bas de l'écran, le joueur a perdu, et on peut le lui dire. 😊

Bien sûr n'importe quel joueur peut quitter à tout moment le jeu avec la touche : 

La vitesse du jeu doit être croissante. A chaque numéro trouvé par le joueur, la vitesse doit augmenter.



N'oubliez pas qu'il ne faut jamais se retrouver avec l'équivalent d'un *Pause* 0

Quand vous aurez fait tout cela, on pourra encore faire quelques ajouts sympas, par exemple :

- Faire un mode difficile (en faisant défiler plusieurs numéros en même temps par exemple).
- Faire une intro au début du programme (histoire de rendre le jeu dynamique).
- Faire apparaître les numéros à des endroits choisis au hasard.
- Et tout ce que vous imaginerez d'autre...

Les commandes à connaître



Mais comment effacer l'écran d'un coup ? comment créer un nombre aléatoire ?

J'allais vous le dire 😊.

Pour effacer l'écran, vous utiliserez la commande *ClrHome* :   .

Pour créer un nombre aléatoire, il faudra utiliser la commande *rand*. Celle-ci va créer un nombre aléatoire entre 0 et 65 535.



La commande *rand* se trouve ici :   .



Mais comment faire pour que ce nombre soit entre 0 et 9 ?

C'est là qu'intervient le super modulo 😎, cette tâche fait partie de ses spécialités :

Code : Axe

```
:rand→A
:A^10→A
```

Et en simplifié :

Code : Axe

```
:rand^10→A
```

En fait, le reste de la division euclidienne de n'importe quel entier par 10, sera toujours compris entre 0 et 9 (car 10 est divisible par 10), de même si je veux créer un nombre aléatoire entre 0 et 95 :

Code : Axe

```
:rand^96→A
```

Pour le reste vous savez faire (si si, je vous assure ! 😊). Sur ce, prenez bien votre temps pour faire ce TP, (plusieurs jours/semaines si il faut), et lisez la suite **SEULEMENT** quand vous aurez fini.



Mais n'hésitez pas à relire les chapitres précédents pour vous aider.

Correction

Après avoir amélioré votre programme mieux que je ne saurais le faire, vous venez ici pour voir ma modeste correction ? 😊

Le menu et le crédit

Soit, commençons par le commencement : il vous faut un menu qui réagisse en fonction des touches choisies.

Ici j'ai choisi la touche **2nde** pour jouer, **A-Lock alpha** pour le crédit et **annul CLEAR** pour quitter :

Secret (cliquez pour afficher)

Code : Axe

```
:.PLUIENUM
:.Le menu principal
:Repeat getKey(15)
:Text(20,10,"2nde : Jouer
:Text(20,20,"Alpha : Credits
:Text(20,30,"Clear : Quitter
:
:If getKey(54)
:Mes fonctions de jeu
:End
:
:If getKey(48)
:.Le credit du jeu
:End
:End
```

Il n'y a pas de surprise, tout ça a déjà été vu.



Les caractères avec accent (é, à, è, ô, etc.) ne sont pas mis ici car la calculatrice ne les reconnaît pas dans ce cas là.

Ensuite je fais mon crédit, qui revient au menu après avoir appuyé sur la touche

précéd résol entrer :

Secret (cliquez pour afficher)

Code : Axe

```
:ClrHome
:Text(0,0,"Jeu Creer par
:Text(0,8,"Kindermoumoute
:Text(0,16,"Octobre 2010
:Text(0,30,"Merci à Wagl pour la
:Text(0,38,"correction de ce tutoriel
:Repeat getKey(9)
:End
:ClrHome
```

J'afface l'écran au début et à la fin de mon crédit, via la commande *ClrHome* vue plus haut. Et j'attends que la touche

précéd résol entrer

soit pressée avec une boucle avec rien dedans.

Code : Axe

```
:Repeat getKey(9)
:End
```

Ce code est l'équivalent de la *Pause* du TI-Basic !

Le jeu

Passons aux choses sérieuses.

D'abord, comment choisir un nombre aléatoire :

[Secret \(cliquez pour afficher\)](#)

Code : Axe

```
:! If T
:rand^10->N
:1->T
:End
```

Il suffira de donner 0 comme valeur à T pour créer un nouveau nombre aléatoire (entre 0 et 9) dans la variable N (avec la commande *rand* vu plus haut).

Ensuite, pour savoir si la touche pressée correspond au nombre N, on peut faire :

[Secret \(cliquez pour afficher\)](#)

Code : Axe

```
:If getKey(33) and (N=0)
:1->I+S->S
:ElseIf getKey(34) and (N=1)
:1->I+S->S
:1->I+S->S
:
:.[etc]
:
:Else getKey(20) and (N=9)
:1->I+S->S
:End
```

Quand la touche appuyée correspond à N, on rajoute 1 à I (devient VRAI), puis met la valeur I+S (I+S) dans la variable S. La variable I va servir à initialiser le numéro, et la variable S va être la somme des points.

Cependant, comme vu dans le QCM précédent, on peut simplifier ce code :

[Secret \(cliquez pour afficher\)](#)

Code : Axe

```
:getKey(33) and (N=0)->I
:getKey(34) and (N=1)+I->I
:getKey(26) and (N=2)+I-I
:getKey(18) and (N=3)+I-I
:getKey(35) and (N=4)+I-I
:getKey(27) and (N=5)+I-I
:getKey(19) and (N=6)+I-I
:getKey(36) and (N=7)+I-I
:getKey(28) and (N=8)+I-I
:getKey(20) and (N=9)+I-I+S->S
```

Comme N n'aura qu'une valeur à chaque fois, il n'y a pas de risque que I soit égal à 2 ou plus.



Attention, il ne faut pas oublier le *+I* dans les calculs, ou la variable I sera toujours remise à zéro à la ligne suivante (sauf pour 9).

Et encore amélioré :

[Secret \(cliquez pour afficher\)](#)

Code : Axe

```
N=0 and getKey(33) or (N=1 and getKey(34)) or (N=2 and getKey(26)) or (N=3 and S
```



Ce code tient sur une seule ligne !

Mais dans ce code il suffirait d'appuyer sur toutes les touches en même temps pour pulvériser les scores 🎯, donc je suis obligé de mettre un code anti-triche en remplaçant les *or* par des *xor* :

[Secret \(cliquez pour afficher\)](#)

Code : Axe

```
:N=0 and getKey(33) xor (N=1 and getKey(34)) xor (N=2 and getKey(26)) xor (N=3 and S
```

Le ou exclusif (*xor*) va obliger qu'une seule touche soit pressée. Quand celui-ci retourne 1 (VRAI), on regarde si il y a également une touche correspondant à la valeur de N. Si il y en a une, alors I=1 (VRAI), et on incrémente S bien entendu.

On affiche N, qui descend de l'écran plus ou moins rapidement :

[Secret \(cliquez pour afficher\)](#)

Code : Axe

```
:Text(40,B,N►Dec
:Pause W
:B++
```

Maintenant, on va utiliser la variable I, qui détecte quand une touche correspond à N. Quand I est égale à 1, la vitesse du jeu doit augmenter et le numéro déjà à l'écran doit s'effacer :

Secret (cliquez pour afficher)

Code : Axe

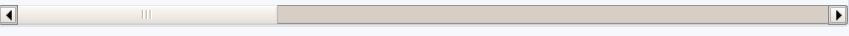
```
:Text(40,B,N►Dec
:If I
:W>1?W--
:ClrHome
:0→I→T→B
:T=0, on recréé un nombre aléatoire
:B=0, N est affiché en haut de l'écran
:End
:Pause W
:B++
```

Toute la partie de jeu :

Secret (cliquez pour afficher)

Code : Axe

```
:On initialise nos variables (ici le temps W commence à 100 ms)
:0→I→T→B+100→W
:On efface le menu précédent
:ClrHome
:
:Le jeu commence, et s'arrête quand la lettre touche le bas de l'écran
:Repeat B=60 or getKey(15)
:On choisit un nombre N aléatoire
:If T
:rand^10→N
:1→T
:End
:On affiche N en fonction de B
:Text(40,B,N►Dec
:On vérifie si la touche pressée correspond à N (avec un code anti-triche)
:N=0 and getKey(33) xor (N=1 and getKey(34)) xor (N=2 and getKey(26)) xor (N=
:Si cela correspond, on initialise les variables et le temps W diminue
:If I
:W>1?W--
:ClrHome
:0→I→T→B
:End
:
:Pause W
:B++
:End
:On efface l'écran
:ClrHome
:Le score s'affiche dans le menu principal
:Text(0,0,"Score :
:Text(26,0,S►Dec
```



Les ajouts possibles

Vous avez fini votre programme, mais deux forces en vous se combattent encore : Le **bien** et le **mieux**. 🤖

Ici je montrerai 3 améliorations pour rendre votre jeu un peu mieux (mais il n'y a jamais de limite dans les améliorations) :

- Faire une intro au début du programme (histoire de rendre le jeu dynamique).

L'intro que j'ai choisie de faire fait défiler le titre du jeu de gauche à droite, et fait apparaître très brièvement des tas de numéros sur l'écran. 🎯

Ce n'est qu'un exemple, et vous êtes libre de faire l'intro que vous voulez :

Secret (cliquez pour afficher)

Code : Axe

```
:For(Z,0,60)
:ClrHome
:Text(Z,2," PLUIE"
:Text(Z,10," DE
:Text(Z,18," NUMERO
:Text(rand^92,Z-(rand^45-B),rand^7-A►Dec
:Text(rand^92,Z+(B-5),A+1►Dec
:Text(rand^92,Z-(B+7),A+2►Dec
:Text(rand^92,Z+(B-7),A+3►Dec
:End
:ClrHome
```

Lorsque l'on utilise la commande *ClrHome* pour effacer l'écran constamment dans une boucle, le bas de l'écran sera plus clair

(gris clair) et le haut plus foncé (gris noir) lors de l'affichage du texte.

 Pour limiter les *rand* (qui ralentissent fortement le programme), j'ai mis dans des variables temporaires B et A qui seront dans tous les cas initialisées par la suite.

- Faire apparaître les numéros à des endroits choisis au hasard.

Il suffit de rajouter une variable A lors de l'affichage du texte *Text(A,B,N▶Dec* qui sera choisi au hasard dans notre fonction déjà présente :

Secret (cliquez pour afficher)

Code : Axe

```
:!If T
:rand^10-N
:rand^92-A
:1-T
:End
```

- Faire un mode difficile (en faisant défiler plusieurs numéros en même temps par exemple).

Dans mon exemple, il y a le niveau 0 (avec 1 numéro) et le niveau 1 (avec 2 numéros). 😊

Le but va être de rajouter un deuxième numéro, sans que cela ne perturbe le premier code. On va donc être obligé de recopier le même code mais avec des noms de variables différents. Il faudra que ce code soit pris en compte seulement lorsque le joueur l'a choisi (via un menu) :

Secret (cliquez pour afficher)

Code : Axe

```
:ClrHome
:Repeat getKey(9)
:Text(0,10,"Niveau :
:Text(28,10,L▶Dec
:L<1 and getKey(4)-(L>0 and getKey(1))+L->L
:End
```

Bien sûr vous êtes incollable sur l'incrémentation et la décrémentation, ce code n'a rien de difficile à vos yeux. 😊

Maintenant on reprend toutes les fonctions de notre premier code, mais avec d'autres variable, et seulement si il y a L :

Secret (cliquez pour afficher)

Code : Axe

```
:Repeat Y=60 or (B=60) or getKey(15)
:.Code
:If L
:!:If U
:rand^10-M
:rand^92-X
:1-U
:End
:End
:.Code
:If L
:Text(X,Y,M▶Dec
:End
:If getKey(33) xor getKey(34) xor getKey(35) xor getKey(36) xor getKey(26) xo
:Ici on laisse la condition, car ça raccourcit le code, vu que l'on a 2 fois
:If L
:(getKey(33) and (M=0)) or (getKey(34) and (M=1)) or (getKey(26) and (M=2)) o
:End
:End
:.Code
:If J
:W>1?W--
:End
:ClrHome
:0-J-U-Y
:End
:.Code
:L?Y++
```

Le code final :

Secret (cliquez pour afficher)

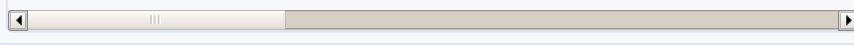
Code : Axe

```
:PLUIENUM
:
:Une petite intro pour le fun
:For(Z,0,60)
:ClrHome
:Text(Z,2," PLUIE
:Text(Z,10," DE
:Text(Z,18," NUMERO
:Text(rand^92,Z-(rand^45-B),rand^7-A▶Dec
:Text(rand^92,Z+(B-5),A+1▶Dec
```

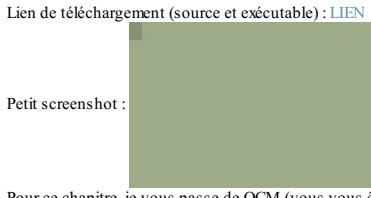
```

:Text(rand^92,Z-(B+7),A+2►Dec
:Text(rand^92,Z+(B-7),A+3►Dec
:End
:ClrHome
:0→S→I→J→T→U→A→B→X→Y+1→L→W
:
:.Le menu principal
:Repeat getKey(15)
:Text(20,10,"2nd : Jouer
:Text(20,20,"Alpha : Credits
:Text(20,30,"Clear : Quitter
:
:.Mes fonctions de jeu
:If getKey(54)
:ClrHome
:.Choix niveau
:Repeat getKey(9)
:Text(0,10,"Niveau :
:Text(28,10,L►Dec
:L<1 and getKey(4)-(L>0 and getKey(1))+L->L
:End
:100→W
:0→S
:ClrHome
:
:Repeat Y=60 or (B=60) or getKey(15)
:!If T
:rand^10→N
:rand^92→A
:1→T
:End
:If L
:!If U
:rand^10→M
:rand^92→X
:1→U
:End
:End
:Text(A,B,N►Dec
:L?Text(X,Y,M►Dec
:If N=0 and getKey(33) xor (N=1 and getKey(34)) xor (N=2 and getKey(26)) xor
:If L
:M=0 and getKey(33) xor (M=1 and getKey(34)) xor (M=2 and getKey(26)) xor (M=
:End
:End
:If I
:W>1?W--
:ClrHome
:0→I→T→B
:End
:If J
:W>1?W--
:ClrHome
:0→J→U→Y
:End
:Pause W
:B++
:L?Y++
:End
:ClrHome
:Text(0,0,"Score :
:Text(26,0,S►Dec
:0→I→J→T→U→A→B→X→Y+1→L→W
:End
:
:.Le credit du jeu
:If getKey(48)
:ClrHome
:Text(0,0,"Jeu Cree par
:Text(0,8,"Kindermoumoute
:Text(0,16,"Octobre 2010
:Text(0,30,"Merci à Wagl pour la
:Text(0,38,"correction de ce tutoriel
:Repeat getKey(9)
:End
:ClrHome
:End
:End

```



Lien de téléchargement (source et exécutable) : [LIEN](#)



Petit screenshot :

Pour ce chapitre, je vous passe de QCM (vous vous êtes déjà assez creusé les méninges 😊).
Je ne reparlerai pas de ce TP dans les chapitres suivants, néanmoins vous allez apprendre de nouvelles choses dans la prochaine partie (des tas de nouvelles choses 🤯) qui pourront être utilisées dans ce programme pour l'optimiser encore plus, le rendre encore plus beau, plus rapide, plus jouable... (plus parfait ?).
Je vous mets l'eau à la bouche?
Bon OK, allez-y 😊.

Ici vous avez principalement vu les bases nécessaires pour faire tourner tout programme. Elles sont souvent similaires entre chaque langage de programmation. Avant d'aller lire la deuxième partie, relisez bien cette partie, vérifiez que vous avez tout saisi, faites des petits programmes pour vous entraîner, retenez votre respiration.. et direction la prochaine partie ==> [].

Partie 2 : Notions de programmation avancées

Dans cette partie nous étudierons les bases propres à l'Axe Parser. La difficulté de ce chapitre est un cran plus élevé que le chapitre précédent, il est donc nécessaire de lire attentivement chaque chapitre et de s'entraîner (c'est comme cela que vous apprendrez). Sur ce allons-y !

Les pointeurs

Vous ne les connaissez peut être pas, mais les pointeurs remplissent une bonne partie des langages de programmation. En Axe on n'y échappe pas, mais je suis convaincu que vous allez bientôt les adopter dans tous vos programmes. 😊

La théorie

Vous devez comprendre ce qu'est un pointeur ! Si vous n'avez jamais utilisé de langage de programmation utilisant les pointeurs avant, soyez très attentifs. Les pointeurs sont des outils puissants et utiles que vous allez utiliser dans pratiquement tous vos programmes. Ils permettent d'organiser la RAM.

Qu'est-ce que la RAM ?

Littéralement *Random-access memory* : mémoire à accès aléatoire.

C'est de la mémoire vive alimentée constamment par vos piles qui est constituée environ de :

- 32 768 octets pour les TI-83+.
- 131 072 octets pour les TI-83+SE.
- 131 072 octets pour les TI-84+ ; pour les révisions matériels A à G sans lettre.
- 49 152 octets pour les TI-84+ ; pour les révisions matériels H et plus.
- 131 072 octets pour les TI-84+SE ; pour les révisions matériels A à G sans lettre.
- 49 152 octets pour les TI-84+SE ; pour les révisions matériels H et plus.
- 131 072 octets pour les TI-Nspire non-CAS en mode 84+SE.

Sur tous les modèles vous ne verrez que 24Ko d'affiché dans le menu mémoire (), c'est

en fait la mémoire allouée pour stocker vos programmes et autres données. 😊

Un octet ? qu'est ce que c'est ?

Un octet est constitué de 8 bits. Un bit est une partie modifiable de la mémoire dans lequel on peut stocker une valeur égale à 0 ou 1. Donc un octet est égale à 8 bits de valeur 0 ou 1.

Voici quelques exemples de combinaisons d'octets :

Secret (cliquez pour afficher)

```
00000000
00000001
00000011
00000110
10010101
00101100
00000111
00000100
01010000
```

Il y a beaucoup de combinaisons possibles, mais je n'en dis pas plus pour l'instant. 😊

Mais comment peut-on gérer toute cette RAM ? 🤔

Par le moyen le plus simple : donner à chaque octet de RAM sa propre adresse.
L'octet 0 a l'adresse 0, l'octet 1 a l'adresse 1, et ce jusqu'au dernier octet 32 767 de l'adresse 32 767 (pour une TI-83+).

 32 767 octets, plus l'octet 0 égale bien 32 768 octets. 😊

Donc lorsque vous démarrez votre programme, celui ci doit être mis dans la RAM.
C'est là qu'intervient le pointeur, celui ci va également être dans la RAM, mais **sa valeur va être l'adresse d'un endroit dans le programme** (dans la RAM également).

La pratique

Voici un exemple de pointeur sur une chaîne de caractères :

Code : Axe

```
:.HELLO
:"Hello World"--Str1
:Disp Str1
```

 Le menu des variables String est ici : 

 Mais c'est exactement pareil qu'en TI-Basic 😊

Faux !

Cela donne l'impression de fonctionner pareil. Regardons ce qui se passe réellement.

Tout d'abord, Str1 est un pointeur qui est un nombre, pas une chaîne réelle.
Donc, la phrase "Hello World" est stockée dans l'exécutable à une certaine adresse et le pointeur Str1 est **juste un nombre qui nous indique où cette chaîne se trouve** dans le programme.

La fonction d'affichage prend un certain nombre comme argument. Ce nombre va indiquer une adresse, et c'est comme cela qu'il sait où trouver la chaîne de caractère à afficher.

Seulement, la chaîne de caractère "Hello world" ne peut pas tenir en un seul octet, il faut donc répartir chaque caractère dans un

octet. Plus précisément, il y a une lettre par octet.

Imaginons que la lettre H soit dans l'octet 1200, Str1 va donc contenir le nombre 1200, et quand on fait appel à lui il va lire la chaîne en lisant les octets aux adresses qui suivent 1200 :

Lettre stockée	Adresse contenant la lettre
"H"	1200
"e"	1201
"l"	1202
"l"	1203
"o"	1204
" "	1205
"W"	1206
"o"	1207
"r"	1208
"l"	1209
"d"	1210
0	1211

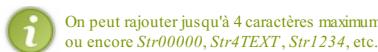


Que vient faire un 0 à l'adresse 1211 ? 🤔

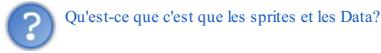
En fait, ce 0 est mis automatiquement à chaque fois qu'une chaîne de caractère est pointée. Son rôle n'est pas négligeable : il permet d'arrêter la lecture des octets. Imaginez, sans lui votre Disp afficherait toute la mémoire vive de votre calculatrice sous forme de caractères.💡

En Axe, Str1 est ce que l'on appelle un **pointeur statique**. Cela signifie qu'une fois qu'il est déclaré, sa valeur ne peut pas changer par la suite dans le programme.

Les pointeurs statiques sont Str, Pic, et GDB. Vous pouvez les utiliser comme vous voulez. Ce sont de simples numéros, mais c'est par convention qu'on utilise Str pour les chaînes de caractères (string), Pic pour les sprites, et GDB pour les Data. En aucun cas ce ne sont les variables utilisées en TI-Basic.



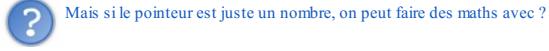
On peut rajouter jusqu'à 4 caractères maximum derrière un pointeur statique, donc notre pointeur peut s'appeler Str1A, ou encore Str0000, StrTEXT, Str1234, etc.



Qu'est-ce que c'est que les sprites et les Data?

⚠️ Pas maintenant, cela fera l'objet d'un prochain chapitre !

Evolution possible



Mais si le pointeur est juste un nombre, on peut faire des maths avec ?

Oui on peut faire des maths avec un pointeur ! Observons ce code par exemple :

Code : Axe

```
:.HELLO
:"Hello World"→Str1
:Disp Str1+6
```

Cela affichera juste "World", car la chaîne est stockée dans l'ordre dans des adresses. Reprenons nos adresses de tout à l'heure, $1200+6=1206$:

Lettre stockée	Adresse contenant la lettre	Statut
"H"	1200	Ignoré
"e"	1201	Ignoré
"l"	1202	Ignoré
"l"	1203	Ignoré
"o"	1204	Ignoré
" "	1205	Ignoré
"W"	1206	Pris en compte
"o"	1207	Pris en compte
"r"	1208	Pris en compte
"l"	1209	Pris en compte
"d"	1210	Pris en compte
0	1211	Pris en compte

Donc, au moment où nous arrivons à l'adresse 1206, nous sommes à "W". C'est pourquoi l'affichage de la partie "Hello " sera sautée.

Un petit exercice de recherche qui va vous aider à comprendre tout ça : lorsque je démarre mon jeu, je veux faire apparaître un "chargement" avec les petits points derrière qui défilent : d'abord aucun, puis le premier apparaît, le deuxième, le troisième, enfin on efface les 3 d'un coup pour n'avoir que le texte "chargement". On peut bien entendu répéter cela plusieurs fois, voire même jusqu'à ce qu'une touche soit pressée.

Secret (cliquez pour afficher)

Dans une boucle *For*, on mettra une condition décrémentée :

Code : Axe

```
:.LOAD
:
:"CHARGEMENT"--Str0LOAD          ::Initialisation des chaînes de c
:"...."--Str4PTS                  ::Initialisation de Y
:4--Y
:
:For(Z,0,10)                      ::Cette boucle For va se répéter
:DS<(Y,4)                          ::Quand Y vaut 0, le code suivant
:ClrHome                           ::Quand Y est égale à 0, on effac
:End
:
:Output(0,0,Str0LOAD              ::On affiche le chargement
:Output(10,0,Str4PTS+Y            ::avec le nombre de points voulu
:
:Pause 600                         ::Sans pause le programme irait t
:End
```

Je ne vous laisserai pas passer si vous n'avez pas compris. ☺

Sans ces pointeurs vos programmes seraient beaucoup moins marrants à lire, c'est un outil indispensable pour la suite du cours.

Les fonctions

Indispensables pour organiser son code, les fonctions ont un rôle très important pour l'optimisation du code. Ce chapitre n'a rien de compliqué, mais il est important d'en avoir compris tout les intérêts.

Les labels

Maintenant que vous savez ce qu'est un pointeur, il vous sera plus facile de comprendre ce qu'est un label (littéralement une étiquette). C'est en fait un repère placé dans votre programme, auquel on peut faire appel lorsque l'on en a besoin.

Dans un programme en Axe, on utilisera la commande *Lbl* suivi du nom du label. Ce nom est composé de 5 caractères (lettre majuscule et minuscule et chiffres) maximum :

Code : Axe

```
:.code A
:Lbl Label
:.code B
:Lbl Test1
:.etc
```



Par les adresses pardi ! 😊

Il suffit d'indiquer l'adresse où l'on veut aller (l'adresse est notre repère) et en un rien de temps notre programme bascule vers le code qui commence à partir de cette adresse.

Heureusement les commandes pour faire cela sont très simples à utiliser, et le compilateur va remplacer tout seul les noms des labels par leur adresse correspondante. 😊

La première façon de l'utiliser avec la commande *Goto*, celle-ci permet d'aller au label par un aller simple :

Code : Axe

```
:Goto HEY
:.code A
:Lbl HEY
:.code B
```



Le code A sera donc sauté, et l'on passera au code B directement.

Le problème c'est que l'on est vite rempli de *Goto* :

Code : Axe

```
:Lbl 0
:Goto HEY
:Lbl CO
:Goto F
:Lbl HEY
:Goto CO
:Lbl F
:Goto 0
```



Donc ils peuvent être utiles, mais il ne faut pas en abuser !

Les goto peuvent donner de mauvaises habitudes de programmation, si vous les utilisez trop, vous risquerez de faire du code rempli de label et de goto, au détriment des boucles, ce qui rendra votre code illisible et très mal optimisé ! 😞



Appeler une fonction

Pour éviter de remplir vos codes de *Goto*, il existe une autre commande très pratique, qui fait appelle à vos label comme des fonctions !



Une fonction va être une partie de code qui va faire une action bien précise, par exemple, si dans mon code principal je dois afficher à plusieurs endroits le même code, je peux créer une fonction pour appeler ce code à ma guise (oui, à ma guise 😊).

Le code d'une fonction est encadré par 2 commandes, la première est le *Lbl* déjà vu, et la deuxième est un *Return*, que l'on trouve

ici :

Donc on aura ce code pour une fonction :

Code : Axe

```
:Lbl Fnct
:.Code de la fonction
:Return
```



Et comment fait-on appel à cette fonction ?

Il existe deux manières d'appeler une fonction, soit avec la commande `sub(` suivie du label, soit en marquant directement le label mais en rajoutant des parenthèses :

Code : Axe

```
:Code A
:sub(Fnct)
:Fnct()
:.Code C
:
:Lbl Fnct
:.Code B
:Return
```

Dans l'ordre on exécutera le code A, le code B deux fois, puis le code C et une troisième fois le code B.



En fait, le programme précédent s'arrêtera au `Return` de la dernière ligne, car celle-ci ne sert pas qu'à terminer les fonctions, elle ferme le programme si aucune fonction n'est en cours. Pour éviter de passer toujours par les fonctions avant de quitter le programme, on mettra le `Return` après le code C :

Code : Axe

```
:Code A
:sub(Fnct)
:Fnct()
:.Code C
:Return
:
:Lbl Fnct
:.Code B
:Return
```



Le `Return` est en quelque sorte comme le 0 que l'on retrouve pour les pointeurs.

Le `Return` va également renvoyer la dernière valeur modifiée ou lue dans la fonction. Par exemple :

Code : Axe

```
:Fnct ()→B
:On récupère la valeur de retour, B vaut donc 1
:
:Lbl Fnct
:.Code
:1
:Return
```

Mais aussi de cette manière en l'indiquant directement derrière le `Return` :

Code : Axe

```
:Fnct ()→B
:.B vaut donc 5
:
:Lbl Fnct
:.Code
:Return 5
```

Il existe également deux instructions de retour conditionnel : `ReturnIf EXP` pour simplifier le code suivant :

Code : Axe

```
:If EXP
:Return
:End
```

En celui-ci :

Code : Axe

```
:ReturnIf EXP
```

Bien sûr il existe le complément `Return!If EXP` qui s'utilise de la même manière :

Code : Axe

```
:Return!If EXP
```

Autre chose à savoir : la fin d'un programme est elle-même un *Return* !

Donc exceptionnellement, pour les fonctions situées en fin de programme, on peut s'abstenir de *Return*.

On peut donc organiser notre code avec les fonctions, à peu près comme sur ce schéma :

Code : Axe

```
:MONPROGR
:
:sub(ZZZ)
:
:sub(A1)
:
:sub(ZZZ)
:
:Return
:
:Lbl A
:.Code A
:Return
:
:Lbl A1
:.Code B
:Return
:
:.Vous mettez toutes les fonctions dont vous avez besoin
:
:Lbl ZZZ
:sub(A)
:.Code Z
:.La fin du programme est un Return
```



💡 Une fonction qui appelle une autre fonction ! C'est possible ça ?

Et oui, comme une fonction n'est que du code, on peut y appeler d'autres fonctions !

Les variables temporaires

Parfois, vous faites plusieurs fois les mêmes calculs mais avec des variables différentes, par exemple :

Code : Axe

```
:A*2+8/255→A
:B*2+8/255→B
:.Ou encore
:Text(cos(X+3),sin(Y-3),"A"
:Text(cos(A+3),sin(B-3),"A
:
```

Pour vous simplifier la vie on a créé des variables dites temporaires ; mais en réalité ce sont de simples variables : **r₁, r₂, r₃, r₄, r₅ et r₆**.



On retrouve le menu pour ces variables ici :



De ce fait, on peut faire tout ce que l'on fait avec les autres variables avec :

Code : Axe

```
:r1*2+8/255
:Text(cos(r1+3),sin(r2-3),"A
```

Mais leur particularité est que l'on peut les utiliser facilement dans les fonctions en tant qu'arguments. Pour cela il faut rajouter les valeurs à envoyer après le label, en ajoutant une virgule :

Code : Axe

```
:sub(FX,A,B,C)
:.Ou encore
:FX(A,B,C)
:.On peut mettre jusqu'à 6 arguments
:Return
:
:Lbl FX
:.r1 vaut A, r2 vaut B, r3 vaut C
```

En reprenant les exemples plus haut :

Code : Axe

```
:Calc1(A)→A
:Calc1(B)→B
:
:sub(TEXT,X,Y)
:sub(TEXT,A,B)
:Return
:
:Lbl Calc1
:r1*2+8/255
:Return
:
:Lbl TEXT
:Text(cos(r1+3),sin(r2-3),"A
```



Mais mon code est plus long, donc c'est moins rentable. 😞

Faux, en apparence votre code paraît plus important, mais en réalité il est simplifié. Je suis sûr que vous vous rendrez compte très rapidement de l'importance des fonctions quand celles ci feront plusieurs lignes. 😊

Exercice d'application

Pour vous entraîner, voici un code à simplifier à l'aide de tous les outils que vous avez vus jusqu'à présent :

Code : Axe

```
:.SRC
:rand^500*14-13-A
:rand^500*14-13-B
:rand^500*14-13-C
:rand^500*14-13-D
:rand^500*14-13-E
:Disp "HEY WORLD
:Repeat getKey(9)
:End
:While getKey(9)
:End
:ClrHome
:rand^10-W
:rand^11-X
:rand^12-Y
:rand^13-Z
:Output(0,0,W►Dec
:Output(0,1,X►Dec
:Output(0,2,Y►Dec
:Output(0,3,Z►Dec
:Repeat getKey(9)
:End
:While getKey(9)
:End
```

Secret (cliquez pour afficher)

Vous aurez peut être trouvé d'autres simplifications que celle ci, mais j'ai préféré supprimer les variables W, X, Y et Z :

Code : Axe

```
:.Src
:Rand1()→A
:Rand1()→B
:Rand1()→C
:Rand1()→D
:Rand1()→E
:Disp "HEY WORLD
:Pause()
:ClrHome
:For(Z,10,13)
:Output(0,Z-10,rand^Z►Dec
:End
:Pause()
:Return
:
:Lbl Pause
:Repeat getKey(9)
:End
:While getKey(9)
:End
:Return
:
:Rand1
:Return rand^500*14-13
```

Ce code paraît plus long, mais il réduit presque de moitié la taille de votre programme !

Ce chapitre est peut être le plus facile à comprendre pour cette partie, relisez le bien, mettez des fonctions partout où vous le pouvez et entraînez vous bien !

Dessiner c'est gagner !

J'imagine que vous êtes impatient de pouvoir faire de beaux dessins sur l'écran de votre calculatrice. Dans ce chapitre vous verrez qu'il existe plusieurs écrans et qu'ils ont chacun une utilité particulière. 😎

Plusieurs écrans en un seul

Vous voyez 96 pixels par 64 pixels, et vous croyez avoir fait le tour de votre écran ? 🤪

Techniquement oui, cependant, il existe plusieurs écrans temporaires prêts à être affichés à tout moment.

Le fonctionnement

Tout d'abord le vrai écran existe (si si croyez moi 🤪), c'est celui que vous voyez, il est composé de $96 \times 64 (=6144)$ pixels : à chaque pixel on donne un statut : 1=allumé, 0=éteint.

Ensuite il y a l'écran temporaire, c'est une mémoire dans votre calculatrice spécifiée pour enregistrer le statut d'un écran, que l'on peut afficher sur le vrai écran quand on veut. Pour une meilleure compréhension, on appellera cet écran le buffer (du mot anglais qui veut dire écran tampon).



Il sert à quoi ce buffer ?

Le problème du vrai écran, c'est qu'il affiche les pixels en temps réel. Ce n'est donc pas très pratique de changer de décor si l'on doit observer tous les pixels changer un par un, cela enlève tout suspens (c'est plus simple à dire comme ça). 🤪

Grâce au buffer, on va pouvoir dessiner entièrement l'image voulue, puis l'afficher.

Par exemple, à gauche le buffer et à droite l'écran :

BONJOUR

le vrai écran affiche "bonjour"

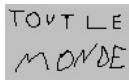


Pour l'instant je n'ai rien dans le buffer

Maintenant, j'affiche "tout le monde" dans le buffer :

BONJOUR

le vrai écran affiche toujours "bonjour"

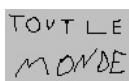


J'affiche "tout le monde" dans le buffer

Puis j'affiche le buffer à l'écran :

TOUT LE
MONDE

le vrai écran affiche "tous le monde"



"tout le monde" est toujours dans le buffer



Lorsque vous écrivez du texte, celui ci s'affiche directement sur l'écran sans passer par le buffer.

Afficher et effacer les différents écrans

Pour effacer le vrai écran, on connaît déjà la commande : *ClrHome*.

Maintenant, si on veut afficher le buffer sur le vrai écran, il y a la commande *DispGraph* :



Puis, si on veut afficher le vrai écran sur le buffer on utilisera la commande *StoreGDB* qui se trouve ici :



Enfin, pour effacer le buffer, on peut utiliser la commande *ClrDraw* qui se trouve ici :



On peut également simplifier le code :

Code : Axe

```
:DispGraph
:ClrDraw
```

On obtient le code suivant :

Code : Axe

```
:DispGraphClrDraw
```

Ce dernier est environ deux fois plus rapide que le précédent.

Pour donner un simple exemple d'utilisation, un programme qui sauve l'écran temporairement :

Code : Axe

```
:Text(20,20,"PLOP"
:PLOP apparaît sur le vrai écran
:StoreGDB
:.PLOP apparaît également sur le buffer
:ClrHome
:.On efface le vrai écran
:Repeat getKey()
:.L'écran reste blanc tant qu'on appuie pas sur une touche
:End
:DispGraphClrDraw
:...
:On affiche le buffer sur le vrai écran, on retrouve donc notre PLOP
:Puis on efface le buffer, mais PLOP est toujours sur le vrai écran
:...
:Pause 2000
```

Pixels et Géométrie

Pour dessiner sur cet écran intermédiaire, on aura ici une palette d'outils géométriques qui sont :

- Afficher un pixel.
- Afficher un cercle.
- Afficher un rectangle.
- Afficher un rectangle inversé.
- Inverser tous les pixels.

Afficher un pixel

Pour afficher un pixel, on utilisera la commande *Pxl-On()* :

Code : Axe

```
:30-X-Y
:Pxl-On (X,Y)
:DispGraph
:Pause 1800
```



La commande *Pxl-On* se trouve ici :

2nde **2ND** **dessin DRAW prgm** **4**

Les axes X et Y sont évidemment compris en 0 et 95 pour X et 0 et 63 pour Y.

On peut également effacer un pixel avec la commande *Pxl-Off()* :

5

Code : Axe

```
:30-X-Y
:Pxl-Off (X,Y)
:DispGraph
:Pause 1800
```

Dans le même principe, on peut inverser un pixel avec la commande *Pxl-Change()* :

6

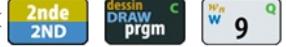


On peut même tester l'état d'un pixel avec *pxl-Test()* :

2nde **2ND** **dessin DRAW prgm** **7**

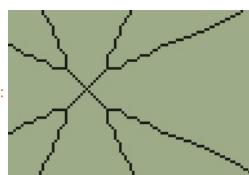
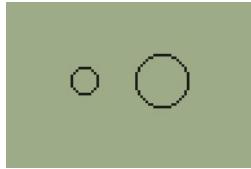
Renvoie 1 si le pixel est allumé, ou sinon renvoie 0 si il est éteint.

Afficher un cercle

Pour afficher un cercle on utilise la commande *Circle* (). Le centre du cercle a pour centre les coordonnées X et Y et a pour rayon R en pixels.

Code : Axe

```
:30→X→Y
:5→R
:Circle(X,Y,R)
:60→X
:10→R
:Circle(X,Y,R)
:DispGraph
:Pause 1800
```



Attention, si R vaut 0, vous aurez droit à l'écran suivant :

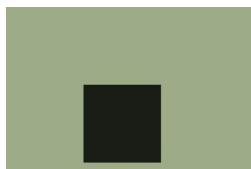
Afficher un rectangle

La commande utilisée pour le rectangle normal est *Rect* ().

Les coordonnées X et Y se situent en haut à gauche du rectangle, W est la largeur (comme width en anglais) et H est la hauteur (comme height en anglais) :

Code : Axe

```
:30→X→Y→W→H
:Rect(X,Y,W,H)
:DispGraph
:Pause 1800
```



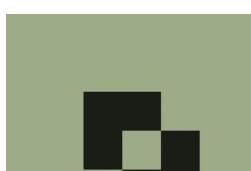
Afficher un rectangle inversant

La commande utilisée pour le rectangle inversant est *RectI* ().

Les coordonnées X et Y se situent en haut à gauche du rectangle, W est la largeur (comme width en anglais) et H est la hauteur (comme height en anglais) :

Code : Axe

```
:30→X→Y→W→H
:Rect(X,Y,W,H)
:W/2+X→X→Y
:RectI(X,Y,W,H)
:DispGraph
:Pause 1800
```



Inverser l'écran

C'est très facile, il faut utiliser la commande *DrawInv* : 

Code : Axe

```
:DrawInv
:DispGraph
```



Le troisième écran

Quoi encore un écran ? Mais ça n'en finit donc jamais ? 😊

Promis, c'est le dernier ! 🍏

Celui-ci s'appelle le back-buffer, il ne peut pas être affiché directement sur le vrai écran ! Il peut être affiché entièrement seulement en passant par le buffer.

Bon, bah, à quoi il sert ce back-buffer ? 😊

Il peut avoir de nombreuses utilisations dans un programme, mais sa principale utilité fera l'objet d'un prochain chapitre sur les niveaux de gris. 🍏

En attendant on peut toujours apprendre à le manipuler.

Modifier et exporter le back-buffer

Pour copier le back-buffer sur le buffer, on utilisera la commande *RecallPic* :

2nde 2ND dessin DRAW prgm ⌘

L2 2 .

Pour effacer le back-buffer, on utilisera la commande *ClrDraw* **r**, composé de *ClrDraw* déjà vu (

2nde 2ND dessin DRAW prgm ⌘

précéd résol entrer) et du r : 2nde 2ND angle apps L3 3 ⌘

Et pour copier le buffer dans le back-buffer, on utilisera la commande *StorePic* :

2nde 2ND dessin DRAW prgm ⌘

L1 1 ⌘

Et peut-on utiliser les fonctions de géométrie directement sur le back-buffer ?

Bien évidemment, il suffit de reprendre toutes les commandes vues précédemment et d'y rajouter le petit **r** :

Secret (cliquez pour afficher)

```
Pxl-On(X,Y) r
Pxl-Off(X,Y) r
Pxl-Change(X,Y) r
Line(X1,Y1,X2,Y2) r
Circle(X,Y,R) r
Rect(X,Y,W,H) r
RectI(X,Y,W,H) r
DrawInv r
```

En conclusion, chaque écran a ses avantages et ses défauts, mais je suis sûr que vous saurez rapidement comment les exploiter au maximum. 😊

Je tiens à remercier [mdrl](#) pour certaines précisions qu'il m'a apporté sur ce chapitre. 😊

Aller, chapitre suivant !

Les Datas 1/2

Je vous ai parlé précédemment d'octets, dans ce chapitre on va essayer d'éclaircir un peu tout cela. Vous allez devoir apprendre des nouvelles notions ; vous saurez ce qu'est l'hexadécimal, le binaire et comment faire rentrer des valeurs dans un octet (qui sera ensuite pointé bien évidemment 😊).

Le binaire

Pour pouvoir comprendre la suite du cours, vous devez connaître quelques notions : le binaire et l'hexadécimal.



Ouais, mais moi je connais par cœur tout ça !

Et alors ? un petit rappel ne fera pas de mal. 😊

Apprenons à compter

Tout d'abord il faut savoir ce qu'est une base numérique. La base la plus répandue aujourd'hui est la base 10 ; l'évolution a fait que l'homme a eu 10 doigts et qu'il a appris à compter avec ses 10 doigts. Donc, cette base 10 (appelée aussi base décimale), est constituée de 10 numéros :

0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Une fois que nous atteignons le dernier numéro, nous créons un nouvel emplacement en face du premier, et en incrémentant de 1 à chaque fois que l'emplacement précédent arrive à plus de 9 (il faut donc créer une dizaine) :

8, 9, 10, 11, 12, ..., 19, 20, ...

Et ainsi de suite pour les centaines, les milliers, etc.

Si l'on étudie notre façon de compter, on se rend compte que chaque emplacement ajouté est une puissance de 10 en plus :

10^3	10^2	10^1	10^0
chiffre	chiffre	chiffre	chiffre
*1000	*100	*10	*1



N'importe quel nombre exposant 0 aura comme valeur 1 : $x^0 = 1$

En fait, lorsque l'on change de base, ce tableau restera le même : on changera juste le nombre 10 par celui de la base voulue.

Le binaire

Le binaire (ou base 2) n'est constitué que de 2 numéros : 0 et 1. On y compte comme dans toutes les bases, en rajoutant un emplacement une fois que le précédent atteint le maximum de la base :

0, 1, 10, 11, 100, 101, etc.

Un petit tableau s'impose pour comparer le binaire et le décimal :

Pour le décimal :	0	1	2	3	4	5	6	7	8	9	10
Pour le binaire :	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010

Ici, je laisse les 0 en trop pour une meilleure vue d'ensemble.



💡 Et si je veux trouver un nombre binaire comme 101010, il n'y a pas un moyen plus simple que d'écrire toute la suite jusqu'à arriver à ce nombre ?

Si, pour cela reprenons le tableau précédent adapté à la base 2 :

2^5	2^4	2^3	2^2	2^1	2^0	-
chiffre	chiffre	chiffre	chiffre	chiffre	chiffre	
*32	*16	*8	*4	*2	*1	

Pour le nombre 101010 :

2^5	2^4	2^3	2^2	2^1	2^0	-
1	0	1	0	1	0	-
*32	*16	*8	*4	*2	*1	-
1 * 32	+0 * 16	+1 * 8	+0 * 4	+1 * 2	+0 * 1	= 42

Il suffit de connaître votre table de 2 et le tour est joué ! 😊

Quoi ? Vous ne la connaissez pas ? Je suis sûr que vous trouverez votre bonheur [ici](#).



💡 Et si je veux convertir un nombre décimal en binaire ?

C'est encore plus simple, la méthode consiste à décomposer le nombre décimal en plusieurs puissances de 2. Je prends le nombre 87 par exemple ; la puissance de 2 inférieure à 87 la plus proche est 64 :

$$87 - 64 = 23$$

On commence donc le nombre binaire par 1. Ensuite si on essaye avec le multiple en dessous :

$$23 - 32 =$$

Cela n'est pas possible car 32 est supérieur à 23, notre nombre binaire est de **10**. Essayons avec la puissance en dessous :

$$23 - 16 = 7$$

Notre nombre binaire est de **101**.

$$7 - 8 = -1$$

Pas possible, 8 est plus grand que 7. Donc en binaire on a **1010**.

$$7 - 4 = 3$$

En binaire **10101**.

$$3 - 2 = 1$$

En binaire **101011**.

$$1 - 1 = 0$$

En binaire **1010111**.

Maintenant, quelques conversions pour vous entraîner, remplissez le tableau ci dessous :

	-	Décimal	Binaire
Expression 1	?	1011	
Expression 2	?	10011001	
Expression 3	?	1111	
Expression 4	15		?
Expression 5	128		?

La correction :

Secret (cliquez pour afficher)

Pour l'expression 1, on a 4 chiffres binaires. On met tout ça dans notre tableau :

	2^3	2^2	2^1	2^0	-
1	0	1	1	-	
*8	*4	*2	*1	-	
1 * 8	+0 * 4	+1 * 2	+1 * 1	= 11	

Dans l'expression 2, on a 8 chiffres binaires. Pareil qu'au dessus :

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	-
1	0	0	1	1	1	0	0	1	-
*128	*64	*32	*16	*8	*4	*2	*1	-	
1 * 128	+0 * 64	+0 * 32	+1 * 16	+1 * 8	+0 * 4	+0 * 2	+1 * 1	= 153	

Dans l'expression 3, vous pouvez faire de nouveau un tableau, mais il y a d'autres méthodes plus simples. On calcule facilement que 10000 en binaire est égale à 16 en décimal (2^4). Donc $16 - 1 = 15$, on peut donc dire que **1111 = 15**

Si vous aviez trouvé l'expression 3, vous n'avez sûrement rien eu à calculer pour l'expression 4, car on a vu que **1111 = 15**.

L'expression 5 est également très facile à trouver, 128 étant une puissance de 2, il fallait juste ne pas se tromper sur le nombre de zéro à rajouter :

$$128 = 10000000$$

Vous retrouverez d'autres façons de faire sur [ce site](#) (pour votre culture personnelle 😊).

L'hexadécimal

Je ne ferai pas complètement le tour du sujet, mais seulement la partie qui nous intéresse autour de l'hexadécimal.

L'hexadécimal par rapport au décimal

L'hexadécimal est en fait la base 16. Seulement, il manque des chiffres pour arriver jusqu'à 16. On a donc utilisé des lettres, en l'occurrence ici les 6 premières lettres de l'alphabet :

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F,...

Puis on passe à l'emplacement suivant (comme dans toutes les bases) :

10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F, 20,..., A0,..., 100,..., A00, etc.



Mais que valent A, B, C, D, E et F ?

En décimal, ils valent respectivement : 10, 11, 12, 13, 14 et 15.

Un tableau s'impose :

En décimal	En hexadécimal
0	0
1	1
2	2
3	3

4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Et quand on arrive à F, on passe à 10 :

En décimal	En hexadécimal
15	F
16	10
17	11
18	12
19	13
20	14
21	15
...	...
25	19
26	1A
27	1B
28	1C
29	1D
30	1E
31	1F
32	20

Et ainsi de suite !

Je ne vais pas m'attarder plus sur la conversion de l'hexadécimal en décimal, je vous conseillerai juste de convertir l'hexadécimal en binaire, puis en décimal.

L'hexadécimal par rapport au binaire

La conversion de l'hexadécimal en binaire (et vice versa) est très simple ! Il vous suffit de connaître la tableau suivant :

hexadécimal	binnaire	décimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Vous devez connaître ce tableau ; prenez quelques minutes, heures s'il le faut pour apprendre ce tableau par cœur.



Mais ce tableau ne convertit que les 16 premiers chiffres (0 à 15), comment faire pour la suite ?

C'est là qu'il faut observer quelque chose d'important sur le tableau : lorsque l'on arrive à F en hexadécimal on arrive à 1111 en binaire. Il y a une relation qui fait qu'à ce moment les deux bases vont augmenter d'un emplacement. Je suis sûr que vous avez

deviné la relation entre ces deux bases, non ? 😊

Je vous aide un peu :

[Secret \(cliquez pour afficher\)](#)

Parce que 16 est une puissance de 2 !

Et que peut-on en conclure ? 😊

On peut en conclure que 1 chiffre hexadécimal est égal à 4 chiffres binaires !

Donc, si vous connaissez le tableau plus haut par cœur, vous arriverez à convertir le nombre 1BF5A0 très rapidement :

00011011111010110100000

Ou plus clairement :

1	B	F	5	A	0
0001	1011	1111	0101	1010	0000

(Soit 1832352 en décimal)

De même, pour trouver la valeur hexadécimal de 010111011011101000110010 :

0101	1101	1011	1010	0011	0010
5	D	B	A	3	2

(Soit 6142514 en décimal)

En fait, pour comprendre la suite du cours, vous aurez besoin de convertir le binaire en hexadécimal (très souvent), et l'hexadécimal en décimal (peu souvent).

Une histoire d'octets

Revenons à nos octets, et particulièrement à la question "qu'est ce qu'un octet ?".

Un octet est composé de 8 bits. Un bit a deux valeurs possibles : 0 ou 1. (une impression de déjà vu ? 😊)

Maintenant on assimile cela aux connaissances vues précédemment ; un bit est un chiffre binaire ; un octet est un nombre binaire de 8 chiffres maximum.

Petit exercice maintenant : déterminez la valeur maximale d'un octet en décimal.

Un indice :

[Secret \(cliquez pour afficher\)](#)

1111111 en Binaire.

Réponse :

[Secret \(cliquez pour afficher\)](#)

255 en décimal.

Ce nombre vous évoque des choses ?

Maintenant, calculez la valeur maximale de **deux** octets :

Un indice :

[Secret \(cliquez pour afficher\)](#)

11111111 11111111 en Binaire.

Réponse :

[Secret \(cliquez pour afficher\)](#)

65535 en décimal.

C'est maintenant que vous vous rappelez ; cette valeur est celle que l'on obtient quand on fait 0-1→A.

Cela signifie que votre variable est en fait composé de 2 octets.



Mais si un octet c'est 8 chiffres binaires, alors à quoi va servir l'hexadécimal ?

Rappelez vous, 1 chiffre hexadécimal vaut 4 chiffres binaires. On pourra donc écrire un octet sous forme de 2 caractères hexadécimaux. 😊



Il y a un truc que je ne comprends pas : avec les chaînes de caractères on arrive pourtant à enregistrer des lettres dans un octet, comment est-ce possible ?

En gros chaque lettre correspond à un nombre, et lors de l'affichage, la calculatrice va convertir ce nombre en lettre. Mais patience, cela fera l'objet d'un prochain chapitre. 😊

Les Datas : de simples données

Maintenant que vous connaissez vos octets sur le bout des doigts, vous allez pouvoir étudier ce qu'il retourne des Datas. 😊

Une liste de données

La commande *Data(* va créer une liste de valeurs que l'on utilisera pendant le programme. Chaque valeur sera en fait stockée dans un octet, et vous l'aurez deviné, on assimilera tout cela avec des pointeurs :

Code : Axe

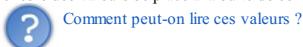
: Data (42, 12, 13, 7) →GDB1



La commande *Data(* se trouve ici :



Quand on fait cela, le pointeur statique *GDB1* va pointer l'octet contenant la valeur 42, et comme pour les chaînes de caractères, le reste des valeurs se place à la suite de cet octet.



Comment peut-on lire ces valeurs ?

Pour manipuler un seul octet à partir d'une adresse, il faut utiliser les accolades :

Code : Axe

```
: {GDB1}
```



L'ouverture et la fermeture des accolades se trouve respectivement ici

**2nde
2ND**

(

**2nde
2ND**

K

)

En fait, *{GDB1}* se manipule comme une variable d'un octet, sa valeur est toujours comprise entre 0 et 255.

En effet, si vous ne pouvez pas changer la valeur d'un pointeur statique, vous pouvez changer la valeur de l'octet pointé :

Code : Axe

```
:Data(42,12,13,7)→GDB1
:Disp {GDB1}►Dec
:.Affiche 42
:1→{GDB1}
:Disp {GDB1}►Dec
:.Affiche 1
```

Et amusons nous à passer d'un octet à un autre 😊 :

Code : Axe

```
:Data(42,12,13,7)→GDB1
:Disp {GDB1+2}►Dec
:.Affiche 13
:{GDB1}+{GDB1+1}→A
:.42+12=54
:Disp A►Dec
:.Affiche 54
```

Ce tableau devrait vous rappeler quelque chose :

Valeur stockée dans l'octet	Adresse contenant la lettre
42	1200
12	1201
13	1202
7	1203



L'adresse 1200 est, encore une fois, un simple exemple d'adresse.

Transformer de l'hexadécimal en Data

Maintenant, nous allons transformer des chiffres hexadécimaux en Data. Cela n'a rien de compliqué ; c'est même fait pour 😊

Toute data hexadécimale doit se mettre **entre crochets** :

Code : Axe

```
: [2A0C0D07]→GDB1
:For (Z,0,3)
:Disp {GDB1+Z}►Dec
:End
```



On retrouve l'ouverture des crochets et la fermeture des crochets respectivement ici :

**2nde
2ND**

[X R

et

] - W

Comme dit plus haut, deux caractères hexadécimaux valent un octet, il faut donc lire les datas ci dessus de la manière suivante :

2A ; 0C ; 0D ; 07

Mais maintenant vous savez trouver (très rapidement 😊) le résultat en décimal :

42 ; 12 ; 13 ; 7

Je pense que vous savez ce qu'il vous reste à faire : pratiquer dès maintenant. 😊

Bien sûr cette partie n'est qu'un avant goût de ce qui vous attend au prochain chapitre ; je vous demanderais de ne pas cliquer sur suivant tant que vous ne vous êtes pas entraîné sur la création et la manipulation des datas vus dans ce chapitre, c'est nécessaire pour pouvoir suivre le chapitre suivant ! 😊

Les Data 2/2

Pour faire un bon programme, il ne faut surtout pas négliger les graphismes ; nous verrons dans ce chapitre comment approfondir l'utilisation des Data, notamment avec la découverte de nouvelles notions : les sprites et les tiles, puis le tiling.

Mon premier sprite

Avec toutes les connaissances que vous avez vues dans les chapitres précédents, nous allons continuer d'explorer l'univers des data ; en commençant par les sprites et les tiles.



Qu'est ce qu'un sprite ou une tile ?

En Axe, les sprites et les tiles sont des images 8*8 pixels que l'on peut manipuler pour faire un personnages ou des décors.



Par convention on parle de sprite pour les personnages et objets, et de tile pour les décors.

La théorie

Pour créer un sprite, il faut vous munir d'un éditeur de sprite : autrement dit juste de quoi dessiner 😊

Vous pouvez donc prendre une feuille et un crayon ou dans mon cas j'utilise paint (il existe des éditeurs de sprites sur calculatrice également).

Le principe est simple : créer une feuille de dessin de 8*8 pixels :

Px	1	2	3	4	5	6	7	8
1	■							
2		■						
3			■					
4				■				
5					■			
6						■		
7							■	
8								■

Dans mon cas, je vais dessiner une émoticône souriante 😊 en noir et blanc ☺ :

Px	1	2	3	4	5	6	7	8
1	■		■	■	■	■		
2		■					■	
3	■		■		■			■
4	■						■	
5	■		■			■		■
6	■			■	■			■
7		■					■	
8	■		■	■	■	■		

Vous remarquerez qu'il n'y a que 2 possibilités : soit blanc, soit noir. De plus chaque ligne est composée de 8 pixels. Je suis sûr que cela vous rappelle quelque chose. 😊 Non ?

Secret (cliquez pour afficher)

Les couleurs ressemblent grandement au binaire : le blanc vaut 0 et le noir vaut 1 !
Et pour chaque ligne il y a 8 possibilités : une ligne est égale à un octet !

Maintenant, essayez de me donner la valeur hexadécimale du sprite, en mettant les nombres obtenus de chaque ligne à suivre.



Secret (cliquez pour afficher)

D'abord on recréé notre tableau en remplaçant les couleurs par les chiffres binaires correspondants :

0	0	1	1	1	0	0
0	1	0	0	0	0	1
1	0	1	0	0	1	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	0	0	1	1	0	1
0	1	0	0	0	1	0
0	0	1	1	1	1	0

Puis on reprend nos outils de conversion pour mettre tout cela en hexadécimal (rappel : 4 chiffres binaires valent 1 chiffre hexadécimal) :

3	C
4	2
A	5

8	1
A	5
9	9
4	2
3	C

Et on met le tout à suivre, entre des crochets :

Code : Axe

```
: [3C42A581A599423C]
```

Et voilà votre premier sprite !

La pratique

Pour l'afficher, c'est très facile : on pointe notre data, on utilise les commandes permettant de dessiner les sprites dans le buffer, et on affiche le buffer.

Code : Axe

```
: [3C42A581A599423C]→Pic1
:On utilise les pointeur Pic pour les sprites
:ClrDraw
:0→X-Y
:Pt-On(X,Y,Pic1
:DispGraph
```



La commande *Pt-On()* permet d'afficher le sprite sans effacer les pixels qui sont déjà allumés : seuls les points noirs du sprite s'affichent. L'angle haut gauche du sprite s'affiche à l'emplacement (X,Y).

La seconde commande est la commande *Pt-Off()* (**2nde 2ND**, **dessin DRAW prgm**, **2**). Cette commande permet

d'afficher le sprite en effaçant tout ce qu'il y avait avant dans l'emplacement de celui-ci :

Code : Axe

```
: [3C42A581A599423C]→Pic1
:ClrDraw
:0→X-Y
:Pt-Off(X,Y,Pic1
:RectI(20,20,16,16
:Pt-Off(24,24,Pic1
:DispGraph
```

Ce code va d'abord afficher le sprite en haut à gauche, puis un carré de côté 16 aura en son centre le sprite ; elle aura effacé tous les pixels du centre de ce carré avant d'être dessinée.

La dernière commande permet d'inverser les pixels correspondant au sprite :

Code : Axe

```
: [3C42A581A599423C]→Pic1
:ClrDraw
:0→X-Y
:Pt-Change(X,Y,Pic1
:RectI(20,20,16,16
:Pt-Change(24,24,Pic1
:DispGraph
```



Cette fois on verra le sprite au centre du carre, mais avec les couleurs inversées.



C'est exactement pareil que pour les fonctions de géométrie, il faut rajouter le *r* juste après :

Secret (cliquez pour afficher)

```
Pt-On()r
Pt-Off()r
Pt-Change()r
```

exercice

Essayez de faire un programme qui affiche notre sprite se baladant à travers l'écran (sans en sortir !) à l'aide de la commande *Pt-Change()*.

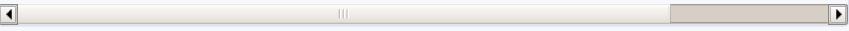
Secret (cliquez pour afficher)

Code : Axe

```

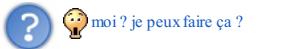
:PLOP
:[3C42A581A599423C]→Pic1
:CirDraw
:0→X→Y
:Rect(28,24,32,32
:Repeat getKey(15)
:(getKey(3) and (X≠88))-(getKey(2) and (X≠0))+X→X
:(getKey(1) and (Y≠56))-(getKey(4) and (Y≠0))+Y→Y
:.Déplacement de X et Y
:sub(CHG)
:.On inverse les pixels du sprite avant d'afficher (on la dessine dans le buf
:DispGraph
:sub(CHG)
:.On inverse les pixels du sprite après avoir affiché (on efface le sprite pa
:End
:Return
:
:Lbl CHG
:Pt-Change(X,Y,Pic1

```



Le tilingmapping

Vous attendiez sûrement ce moment avec impatience, mais le voici : vous êtes capable, à l'aide de vos connaissances, de créer une map composée de tiles 8*8, et de l'afficher à l'écran ! 😊



Je vais vous orienter un peu bien sûr. 😊

création de la map

Comme pour les sprites, une map ne s'improvise pas, il faut la dessiner avant.

Dans mon cas j'ai fait vite fait une maison et de l'herbe :

L'herbe :

Px	1	2	3	4	5	6	7	8
1	■							
2		■				■		
3	■				■			
4	■				■			
5								
6	■			■				
7			■			■		
8			■			■		

Le code hexadécimal est donc :

Secret (cliquez pour afficher)

[0044888800112222]

Maison haut gauche :

Px	1	2	3	4	5	6	7	8
1	■						■	
2					■	■	■	
3			■	■			■	
4		■	■				■	
5	■						■	
6	■						■	
7	■						■	
8	■						■	

Le code hexadécimal est donc :

Secret (cliquez pour afficher)

[01071961818181]

Maison haut droite :

Px	1	2	3	4	5	6	7	8
1	■							
2	■	■	■					
3	■			■	■			

4	■				■	■	
5	■					■	
6	■						■
7	■						■
8	■						■

Le code hexadécimal est donc :

[Secret \(cliquez pour afficher\)](#)

[80E0988681818181]

Maison bas gauche :

Pxl	1	2	3	4	5	6	7	8
1	■							■
2	■					■	■	
3	■			■	■			
4	■	■	■					
5	■		■	■	■	■		
6	■		■			■		
7	■		■		■	■		
8	■		■			■		

Le code hexadécimal est donc :

[Secret \(cliquez pour afficher\)](#)

[818698E09E929692]

Maison bas droite :

Pxl	1	2	3	4	5	6	7	8
1	■							■
2		■	■					■
3			■	■				■
4					■	■	■	
5		■	■	■				■
6		■		■				■
7		■	■	■				■
8								■

Le code hexadécimal est donc :

[Secret \(cliquez pour afficher\)](#)

[8161190771517101]

Donc on a dans l'ordre :

[Code : Axe](#)

```
: [0044888800112222] -> Pic1
: .Herbe
: [01071961818181]
: .Maison haut/gauche
: [80E09886818181]
: .Maison haut/droite
: [818698E09E929692]
: .Maison bas/gauche
: [8161190771517101]
: .Maison bas/droite
```



Il est important de remarquer que, comme deux caractères valent un octet, 8 octets valent une tile.

Sur ce principe, on affiche de la manière suivante les tiles :

[Code : Axe](#)

```
: Pt-On(X, Y, Pic1+0
: .Herbe
: Pt-On(X, Y, Pic1+8
: .Maison haut/gauche
: Pt-On(X, Y, Pic1+16
: .Maison haut/droite
: Pt-On(X, Y, Pic1+24
```

```
:Maison bas/gauche
:Pt-On(X,Y,Pic1+32
:.Maison bas/droite
```

Comme pour les chaînes de caractères, on peut **manipuler les pointeurs** pointant des tiles.

On remarque tout de suite que 0, 8, 16, 24 et 32 sont multiples de 8 : 0*8, 1*8, 2*8, 3*8 et 4*8.
Le seul nombre changeant peut être une variable, on obtient donc la simplification suivante :

Code : Axe

```
:La variable A vaut la valeur de la tile correspondante
:Pt-On(X,Y,A*8+Pic1
```

L'écran est composé de 96 pixel fois 64 pixels : soit 12*8 pixels et 8*8 pixels. On peut donc faire une **carte 12*8** composé de tiles 8*8 😊

Dans mon cas, je mettrai une maison au milieu et de l'herbe partout autour :

Code : Axe

```
: [000000000000]→GDB1
: [000000000000]
: [000000000000]
: [000001200000]
: [000003400000]
: [000000000000]
: [000000000000]
: [000000000000]
:.8 ligne fois 12 caractères par ligne.
```

Et voilà, tous les éléments sont réunis pour afficher notre tilemap !

Affichage de la tilemap

C'est là que les choses se corsent, on a pour l'instant ce code :

Code : Axe

```
: [0044888800112222]→Pic1
: Herbe
: [0107196181818181]
: Maison haut/gauche
: [80E0988681818181]
: Maison haut/droite
: [818698E09E929692]
: Maison bas/gauche
: [8161190771517101]
: Maison bas/droite
:
: [000000000000]→GDB1
: [000000000000]
: [000000000000]
: [000001200000]
: [000003400000]
: [000000000000]
: [000000000000]
```

La méthode que je vous présente consiste à afficher ligne par ligne chaque tile 8*8 à son emplacement donné dans l'écran comme vu plus haut (12*8). Pour cela j'utiliserai un système de double boucle avec les variables X et Y :

Code : Axe

```
:For (Y,0,7)
:For (X,0,11)
:Pt-On(X*8,Y*8,EXP)
:End
:End
```

EXP désignant l'expression (le calcul) à faire pour obtenir la tile voulu.



On peut faire ça par calcul ? Comment faire ? 😊

Une commande nous arrive alors, tout cuit dans le bec : la commande *nib{ }*. Cette commande est très utile dans le tilemapping car elle permet de récupérer un **quartet** (en Anglais *nibble*), soit une moitié d'octet, donc un seul caractère hexadécimal.



On va donc passer en argument à la commande l'adresse du quartet à renvoyer.



Dans la mesure où il y a 2 fois plus de quartets que d'octets (car 1 octet = 2 quartets), il faut multiplier le pointeur par 2.

Par exemple, pour avoir le deuxième quartet de GDB1 (pour nous la deuxième tile à afficher), on va procéder comme ceci :

Code : Axe

```
nib{GDB1*2+1}
```



Attention aux priorités de calcul ! C'est uniquement GDB1 qu'il faut multiplier par 2, pas toute l'expression !

Grâce à cette commande, nous allons pouvoir extraire toute notre tilemap à l'aide d'une seule expression. Et n'attendons plus, cette expression est :

```
nib{lignes * tilesParLigne + colonne + (ptrMap * 2)}
```

Lorsque le nombre de *tilesParLigne* est paire on peut simplifier en :

```
nib{tilesParLigne / 2 * lignes + ptrMap * 2 + colonne}
```

Vous ne pouvez que comprendre puisqu'on vient à l'instant même de dire comment cela fonctionnait. 😊

Si on reprend notre boucle, Y indique le numéro de la ligne et X le numéro de la colonne. Quand au nombre de tiles par ligne il va être de 12 quartets, soit **6 octets**. On obtient finalement :

Code : Axe

```
:.TILEMAP
:
:[0044888800112222]→Pic1
:.Herbe
:[0107196181818181]
:.Maison haut/gauche
:[80E09886818181]
:.Maison haut/droite
:[818698E09E929692]
:.Maison bas/gauche
:[8161190771517101]
:.Maison bas/droite
:
:[000000000000]→GDB1
:[000000000000]
:[000000000000]
:[000001200000]
:[000003400000]
:[000000000000]
:[000000000000]
:
:For (Y,0,7)
:For (X,0,11)
:nib(Y*6+GDB1*2+X)→A
:Pt-On (X*8,Y*8,A*8+Pic1)
:End
:End
:
:.Une boucle pour l'affichage
:Repeat getkey(15)
:DispGraph
:End
```

Un petit screen pour donner un aperçu :



Voici votre première tilemap !

Encore plus de Data !

Par la suite on utilisera les Datas pour faire de la musique, des jeux multijoueurs,... ils envahiront vos programmes. 😊

Pour vous faciliter la vie, il existe plusieurs outils pour manipuler et exploiter les Datas au maximum.

Modifier nos Datas

Je vous ai déjà dit que nos datas pouvaient se manipuler comme des variables d'un octet, mais je ne vous ai pas montré d'exemple :

Code : Axe

```
:[09]→GDB9          :.L'octet que pointe GDB9 vaut 9
:Disp {GDB9}►Dec    :.On affiche 9
:18→{GDB9}          :.On modifie la valeur du premier octet pointé par GDB9
:Disp {GDB9}►Dec    :.On affiche 18
```



Ouais bah ce sont des simples variables. 😊

Non !

En fait, le gros avantage des Datas c'est que les données enregistrées seront toujours stockées dans l'exécutable. Du coup, on peut les modifier, quitter le programme, éteindre la calculatrice, la rallumer, revenir dans le programme, et elles seront toujours là !



Certains shells démarrent une copie de votre programme, donc **ceci ne marchera pas**. De même pour les applications qui sont démarré à partir de la mémoire flash.



Et je fais comment si ma Data doit contenir une valeur supérieure à 255 ? 😊

On peut également manipuler 2 octets à la fois pour avoir un nombre entre 0 et 65535. Votre Data devra juste avoir un **r** juste après la fermeture des accolades.

Code : Axe

```
: [FFFF]→GDB9
:
:Disp {GDB9}►Dec      ::Affiche 255
:Disp {GDB9}↑►Dec     ::Affiche 65535
```

Pour modifier sa valeur c'est exactement pareil :

Code : Axe

```
:10012→{GDB9} r
```

On peut même manipuler 2 octets... inversés 😊. Il faut pour cela utiliser un deuxième **r** à la suite du premier :

Code : Axe

```
: [0100]→GDB9
:
:Disp {GDB9}↑►Dec    ::Affiche 1
:Disp {GDB9}↑↑►Dec   ::Affiche 256 (sois 0100 en hexadécimal)
```

La commande *length()*

La commande *length()* (signifiant littéralement "longueur") permet de calculer le nombre d'octets qui suivent à partir d'un octet donné, et jusqu'au prochain octet qui vaut 0 non compris.

Voici un exemple concret d'utilisation :

Code : Axe

```
:Data(11,22,42,31,0)→Pic12
:length(Pic12)→A
:For(Z,0,A)
:Output(0,Z,{Pic12+Z}►Dec
:End
:.11, 22, 42, 31 et 0 sont affichés sous forme de colonne
```



Le nombre d'octets trouvé déterminera le nombre de fois que la boucle va se répéter en affichant à chaque fois un nouvel octet.

Rechercher une valeur dans les Data

De la même manière que la commande *length()*, la commande *inData()* permet de trouver l'emplacement de l'octet cherché :

Code : Axe

```
:Data(11,22,42,31,0)→Pic12
:inData(42,Pic12)→A
:For(Z,0,A-1)
:Output(0,Z,{Pic12+Z}►Dec
:End
:.11, 22 et 42 sont affichés sous forme de colonne
```



Si l'octet recherché n'est pas trouvé, alors la valeur 0 est renvoyé, ou sinon la commande renvoie l'emplacement de l'octet trouvé en commençant par 1, puis 2, puis 3, etc.

Avant de clôturer ce chapitre, je tiens à vous montrer un code grandement simplifiable grâce à la commande *inData()* :

Code : Axe

```
:If A=1 or (A=3) or (A=10) or (A=12)
:Code
:End
```

Ce code peut donc s'écrire comme ceci :

Code : Axe

```
:If inData(A,Data(1,3,10,12,0))
:Code
:End
```

Cela ne marche que si A est différent de zéro. Or ce zéro est indispensable, sinon il recherchera indéfiniment dans la mémoire, jusqu'à rencontrer un autre zéro.

Ce chapitre est sans doute le plus difficile pour cette deuxième partie ; ne le négligez pas car c'est ici que repose une grande partie de la puissance de l'Axe Parser.

Plus tard vous apprendrez même à faire des data pour la musique, les grayscales,... le principe reste le même, mais l'utilisation est différente.

Les listes

 Mes programmes commencent à se remplir de variables, je ne sais plus où en trouver !? 

Pas de problème, les listes sont là pour vous. 

Vous découvrirez dans ce chapitre l'utilité des listes et leurs particularités souvent très utiles !

présentation des listes

Une liste est une partie de la mémoire dans laquelle on peut stocker des informations durant l'exécution du programme. C'est comme des Datas qu'on ne voudrait pas garder.
Il y en a 6 en tout, et chacune est utilisée régulièrement par la calculatrice : en fonction de vos programmes toutes les listes ne sont pas exécutables.

La liste 1 L₁ : saveSScreen

L₁ se trouve ici : 

"saveSScreen" pour les intimes, cette liste est la plus sûre. Elle permet d'utiliser 714 octets.

 C'est une variable de 714 octets ? 

Non, L₁ est un simple pointeur, les octets qui le suivent s'utilisent comme les Datas :

Code : Axe

```
:12→{L1}
:Disp {L1}►Dec
:.Affiche 12
:
:42→{L1+1}
:Disp {L1+1}►Dec
:.Affiche 42
```

Les octets d'une liste sont toujours inconnus au démarrage d'un programme, donc il est recommandé de les initialiser au démarrage du programme. Dans le cas où l'on veut utiliser tous les octets de L₁ :

Code : Axe

```
:For (Z, 0, 713)
:0→{L1+Z}
:End
```

 Il ne faut surtout pas dépasser les 714 octets réservés, car je ne peux pas prédire de ce qu'il va arriver : la mémoire de votre calculatrice risque d'être corrompue et cela peut provoquer au mieux des *ram cleared*, et au pire... rien de très sympathique  (cela s'applique également pour toutes les autres listes).

La liste 2 L₂ : statVars

De son vrai nom "statVars", cette liste a une longueur de 531 octets. Il y a un risque si elle est utilisée en même temps qu'une interruption personnalisée, notamment si vous utilisez MirageOS.

La manière de l'utiliser est la même que pour toutes les autres listes, ici l'exemple de l'initialisation :

Code : Axe

```
:For (Z, 0, 530)
:0→{L2+Z}
:End
```

L₂ se trouve ici : 

La liste 3 L₃ : appBackUpScreen

En fait vous connaissez déjà cette liste... mais sous un autre nom : le back-buffer. En effet, cette mémoire permet de sauvegarder l'écran entièrement, elle est constituée de 768 octets.

 Cela veut dire que l'écran fait 768 octets ?

Exactement ! 

Rappelez vous, 96*64=6144 pixels, soit 6144 bits. Donc pour trouver combien cela fait en octet, il faut faire 6144/8 soit 768.

Du coup, cette liste ne doit pas être utilisée si vous utilisez le back-buffer pendant votre programme.

Le code pour initialiser serait :

Secret (cliquez pour afficher)

Code : Axe

```
:For (Z, 0, 767)
:0→{L3+Z}
:End
```

L₃ se trouve ici : 

La liste 4 L₄ : tempSwapArea

Cette liste est utilisée lorsque vous archivez ou désarchivez un programme par exemple (chose que nous verrons dans la troisième partie ). Elle permet d'utiliser 256 octets, donc en reprenant notre exemple d'initialisation :

Secret (cliquez pour afficher)

Code : Axe

```
:For(Z,0,255)
:0→{L4+Z}
:End
```

L₄ se trouve ici :

La liste 5 L₅ : textShadow

Lorsque vous affichez du texte à l'écran, c'est cette liste qui est utilisée : il est donc recommandé de ne pas afficher de texte à l'écran pendant son utilisation (mais on peut l'utiliser distinctement 😊). Elle fait 128 octets.

Secret (cliquez pour afficher)**Code : Axe**

```
:For(Z,0,127)
:0→{L5+Z}
:End
```

L₅ se trouve ici :

La liste 6 L₆ : plotSScreen

Cette liste est la plus utilisée de toutes : c'est la mémoire du buffer. Celui-ci est utilisé fréquemment dans les programmes en Axe, du coup son utilisation reste assez spécifiée. Comme pour le back-buffer, il y a 768 octets :

Secret (cliquez pour afficher)**Code : Axe**

```
:For(Z,0,767)
:0→{L6+Z}
:End
```

L₆ se trouve ici :

Les tableaux

Les tableaux, ce n'est qu'une manière particulière de manipuler les listes... ou les Datas ! Cela ne vous rappelle rien ?

Secret (cliquez pour afficher)

Lorsque vous afficher une tilemap dans le buffer :

Code : Axe

```
:For(Y,0,7)
:For(X,0,11)
:{Y*6+X+GDB1}→A
:...
:End
:End
```

Ceci est un tableau d'abscisse X et d'ordonnée Y ! 😊

En général on établit un tableau de la manière suivante : {<Numéro de la ligne>*<Nombre total de colonne>+<Numéro de la colonne><Pointeur>} :

Soit en Axe :

Code : Axe

```
:...
:GDB1 le pointeur
:Y le numéro de la ligne
:X le numéro de la colonne
:T le total de colonne par ligne
:...
:{Y*T+X+GDB1}
```

Pour naviguer dans le tableau, il suffit de modifier les axes X et Y.



Vous pouvez même créer des tableaux à trois dimensions, ou même plus. 😊

D'autres commandes utiles

Toutes les commandes vues pour les Datas peuvent être utilisées pour les listes. De même les commandes qui vont suivre peuvent être utilisées pour les Datas.

Copier les octets avec la commande Copy

Cette commande permet de copier un nombre d'octets à partir d'un pointeur, vers le même nombre d'octets suivant un autre pointeur.



Copy(se trouve ici :

Elle prend donc trois arguments, dans l'ordre : le pointeur de départ (Strl), le pointeur d'arrivé (L₁), le nombre d'octet (A) :

Code : Axe

```
:Copy(Str1,L1,A)
```



Le nombre d'octets doit être strictement au dessus de 0.

Par exemple, je peux recréer facilement la commande *RecallPic* :

Code : Axe

```
:Copy (L3,L6,768)
```

De la même manière, il existe une commande pour copier les octets à partir de la fin : *Copy()*



Échanger les octets

Oui, on peut même échanger les octets ! Les arguments sont exactement les mêmes que pour la commande *Copy()*, seulement il faut utiliser la commande *Exch()* :



et appuyez sur

Reprenez notre buffer et notre back-buffer, il n'existe pas de commande implémentée pour les échanger, on peut donc facilement créer une fonction :

Code : Axe

```
:: Échanger le buffer et le back-buffer
:Lbl CHB
:Exch (L3,L6,768)
:Return
```

Remplir les octets

La commande *Fill()* est de celles qui nous facilitent la vie . Elle permet de remplir à partir d'un octet, tous ceux qui suivent (jusqu'à un nombre en argument) par la valeur du premier. Traduction, le code que l'on a vu pour initialiser les valeurs de L1 à 0 :

Code : Axe

```
:For (Z,0,713)
:0→{L1+Z}
:End
```

Peut s'écrire plus simplement avec la commande *Fill()* :

Code : Axe

```
:0→{L1}
:Fill (L1,713)
```



Fill() se trouve ici :



Votre cours pour la deuxième partie est terminé... mais un petit TP vous attend ! Relisez une dernière fois ce chapitre avant de continuer.

TP n°2 : en quête de l'échec

Ce TP n'est pas seulement une restitution des connaissances apprises durant cette partie, mais une exploitation pure et dure : vous allez surer devant vos lignes de codes. 🧙

Vous devrez créer un mini RPG qui vous servira de base dans la création de beaucoup de vos jeux par la suite.

Présentation du jeu

 Avant de commencer, je tiens à préciser que je ne vous impose pas un jeu à créer, mais que je vous donne une idée de jeu. Dès que vous avez une idée autre que celle évoquée, n'hésitez pas à tester, puis appliquer ! 😊

Le contexte

C'est très important quand vous créez un jeu de ce genre : il vous faut un imaginaire, un contexte. Dans mon exemple, Zozor a perdu son échiquier. Pour le retrouver, il doit sortir de sa prairie et récupérer le maximum de zéros (le numéro) possible.

Les tiles et les sprites

Après avoir déterminer le contexte, il vous faudra dessiner vos propres sprites !



Moi dessiner ?💡

Rassurez vous, si le dessin n'est pas votre tasse de thé, je vous ai prévu quelques sprites et tiles d'urgence :

Secret ([cliquez pour afficher](#))

- Le sprite de Zozor :

Pxl	1	2	3	4	5	6	7	8
1			■					■
2				■	■	■	■	
3			■					■
4			■			■	■	■
5			■		■		■	■
6	■	■	■					■
7	■	■	■	■	■	■	■	
8	■		■					

- La tile de l'herbe :

Pxl	1	2	3	4	5	6	7	8
1								
2			■					
3		■						
4		■				■		
5					■			
6						■		
7								
8								

- La tile d'un mur :

Pxl	1	2	3	4	5	6	7	8
1			■				■	
2	■	■	■	■	■	■	■	■
3	■				■			
4	■	■	■	■	■	■	■	■
5			■				■	
6	■	■	■	■	■	■	■	■
7	■				■			
8	■	■	■	■	■	■	■	■

- La tile d'une barrière de clairière (horizontal) :

Pxl	1	2	3	4	5	6	7	8
1								
2			■				■	
3	■	■		■	■	■		■
4		■		■		■		■
5	■	■		■	■	■		■
6		■		■		■		■
7	■	■		■		■		■
8								

- La tile d'une barrière de clairière (vertical) :

Pxl	1	2	3	4	5	6	7	8
1								
2								
3								
4								
5								
6								
7								
8								

1					■	■		■
2					■	■	■	
3			■		■	■	■	
4	■		■	■	■	■		■
5	■		■	■	■	■		■
6	■		■	■	■	■		■
7	■		■	■	■	■		
8	■		■	■	■			

- La tile d'une porte prête à être mise dans un mur :

Pxl	1	2	3	4	5	6	7	8
1			■				■	
2	■	■	■	■	■	■	■	■
3	■						■	
4	■						■	■
5	■				■		■	
6	■			■		■	■	
7	■				■			
8	■				■	■		

Le déroulement du jeu

Lorsque l'on démarre le jeu, il faut arriver dans un menu avec de beaux graphismes. On devra pouvoir choisir, à l'aide d'un curseur, si l'on veut jouer, quitter le jeu, ou aller dans les crédits.

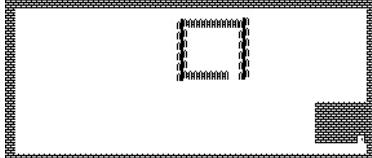
Remarque : pour rendre le menu plus intuitif, on devra pouvoir choisir avec la touche **2nde 2ND** ou la touche



. Et l'on pourra quitter le jeu dès que la touche **annul CLEAR** est pressée.

Dans le jeu, notre zozor doit démarrer au milieu de sa prairie. La tilemap devra être plus grande que l'écran !

Pour vous aider, voici un exemple de tilemap avec les tiles vues plus haut :



Encore une fois, je vous encourage à créer votre propre tilemap.

Votre Zozor doit donc pouvoir se balader, et le décor du fond doit défiler en même temps. Zozor doit pouvoir atteindre les 4 angles de la tilemap !

Des zéros seront dispersés un peu partout sur la carte, et dès que Zozor passe sur l'un d'entre eux, celui-ci disparaît. Lorsque tous les zéros ont été récupérés, un passage s'ouvre et permet d'accéder au jeu d'échec !

Les ajouts possibles

Pour rendre le jeu encore plus complet, il faudrait rajouter quelques trucs :

- Passage dans la cave via la porte au bout d'un certain nombre de zéro trouvé. (voir [biographie de Zozor](#))
- Afficher du texte dans l'introduction et pendant le jeu (pour suivre l'imaginaire).
- Afficher la tête de Zozor vers la gauche quand on va vers la gauche, et vers la droite quand on va vers la droite.

Quelques commandes et conseils utiles

Pour faire un menu digne de ce nom

Je vous conseille d'importer une Pic de [votre calculatrice](#) que vous aurez dessinée au préalable.



Comment est ce qu'on fait ça ?

Il suffit d'indiquer la variable de votre Pic entre crochet (par exemple : `[Pic1]`), et de la pointer. Votre image va donc être intégrée dans votre executable, sous forme de Data. Pour l'afficher sur l'écran, une commande est toute faite : il suffit d'écrire la commande `DispGraph` et spécifier entre parenthèses le pointeur de votre Pic :

Code : Axe

```
:[Pic1]→Pic1
:Le pointeur peut être Pic1, ou n'importe quoi d'autre
:
:DispGraph(Pic1)
```



Mais moi je ne sais pas dessiner sur des Pic de la calculatrice 😊.

Il vous suffit de dessiner à l'aide de l'éditeur d'image de votre calculatrice [en dehors d'un programme](#) :



C
prgm



Puis de l'enregistrer à l'aide de la commande *StorePic* :



Pour rappeler l'image il faudra utiliser la commande *RecallPic*, juste en dessous *StorePic*. Si les axes vous gênent, désactivez les :



Je vous recommanderai cependant de modifier les images 96*64 à partir de Paint, et de les convertir en Pic grâce à TI-Connect (voir annexe).

S'organiser



Il y a tellement de chose à faire... mais par où commencer ? 🤔

Tout d'abord, relisez bien encore une fois ce qui est demandé, soufflez un bon coup et regardez ce qui vous semble le plus dur à faire.

Si vous ne voyez vraiment pas par quoi commencer, je vous conseille de convertir les tiles en hexadécimal, puis de créer la tilemap... avec le moteur de déplacement qui va avec (là ça vous demandera un peu de réflexion 😊).

Une fois que vous avez vraiment un truc stable et opérationnel, vous pourrez commencer à rajouter Zozor, puis gérer les collisions de celui-ci avec en fonction de la tilemap. Après pourquoi ne pas s'attaquer au menu, ainsi qu'au crédit si vous avez du temps.

Puis il vous faudra créer les tiles des zéros (le modèle à suivre : "0" 🎯), et les afficher (sans créer de conflit avec la tilemap), il faudra trouver comment permettre d'accéder au jeu d'échec lorsque les 4 zéros ont été découverts.

Si vous êtes vraiment motivé, vous pourrez créer une cave, qui permettra d'étendre les recherches de Zozor.

Correction



La correction proposée ici est en fait un exemple de résultat. Chacun programme à sa manière, et votre résultat sera donc forcément différent du mien.

J'ai organisé mon code en 3 grandes parties :

- Le menu
- Le système de jeu
- Tout ce qui touche à la tilemap

De cette manière, la tilemap est placée en fin de source, et l'on peut avoir accès assez rapidement au code important lorsqu'on édite sur calculatrice (c'est une simple habitude de développement).

Les tiles

D'abord, j'ai commencé par convertir les tiles et les sprites vues plus haut :

Secret ([cliquez pour afficher](#))

- Le sprite de Zozor :

Pxl	1	2	3	4	5	6	7	8
1	■	■	■	■	■	■	■	■
2	■	■	■	■	■	■	■	■
3	■	■	■	■	■	■	■	■
4	■	■	■	■	■	■	■	■
5	■	■	■	■	■	■	■	■
6	■	■	■	■	■	■	■	■
7	■	■	■	■	■	■	■	■
8	■	■	■	■	■	■	■	■

Son code :

Code : Axe

211E212737E1FEA0

- La tile de l'herbe :

Pxl	1	2	3	4	5	6	7	8
1	■	■	■	■	■	■	■	■
2	■	■	■	■	■	■	■	■
3	■	■	■	■	■	■	■	■
4	■	■	■	■	■	■	■	■
5	■	■	■	■	■	■	■	■
6	■	■	■	■	■	■	■	■
7	■	■	■	■	■	■	■	■
8	■	■	■	■	■	■	■	■

Son code :

Code : Axe

0020404204040000

- La tile d'un mur :

Pxl	1	2	3	4	5	6	7	8
1	■	■	■	■	■	■	■	■
2	■	■	■	■	■	■	■	■
3	■	■	■	■	■	■	■	■
4	■	■	■	■	■	■	■	■
5	■	■	■	■	■	■	■	■
6	■	■	■	■	■	■	■	■
7	■	■	■	■	■	■	■	■
8	■	■	■	■	■	■	■	■

Son code :

Code : Axe

: [22FF88FF22FF88FF]

- La tile d'une barrière de clairière (horizontal) :

Pxl	1	2	3	4	5	6	7	8
1	■	■	■	■	■	■	■	■
2	■	■	■	■	■	■	■	■
3	■	■	■	■	■	■	■	■
4	■	■	■	■	■	■	■	■
5	■	■	■	■	■	■	■	■
6	■	■	■	■	■	■	■	■
7	■	■	■	■	■	■	■	■
8	■	■	■	■	■	■	■	■

Son code :

Code : Axe

: [002255DD55DD5555]

- La tile d'une barrière de clairière (vertical) :

Pxl	1	2	3	4	5	6	7	8
1				■	■		■	
2				■	■	■		■
3		■		■	■	■		■
4	■		■	■	■	■		■
5	■		■	■	■	■		■
6	■		■	■	■	■		■
7	■		■	■	■			
8	■		■	■	■			

Son code :

Code : Axe

: [1A1D5DBDBDBDB8B8]

- La tile d'une porte prête à être mise dans un mur :

Pxl	1	2	3	4	5	6	7	8
1			■			■		
2	■	■	■	■	■	■	■	■
3	■					■		
4	■					■		■
5	■				■		■	
6	■				■		■	
7	■					■		
8	■					■		■

Son code :

Code : Axe

: [22FF82838A8B8283]

Mais j'ajoute en plus un sprite pour les zéros, et une tile pour le jeu d'échec :

[Secret \(cliquez pour afficher\)](#)

Pxl	1	2	3	4	5	6	7	8
1								
2			■	■	■			
3		■				■		
4		■				■		
5		■				■		
6		■				■		
7			■	■	■	■		
8								

Son code :

Code : Axe

: [003C424242423C00]

Pxl	1	2	3	4	5	6	7	8
1		■		■		■		■
2	■		■		■		■	
3		■		■		■		■
4	■		■		■		■	
5		■		■		■		■
6	■		■		■		■	
7		■		■		■		■
8			■	■	■	■		

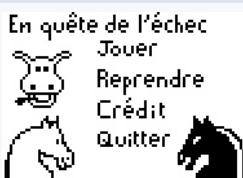
8							
---	--	--	--	--	--	--	--

Son code :
Code : Axe
: [55AA55AA55AA55AA]

Le menu

Pour le menu, j'ai choisi de créer une image principale avec les différents choix : jouer, reprendre, crédit et quitter :

Secret (cliquez pour afficher)



Pour la convertir, il faut passer par un outil de TI-Connect, voir [annexe sur l'utilisation de TI-Connect](#).

Pour sélectionner un choix, j'ai pensé mettre un curseur, sous forme d'un sprite 8*8 :

Secret (cliquez pour afficher)

Pxl	1	2	3	4	5	6	7	8
1								
2					■			
3					■	■		
4					■		■	
5					■			■
6					■		■	
7					■	■		
8					■			

Son code :
Code : Axe
: [00080C0A090A0C08]

Pour pouvoir manipuler le sprite en même temps que le menu, j'utilise la commande vue plus haut, combinée à un *StoreGDB* qui va enregistrer l'écran dans le buffer. Comme cette commande va être utilisée plusieurs fois, je la mets dans une fonction :

Code : Axe

```
:Lbl MENU
:DispGraph(Pic0MENU)
:StoreGDB
:Return
```

Pour le système du menu, j'utilise la variable Θ (théta), qui va contenir le choix en temps réel : 0 pour **jouer**, 1 pour **reprendre**, 2 pour **crédit** et 3 pour **quitter**.

On modifie Θ en fonction des touches et :

Code : Axe

```
:<3 and getKey(1) - (<0 and getKey(4)) + <->
```

Pour éviter de défiler les choix trop rapidement, on fera appeler à une fonction qui attend que plus aucune touche ne soit pressée. On l'appellera très souvent :

Code : Axe

```
:Lbl NOKEY
:While getKey(0)
:Pause 5
:End
:Return
```



La *Pause 5* permet d'éviter les faux contacts (sur les calculatrices un peu vieilles comme la mienne 😊).

Pour le sprite du curseur, j'utilise ici la commande *Pt-Change()*, ce qui permet de ne pas modifier le buffer si on inverse une fois avant le *DispGraph* et qu'on ré-inverse après ! Puis, en prenant les repères sur mon image de menu, le curseur doit être placé à 26 sur l'axe des X. Sur l'Axe des Y, l'écart est de 12 pixels entre chaque choix, donc Le premier choix se trouvant à 9 pixels, on trouve $\Theta * 12 + 9$:

Code : Axe

```
: [00080C0A090A0C08]--Pic0CURS
:Pt-Change(26,θ*12+9,Pic0CURS
:DispGraph
:Pt-Change(26,θ*12+9,Pic0CURS
```

Le choix se validera en appuyant sur la touche  ou , on retrouvera ces deux touches à divers endroits

par la suite, on peut donc les mettre dans une fonction :

Code : Axe

```
:Lbl 954
:getKey(9) or getKey(54)
:Return
```

Notre choix sera définitif quand une de ces touches sera pressée ! A ce moment là, si θ vaut 0, on joue, si il vaut 1, on reprend le jeu, si θ vaut 2, on affiche le crédit.:

Code : Axe

```
:If sub(954)
:!If θ
:sub(GAME)
:ElseIf θ=1
:sub(LOAD)
:ElseIf θ=2
:sub(NOKEY)
:ClrHome
:Output(0,0,"Janvier 2011 Pour le SdZ"
:sub(PAUSE)
:End
```

Les deux fonctions GAME et LOAD seront vues par la suite.

Pour le crédit, on attend que plus aucune touche ne soit pressée (la fonction *NOKEY*), puis on efface l'écran pour afficher le texte et faire une PAUSE :

Code : Axe

```
:Lbl PAUSE
:Repeat sub(954)
:End
:Return
```

De cette manière, on attend que la touche  ou  est pressée !

On met le tout dans une boucle qui quittera le programme quand on appuie sur la touche , ou si le choix **quitter** est

validé :

Le code final du menu :

Secret ([cliquez pour afficher](#))

Code : Axe

```
:[Pic1]--Pic0MENU
:[00080C0A090A0C08]--Pic0CURS
:
:sub(MENU)
:0→θ
:Repeat θ=3 and sub(954) or getKey(15)
:θ<3 and getKey(1)-(θ>0 and getKey(4))+θ→θ
:sub(NOKEY)
:Pt-Change(26,θ*12+9,Pic0CURS
:DispGraph
:Pt-Change(26,θ*12+9,Pic0CURS
:If sub(954)
:!If θ
:sub(GAME)
:ElseIf θ=1
:sub(LOAD)
:ElseIf θ=2
:sub(NOKEY)
:ClrHome
:Output(0,0,"Janvier 2011 Pour le SdZ"
:sub(PAUSE)
:End
:sub(MENU)
:End
:End
:Return
:
:Lbl PAUSE
:Repeat sub(954):End
:Return
:
:Lbl 954
:getKey(9) or getKey(54)
:Return
:
:Lbl MENU
:DispGraph(Pic0MENU)
:StoreGDB
:Return
:
:Lbl NOKEY
:While getKey(0)
:Pause 5
```

```
:End
:Return
```

Et voici un petit aperçu de ce que cela devrait donner :



Le tlemapping... et le scrolling

Dans mon cas, j'ai choisi de faire une map 36 de largeur fois 16 tiles de hauteur. Pour pouvoir afficher toute la tilemap, il fallait donc réfléchir à un moyen de la faire défiler : c'est ce qu'on appelle le **scrolling**.

En reprenant les tiles et la tilemap vues plus haut, j'ai organisé mes Datas de la façon suivante :

Code : Axe

```
:L'herbe
:[0020404204040000]→Pic1
:
:.Le sol (pas utilisé ici)
:[0000000008000000]
:
:.Le mur
:[22FF88FF22FF88FF]
:
:.Le jeu d'échec
:[55AA55AA55AA55AA]
:
:.La barrière horizontale
:[002255DD55D5555]
:
:.La barrière verticale
:[1A1D5DBDBDBDB8B8]
:
:.La porte (pas utilisé ici)
:[22FF82838A8B8283]
:
:
:[22222222222222222222222222222222]→GDB1
:[200020000000000000000000000000002]
:[20002000000000000544445020000022002]
:[202020000000000005000005020000220022]
:[20222000200000000500000502222200222]
:[200000000000000005000005000000000002]
:[200000000000000005000005000000000002]
:[200000000000000005444405000000000002]
:[200000000000000005444405000000000002]
:[200000000000000005000000000000000002]
:[200000000000000005000000000000000002]
:[200000000000000005000000000000000002]
:[202220000000000000000000000000002222222]
:[202020000000000000000000000000002222222]
:[2000200000000000000000000000000020000032]
:[222222222222222222222222222222222222]
```

C'est là que les problèmes commencent. 😱

En fait, il faut juste voir la situation clairement :

- On sait modifier des coordonnées A et B en fonction des flèches directionnelles.
- On sait extraire un octet en fonction des axes X et Y.
- On sait afficher une tilemap par extraction d'octet.

Quelque chose ne vous saute pas à l'esprit ? 🤔

Secret (cliquez pour afficher)

A et B peuvent être mélés respectivement à X et Y !

La tilemap appellera donc une fonction ZIP, en envoyant les arguments X+A et Y+B :

Code : Axe

```
:Lbl TLMAP
:For(Y,0,7)
:For(X,0,11)
:.rl vaut la valeur de l'octet extrait
:sub(ZIP,X+A,Y+B)→rl
:Pt-On(X*8,Y*8,rl*8+Pic1
:End
:End
:Return
```

L'extraction se fait en fonction de ces deux arguments comme pour une tilemap classique :

Code : Axe

```
:Lbl ZIP
:nib(r2*18+GDB1*2+r1)
:Return
```

Quand on a fait ça, on a pausé les bases qui serviront au déplacement du personnage !

Pour se déplacer, il ne faut pas oublier de gérer les collisions avec la tilemap. Pour cela, notre fonction *ZIP* est encore très utile : en fonction de la direction choisie, on anticipe si une tile se trouve à cette endroit :

Code : Axe

```
:Lbl MOV
:If getKey(3) and (sub(ZIP,A+1,B)=0
:A++
:ElseIf getKey(2) and (sub(ZIP,A-1,B)=0
:A--
:End
:If getKey(1) and (sub(ZIP,A,B+1)=0
:B++
:ElseIf getKey(4) and (sub(ZIP,A,B-1)=0
:B--
:End
:Return
```

Le problème avec ce code est que quand vous arrivez aux angles de la map, il affiche une partie de votre mémoire sous forme de tile (car on a rien précisé pour ce qu'il y a après la "tilemap"). Vous risquez de voir quelque chose du genre :



Pour régler ce problème, j'utilise une petite astuce dans la condition : j'ajoute des "murs" pour que la tilemap ne dépasse pas les dimensions imposées. Par exemple, après le *getKey(3)* and (*sub(ZIP,A+U+1,B+V)=0*) je rajoute un *If A<24*.

? Comment notre Zozor va-t-il atteindre les angles alors ?

C'est là qu'interviennent deux autres variables : U et V. Ce sont les coordonnées de Zozor... sur l'écran. Zozor sera donc affiché comme cela :

Secret (cliquez pour afficher)

```
{211E212737E1FEA0}→Pic1ZZ
:Pt-Off(U*8,V*8,Pic1ZZ)
```

Zozor est donc au milieu de l'écran quand *U=5* et *V=4*. U doit être compris entre 0 et 11, et V entre 0 et 7. On adapte tout ça en fonction de chaque direction, et voici notre moteur de déplacement :

Code : Axe

```
:Lbl MOV
:If getKey(3) and (sub(ZIP,A+U+1,B+V)=0
:If A<24 and (U=5)
:A++
:ElseIf U<11
:U++
:End
:End
:ElseIf getKey(2) and (sub(ZIP,A+U-1,B+V)=0
:If (U=5) and (A>0)
:A--
:ElseIf U>0
:U--
:End
:End
:If getKey(1) and (sub(ZIP,A+U,B+V+1)=0
:If B<8 and (V=4)
:B++
:ElseIf V<7
:V++
:End
:ElseIf getKey(4) and (sub(ZIP,A+U,B+V-1)=0
:If B>0 and (V=4)
:B--
:ElseIf V>0
:V--
:End
:End
:Return
```



Ce code peut être placé avant la tilemap, car il n'y a que la fonction *ZIP* et *TLMAP* qui ont besoin des Data.

Vous devriez avoir un Zozor qui se déplace comme ça :



Pour faire apparaître Zozor au milieu de la clairière, il faut initialiser les variables A, B, U et V au début du jeu :

Code : Axe

```
:0→B
:14→A
:4→V
:5→U
```

Les zéros

Pour rendre le jeu interactif, il y a un défi à relever : récupérer tous les zéros. Dans mon cas, j'en ai mis 4. Chaque zéro a ses coordonnées, et son état (0 = il apparaît, 1 = il n'apparaît pas). Tout ça est géré avec la liste 1, donc on aura un tableau du genre $\{Y^*3+X+L_1\}$, où les abscisses seront manipulables quand $X=0$, les ordonnées quand $X=1$ et l'état quand $X=2$. À chaque début de partie, j'initialise l'état, et les coordonnées de chaque zéro :

Code : Axe

```
:L'état de chacun est à 0
:0→{L1+2}→{L1+5}→{L1+8}→{L1+11}
:
:Je mets à peu près un zéro à chaque angle
:3→{L1}→{L1+1}→{L1+3}→{L1+7}
:13→{L1+4}→{L1+10}
:26→{L1+6}→{L1+9}
```

Pour manipuler tous les zéros en même temps, on fait une boucle *For*:

Code : Axe

```
:Lbl 0
:.Rappel, r1 est une simple variable
:For(r1,0,3)
:
:End
:Return
```

Pour afficher chaque zéro, il existe une astuce : faire la différence des coordonnées du zéro, par celles du personnages sur la tilemap :

Code : Axe

```
:{r1*3+L1}-A→r2
:{r1*3+L1+1}-B→r3
```

Mais comme le code $r1*3+L1$ reviendra très fréquemment dans notre boucle *For*, on peut simplifier en mettant ce calcul dans une variable temporaire *r4* :

Code : Axe

```
:{r1*3+L1-r4}-A→r2
:{r4+1}-B→r3
```

r2 contient l'abscisse du zéro, et *r3* son ordonnée... sur l'écran !

On vérifie donc que notre Zozor est arrivé sur le zéro, et on change son état si c'est le cas :

Code : Axe

```
:If r2=U and (r3=V) and ({r4+2}=0)
:{r4+2}++
:End
```

Si il n'a toujours pas d'état, et qu'il rentre dans le cadre de l'écran, alors on l'affiche sur le buffer :

Code : Axe

```
:!If {r4+2}
:If r2<12 and (r3<8
:Pt-Off(r2*8,r3*8,[003C424242423C00]
:End
:End
```

Pour savoir si les 4 zéros ont leur état à 1, j'utilise une variable *r5* qui s'incrémentera si l'état du zéro est à 1.

Code : Axe

```
:If {r4+2}
:r5++
:End
```

Et après la boucle je mets ce code :

Code : Axe

```
:If r5=4
:0→{266+GDB1}
:End
```

266 octets après le pointeur *GDB1* correspond à l'endroit où se trouve le mur qui bloque en bas à droite. Notre fonction pour manipuler les zéros donne au final :

Code : Axe

```
:Lbl 0
:0→r5
:For(r1,0,3)
:{r1*3+L1-r4}-A→r2
:{r4+1}-B→r3
:If r2=U and (r3=V) and ({r4+2}=0)
:{r4+2}++
:End
:!If {r4+2}
:If r2<12 and (r3<8
:Pt-Off(r2*8,r3*8,[003C424242423C00]
:End
```

```
:Else
:r5++
:End
:End
:If r5=4
:0→{266+GDB1}
:End
:Return
```

Autres détails

La label *GAME* est placé juste avant l'initialisation des variables, créant ainsi une nouvelle partie. Mais le label *LOAD* est placé après, ce qui signifie que l'on peut revenir au menu, quitter brièvement le programme et revenir dans la partie sans trop de problème.

 Je dis bien brièvement, car la calculatrice a recours très souvent aux mémoires que nous utilisons, et chaque calcul fait pourra modifier la partie. 

Pour savoir si l'on a gagné (en retrouvant le jeu d'échec), on modifie la condition pour aller à droite, en repérant si l'on est arrivé devant le jeu d'échec :

Code : Axe

```
:.If getKey(3)
:If sub(ZIP,A+U+1,B+V)→r1=0
:If A<24 and (U=5)
:A++
:ElseIf U<11
:U++
:End
:ElseIf r1=3
:ClrHome
:Output(0,0,"Enfin, le jeu d'échec a été retrouvé...
:sub(PAUSE)
:sub(FIN)
:End
```

La fonction *FIN* supprime tout simplement le jeu d'échec de la tilemap :

Code : Axe

```
:Lbl FIN
:2→{269+GDB1}
:Return
```



Cette fonction est également placée après les Datas.

Il ne faut surtout pas oublier de remettre la tilemap "en bon état" lors de l'initialisation des variables, en faisant appel à une fonction *INI*, placée après les Datas :

Code : Axe

```
:LblINI
:50→{269+GDB1}
:32→{266+GDB1}
:Return
```

Notre système de jeu en entier :

Secret (cliquez pour afficher)

Code : Axe

```
:[211E212737E1FEA0]→Pic1ZZ
:[847884E4EC877F05]
:Lbl GAME
:0→B→{L1+2}→{L1+5}→{L1+8}→{L1+11}
:14→A
:4→V+1→U
:3→{L1}→{L1+1}→{L1+3}→{L1+7}
:13→{L1+4}→{L1+10}
:26→{L1+6}→{L1+9}
:sub(INI)
:
:Lbl LOAD
:
:Repeat getKey(15)
:sub(MOV)
:DispGraphClrDraw
:sub(TLMAP)
:sub(0)
:Pt-Off(U*8,V*8,Z*8+Pic1ZZ
:End
:sub(NOKEY)
:Return
:
:Lbl MOV
:If getKey(3)
:If sub(ZIP,A+U+1,B+V)→r1=0
:If A<24 and (U=5)
:A++
:ElseIf U<11
:U++
:End
:ElseIf r1=3
:ClrHome
:Output(0,0,"Enfin, le jeu d'échec a été retrouvé...
:sub(PAUSE)
:sub(FIN)
:End
```

On mélange le tout, et voici le code menu + système de jeu :

Secret (cliquez pour afficher)

Code : Axe

```
: [Pic1]→PicPic0MENU
: [00080C0A090A0C08]→Pic0CURS
:
: sub(MENU)
: 0→θ
: Repeat θ=3 and sub(954) or getKey(15)
: θ<3 and getKey(1)-(θ>0 and getKey(4))+θ→θ
: sub(NOKEY)
: Pt-Change(26,θ*12+9,Pic0CURS
: DispGraph
: Pt-Change(26,θ*12+9,Pic0CURS
: If sub(954)
: L15,0
```


Et un petit screenshot pour les yeux:



Les ajouts possibles

Pour rendre notre jeu plus complet, j'avais proposé de mettre quelques fonctions en plus :

- Passage dans la cave via la porte au bout d'un certain nombre de zéro trouvé. (voir [biographie de Zozor](#))
 - Afficher du texte dans l'introduction et pendant le jeu (pour suivre l'imaginaire).
 - Afficher la tête de Zozor vers la gauche quand on va vers la gauche, et vers la droite quand on va vers la droite.

Ce sont des petites choses qui rendent le jeu plus crédible, et qui vous resserviront par la suite.

La sprite de Zozor dans les deux sens

Dans notre sprite actuelle, Zozor a la tête de tournée vers la droite. Pour qu'il la tourne vers la gauche, j'ai choisi de créer une autre sprite qui lui tourne la tête par la gauche :

Secret (cliquez pour afficher)

PxL	1	2	3	4	5	6	7	8
1	█						█	
2		█	█	█	█	█		
3	█						█	
4	█	█	█				█	
5	█	█	█			█	█	
6	█						█	█
7		█	█	█	█	█	█	█
8						█		█

Son code :

Code : Axe

:「847884E4EC877F05」

i

Il existe également des commandes toutes faites pour les rotations de sprite. Nous les verrons dans la troisième partie de ce cours.

On mettra donc le code de cette sprite à la suite du premier :

Code : Axe

```
: [211E212737E1FEA0]→Pic1ZZ
: [847884E4EC877F05]
```

Pour afficher à gauche ou à droite, on utilisera une variable Z qui affichera le premier Zozor si elle vaut 0, et le deuxième si elle vaut 1 :

Code : Axe

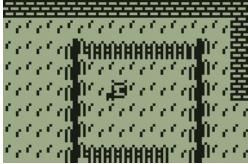
```
: Pt←Off(U*8,V*8,Z*8+Pic1ZZ)
```

Ensuite il suffit de modifier la condition de déplacement :

Code : Axe

```
:If getKey(3)
:Le code de déplacement
:0→Z
:ElseIf getKey(2)
:Le code de déplacement
:1→Z
:End
```

Vous devriez avoir ce résultat :

***Le passage de la porte...***

Avant de s'attaquer au système de niveau, on va créer 3 tiles en plus qui feront le décors de la cave :

Secret ([cliquez pour afficher](#))

Le sol de la cave :

Px	1	2	3	4	5	6	7	8
1	■							
2								
3	■							
4	■							
5	■			■				
6	■							
7	■							
8	■							

Son code :

Code : Axe

```
: [0000000008000000]
```



Cette tile était déjà présente dans la tilemap vue plus haut, mais n'était pas utilisée.

La gauche d'un ordinateur :

Px	1	2	3	4	5	6	7	8
1	■	■	■	■	■	■	■	
2	■				■			
3	■				■	■		
4	■				■		■	
5	■	■	■	■	■	■	■	
6	■		■	■				
7	■	■	■	■	■			
8	■	■	■	■	■			

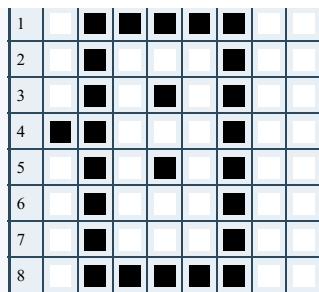
Son code :

Code : Axe

```
: [FC848685FC307878]
```

La droite d'un ordinateur :

Px	1	2	3	4	5	6	7	8



Son code :
Code : Axe

: [7C4454C45444447C]

Après, il suffit de créer une tilemap utilisant ces sprites, avec les mêmes dimensions que la précédente :

Secret (cliquez pour afficher)

Code : Axe



Vous remarquerez qu'une porte est placée en haut à gauche de la tilemap, c'est elle qui fera la liaison avec la clairière. Le jeu d'échec a été déplacé en bas à droite de la cave.

La tilemap de base doit donc être modifiée de façon à avoir une porte également, sans oublier de supprimer le jeu d'échec :

Secret (cliquez pour afficher)

Code : Axe



Mais comment faire cette transition de niveau ?

En fait, la cave et la prairie sont pointées par un seul pointeur : `GDB1`. La transition se fera donc à l'extraction de l'octet. La tilemap étant de 16 fois 36 caractères, le nombre d'octet à sauter est de $16 * 36 / 2 = 288$.

On crée une variable N qui vaut le niveau où se trouve Zozor, donc la fonction ZIP devient :

Code : Axe

```
:nib{N*16+r2*18+GDB1*2+r1}  
:Return
```



Mais comment Zozor peut-il se déplacer dans la tilemap de la cave, le sol est composé de "1", ce sont donc des obstacles pour lui ? 

Très bonne remarque ! 😊

Pour régler ce problème, il faut modifier les conditions des déplacement qui détectent si il n'y a pas de tile, et les remplacer par

Code : Axe

; If sub(ZIP,A+U,B+V+1)<2

Maintenant il faut faire la transition entre les portes, de la même manière que l'on repère le jeu d'échec :

Code : Axe

```
:ElseIf getKey(4)
:If sub(ZIP,A+U,B+V-1)→r1<2
:.Déplacement
:ElseIf r1=6
:.Changement de niveau
:End
```

Pour changer de niveau, j'utilise un inverseur : $I-N \rightarrow N$. Puis je modifie les coordonnées de Zozor sur la tilemap, pour qu'il se retrouve devant la porte du niveau suivant :

Code : Axe

```
If 1=N-N
:Les coordonnées quand il arrive dans la cave
:0=A-B
:1=V-U
:Output(0,0,"Zozor entre dans la cave...
:sub(PAUSE)
:Else
:Les coordonnées quand il arrive dans la clairière
:24=A
:8=B
:6=V
:10=U
```

Dans la détection des zéros, $\{r4+2\}=0$ se transforme en $\{r4+2\}=N$ et la suppression du mur devient :

Code : Axe

```
:If r5=4
:!If N
:.On supprime le mur dans la clairière
:0-{266+GDB1}
:Else
:.On supprime le mur dans la cave
:17-{554+GDB1}
:End
```

Le code pour la fonction *O* sera donc :

Code : Axe

```

:Lbl 0
:0->r5
:For(r1,0,3)
:{r1*3+L1-r4)-A->r2
:{r4+1)-B->r3
:If r2=U and (r3=V) and ({r4+2)=N)
:{r4+2)++
:End
:If {r4+2)=N
:If r2<96 and (r3<64
:Pt-Off(r2*8,r3*8,[003C424242423C00]
:End
:Else
:r5++
:End
:End
:If r5=4
:!If N
:0-(266+GDB1)
:Else
:17-{554+GDB1}

```

```
:End
:End
:Return
```

Il ne faut pas oublier de modifier les fonction *INI* et *FIN* pour faire réapparaître les tiles au bon endroit lors de la création d'une nouvelle partie :

Code : Axe

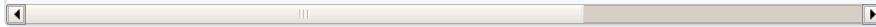
```
:Lbl FIN
:18-{557+GDB1}
:Return
:
:LblINI
:32-{266+GDB1}+1-{554+GDB1}
:50-{557+GDB1}
:Return
```

Afficher le texte

Là, il n'y avait pas de difficulté, il suffisait d'insérer du texte après l'initialisation de variables avec des *Pause* et des *ClrHome* comme il faut :

Code : Axe

```
:ClrHome
:.Les espaces en plus sont là pour équilibrer le texte sur l'écran
:Output(0,0,"Zozor se leve, quand une envie de jouer aux echecs le prend.Il s
:sub(NOKEY)
:sub(PAUSE)
:ClrHome
:Output(0,0,"mais rien y fait... il a perdu son jeu !
:sub(NOKEY)
:sub(PAUSE)
```



Le code final

Secret (cliquez pour afficher)

Code : Axe

```
:.QUETEO0
:[Pic1]→Pic0MENU
:[00080C0A090A0C08]→Pic0CURS
:
:sub(MENU)
:0→θ
:Repeat θ=3 and sub(954) or getKey(15)
:θ<3 and getKey(1)-(θ>0 and getKey(4))+θ→θ
:sub(NOKEY)
:Pt-Change(26,θ*12+9,Pic0CURS
:DispGraph
:Pt-Change(26,θ*12+9,Pic0CURS
:If sub(954)
:!If θ
:sub(GAME)
:ElseIf θ=1
:sub(RPD)
:ElseIf θ=2
:sub(NOKEY)
:ClrHome
:Output(0,0,"Janvier 2011 Pour le SdZ"
:sub(PPAUSE)
:End
:sub(MENU)
:End
:End
:Return
:
:Lbl PAUSE
:Repeat sub(954):End
:Return
:
:Lbl 954
:getKey(9) or getKey(54)
:Return
:
:Lbl MENU
:DispGraph(Pic0MENU)
:StoreGDB
:Return
:
:Lbl NOKEY
:While getKey(0)
:Pause 5
:End
:Return
:
:[211E212737E1FEA0]→Pic1ZZ
:[847884E4EC877F05]
:
:Lbl GAME
:0→B→{L1+2}→{L1+5}→{L1+8}→{L1+11}→N
:14→A
:4→V+1→U
:3→{L1}→{L1+1}→{L1+3}→{L1+7}
:13→{L1+4}→{L1+10}
:26→{L1+6}→{L1+9}
:sub(INI)
:ClrHome
:Output(0,0,"Zozor se leve, quand une envie de jouer aux echecs le prend
:sub(NOKEY)
```


Pour ceux qui sont intéressés, voici le code final compatible TI-Editor :
Secret (cliquez pour afficher)

Code : Axe

```

: .QUETEO
:[Pic1]→Pic0MENU
:[00080C0A090A0C08]→Pic0CURS
:
:sub(MENU)
:0→θ
:Repeat θ=3 and sub(954) or getKey(15)
:θ<3 and getKey(1)-(θ>0 and getKey(4))+θ→θ
:sub(NOLEY)
:Pt-Change(26,θ*12+9,Pic0CURS
:DispGraph
:Pt-Change(26,θ*12+9,Pic0CURS
:If sub(954)
:  !If θ
:sub(GAME)
:ElseIf θ=1
:sub(RPD)
:ElseIf θ=2
:sub(NOLEY)
:ClrHome
:Output(0,0,"Janvier 2011      Pour le SdZ"
:sub(PAUSE)
:End
:sub(MENU)
:End
:End
:Return
:
:Lbl PAUSE
:Repeat sub(954):End
:Return
:
:Lbl 954
:getKey(9) or getKey(54)
:Return
:
:Lbl MENU
:DispGraph(Pic0MENU)
:StoreGDB
:Return
:
:[211E212737E1FEA0]→Pic1ZZ
:[847884E4EC877F05]
:Lbl GAME
:0→B→{L<sub>1</sub>+2}→{L<sub>1</sub>+5}→{L<sub>1</sub>+8}→{L<sub>1</sub>+11}
:14→A
:4→V+1→U
:3→{L<sub>1</sub>}→{L<sub>1</sub>+1}→{L<sub>1</sub>+3}→{L<sub>1</sub>+7}
:13→{L<sub>1</sub>+1}→{L<sub>1</sub>+4}→{L<sub>1</sub>+10}
:26→{L<sub>1</sub>+6}→{L<sub>1</sub>+9}

```


Vous pouvez télécharger les fichiers du jeu et le source : [ici](#).

Et voici un screenshot du résultat :



Bon, pas besoin de QCM, je pense que vous avez assez travaillé sur ce TP comme ça 😊. N'hésitez pas à améliorer votre jeu, ou à le reprendre comme base pour d'autres jeux !

J'imagine que maintenant vous avez des tas d'idées de jeux mais vous vous retrouverez vite confronté aux limites de l'Axe Parser.



Pourquoi ne pas passer à la partie suivante alors ? 

Partie 3 : Repoussons les limites !

Dans cette troisième partie, vous allez mettre en pratique vos connaissances, en apprenant à faire des programmes plus esthétiques grâce aux niveaux de gris, des jeux qui sauvegardent les scores avec les *appvars*, des fonctions plus rapides grâce aux techniques d'optimisation...et bien plus encore ! 😊

Si ce chapitre est plus facile que les deux précédent, cela ne signifie pas qu'il faut foncer tête baissée ! N'hésitez pas à relire attentivement les parties 2 et 3 pour être sûr d'avoir acquis les bases nécessaires. Dès que vous vous sentez près, lancez-vous !



Les caractères ASCII et les tokens

Vous avez sûrement remarqué que la calculatrice affichait beaucoup de caractères spéciaux (►, #, @, π,...). Dans ce chapitre nous étudierons les caractères ASCII ainsi qu'une partie des tokens, puis nous apprendrons à utiliser la commande *input*.

Les caractères ASCII

Il faut savoir une chose importante : comme dans tout ordinateur, votre calculatrice ne peut retenir que des nombres.

Mais comment le texte peut-être retenu dans la mémoire ? 😊

C'est là qu'intervient les caractères ASCII, très utilisés dans beaucoup de langage de programmation, ce sont de simples nombres qui sont affichés sous forme de caractères (plus précisément, un octet = un caractère), à l'aide de la table ASCII.

Comment cela est possible ?

En fait, pour afficher une chaîne de caractères, on l'écrit, on la pointe, puis on demande à la calculatrice de l'afficher : ça c'est ce qu'on fait dans notre programme.

De son côté, la calculatrice récupère la chaîne de caractères, convertit le premier octet en sprite associé au numéro, l'affiche à l'emplacement indiqué, puis fait de même pour tous les autres caractères (à la suite du premier bien sûr).

Pour information, ces sprites font 5 pixels de largeur pour 7 de hauteur, en général, mais je ne m'aventurerai pas plus loin, la suite relève de l'Asmz80. 😊

Pour revenir à nos caractères, voici tout ceux présents sur nos calculatrices :

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ñ	uvvw	†	*	J	X	+	-	·	π	3	F			
1	J	-12	z	o	r	T	≤	≥	-	E	→	10	↑	↓	
2	!	"	#	%	&	'	()	*	,	-	/			
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N
5	P	Q	R	S	T	U	V	W	X	Y	Z	Ø	×	1	^
6	~	a	b	c	d	e	f	g	h	i	j	k	l	m	n
7	P	q	r	s	t	u	v	w	x	y	z	{	}	~	█
8	0	1	2	3	4	5	6	7	8	9	À	Á	Ã	à	á
9	À	Á	Ã	à	á	É	É	É	É	É	Ê	Ê	Ê	Ê	í
A	Í	Ó	Ó	Ó	Ó	Ó	Ó	Ó	Ó	Ó	Ó	Ó	Ó	Ó	Ó
B	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù
C	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù
D	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù
E	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù
F	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù	Ù

Mais comment peut-on les faire apparaître ? 😊

Repérez votre caractère, puis regardez le nombre hexadécimal correspondant sur **la ligne**, puis sur **la colonne**, et mettez les à la suite.

Par exemple le numéro "6" ce sera le nombre 36 hexadécimal, donc 54 en décimal :

Code : Axe

```
:Disp 54►Char
```

A quoi sert le ►Char après 54 ?

Cela permet d'afficher un seul caractère ASCII, on le trouve ici :



Rappelez vous : si vous ne le mettez pas, toute la mémoire de la calculatrice va s'afficher sous forme de chaîne de caractères ! Pourquoi ? Encore la **faute du zéro** qui manque pour arrêter la lecture. 😊

Cependant, on peut utiliser assez habilement les Data pour afficher une chaîne de caractères avec tous les caractères du tableau : en mettant les coordonnées (ligne; colonne) du caractère entre crochet.

Mettions que je veuille afficher "@Noë", je vois que "@" est à l'emplacement [40], "N" est à [4E], "o" à [6F], "ë" à [99] et "ë" à [F1].

Notre code est donc :

Code : Axe

```
Disp [404E6F99F100]
```

En image :



Encore une fois, n'oubliez pas le zéro à la fin des Datas !

Les tokens

Vous avez sûrement remarqué que les commandes qu'on utilise en Axe, ou même n'importe quelle autre commande de la calculatrice, sautent plusieurs lettres d'un coup, comme si ce n'était qu'un caractère :

```
PROGRAM:AAA
::A
::BisP
::Char
::For<
```

En fait chaque commande, ou même les lettres et les numéros, sont ce qu'on appelle un token.



Token ? Quésako ?

Un caractère token est un ensemble d'un ou plusieurs caractères ASCII qui n'est en fait... qu'un simple nombre ! 😊

De cette manière, la calculatrice n'a juste qu'à regarder le nombre qu'on lui envoi, puis elle le convertit en chaîne de caractères, et enfin elle l'affiche avec la méthode vu plus haut. Mais encore une fois, je ne m'aventurai pas plus loin dans l'explication. 😊

Les tokens d'un octet

Tout comme pour les caractères ASCII, il y a des tableaux de tokens. Voici celui pour les tokens d'un octet :

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	Aucun	►DMS	►Dec	►Frac	→	Boxplot	[]	{	}	r	o	-1	2	T	3
1	()	round(pxl-Test(augment(rowSwap(row+(*row(*row+(max(min(R►Pt(R►Pθ(P►Rx(P►Ry	median(
2	randM(mean(solve(seq(fInt(nDeriv(Aucun	fMin(fMax((espace)	"	,	i	!	CubicReg	QuartReg
3	0	1	2	3	4	5	6	7	8	9	.	E	or	xor	:	(nouvelle ligne)
4	and	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	θ	token de deux octets	token de deux octets	token de deux octets	prgm
6	token de deux octets	Radian	Degree	Normal	Sci	Eng	Float	=	<	>	≤	≥	≠			
7	+	-	Ans	Fix	Horiz	Full	Func	Param	Polar	Seq	IndpntAuto	IndpntAsk	DependAuto	DependAsk	token de deux octets	o
8	+	.	*	/	Trace	ClrDraw	ZStandard	ZTrig	ZBox	Zoom In	Zoom Out	ZSquare	ZInteger	ZPrevious	ZDecimal	ZoomStat
9	ZoomRcl	PrintScreen	ZoomSto	Text(nPr	nCr	FnOn	FnOff	StorePic	RecallPic	StoreGDB	RecallGDB	Line(Vertical	Pt-On(Pt-Off(
A	Pt-Change(Pxl-On(Pxl-Off(Pxl-Change(Shade(Circle(Horizontal	Tangent(DrawInv	DrawF	token de deux octets	rand	π	getKey	'	?
B	- (négatif)	int(abs(det(identity(dim(sum(prod(not(iPart(fPart(token de deux octets	√(³√(ln(e^(
C	log(10^(sin(sin ⁻¹ (cos(cos ⁻¹ (tan	tan ⁻¹ (sinh(sinh ⁻¹ (cosh(cosh ⁻¹ (tanh(tanh ⁻¹ (If	Then
D	Else	While	Repeat	For(End	Return	Lbl	Goto	Pause	Stop	IS>(DS<(Input	Prompt	Disp	DispGraph
E	Output(ClrHome	Fill(SortA(SortD(DispTable	Menu(Send(Get(PlotsOn	PlotsOff	L	Plot1(Plot2(Plot3(token de deux octets (spécifiques à la 84+)
F	^	×\	I-Var Stats	2-Var Stats	LinReg(a+bx)	ExpReg	LnReg	PwrReg	Med-Med	QuadReg	ClrList	ClrTable	Histogram	xyLine	Scatter	LinReg(ax+b)



Comment afficher un caractère token ? 🎓

Il existe pour cela la commande ►Tok qui s'utilise comme pour les caractères ASCII :

Code : Axe

```
: [A6]→GDB1
: Disp {GDB1}►Tok
```



La commande ►Tok se trouve ici :



Les tokens de deux octets

Vous l'aurez deviné, il existe plus de token que cela. La méthode : le nombre est réparti sur 2 octets. La liste des tokens à 2 octets est tellement longue que je ne peux pas tout réécrire. Cependant, pour les intéressés, je conseil d'aller jeter un coup d'œil sur ce site (en anglais).

La méthode pour afficher un token de 2 octets est soit d'utiliser une variable normal, soit de ne pas oublier le petit r :

Code : Axe

```
: [6109]--GDB1
: Disp {GDB1}▶Tok
```

Petit exercice

Afin d'assimiler tout ça, je vous propose de créer un petit utilitaire qui affichera en même temps le nombre, son caractère ASCII et son caractère token. Pour aller plus rapidement, on pourra soit augmenter/diminuer de 1, soit de 10, soit de 100.

Résultat

[Secret \(cliquez pour afficher\)](#)

Code : Axe

```
: .TOKASCII
: ClrHome
: 0→A
: Repeat getKey(15)
: getKey(10)-getKey(11)*10+getKey(4)-getKey(1)*10+getKey(3)-
: getKey(2)+A-A
: If getKey(0)
: ClrHome
: Output(0,0,"Le nombre :
: Output(0,1,A▶Dec
: Output(0,2,"Le caractère :
: Output(0,3,A▶Char
: Output(0,4,"Le token :
: Output(0,5,A▶Tok
: End
: End
```

Screenshot :

```
Le nombre:
0
Le caractère:
■
Le token:
?
```



Vers le numéro 65535, certains tokens prennent plusieurs pages.

La commande input

Je vous recommande fortement d'essayer de créer votre propre fonction `input`, cependant il faut savoir qu'il existe une fonction toute faite en Axe, exactement comme celle utilisée en TI-Basic. Elle renvoie une chaîne de tokens, qui peut ensuite être pointée.



Comment peut on pointer des valeurs qu'on a pas encore créé ?

Il y a une chose que je ne vous ai pas encore présenté, ce sont les pointeurs... non-statiques. Vous les connaissez déjà, ce sont vos variables : rappelez-vous, un pointeur est un simple nombre, et une variable peut stocker un nombre, donc on peut pointer nos Datas avec des simples variables.

Ce sera utile par la suite pour pointer des variables externes au programme, mais ici cela nous sert à récupérer la chaîne de tokens (qui est elle-même placée par défaut à un endroit précis de la mémoire).

Passons au code, vous connaissez déjà tout, sauf qu'on remplace les Datas par la fonction `input`, et le pointeur par une variable :

Code : Axe

```
: input→A
```



La commande `input` se trouve ici :



A pointe donc une chaîne de Token, il ne nous reste plus qu'à l'afficher.



Comment afficher une chaîne de token ?

On peut utiliser une boucle `For` après avoir calculé la longueur de la chaîne pour afficher un par un les caractères :

Code : Axe

```
: .A
: input→A
: length(A)→B
: For(C,0,B)
: Output(C,1,{A+C})▶Tok
: End
```

Mais cette solution comporte de nombreux bugs à l'affichage. L'autre possibilité est de remarquer que les lettres de l'alphabet et les numéros ont les mêmes valeurs en tokens et en ASCII. Il suffit donc d'afficher comme pour une chaîne de caractère normale :

Code : Axe

```
: .A
: input→A
: Output(0,1,A)
```



Là encore il y a des risques d'erreurs d'affichage, car les autres caractères ne correspondent pas sur les deux tableaux.

Ce chapitre est assez facile à comprendre mais il est important à connaître. Grâce à ces connaissances vous pourrez plus tard manipuler les programmes et plein d'autres variables de la calculatrice insoupçonnées.

Mais pour le moment, direction le chapitre suivant. 

Les niveaux de gris

Si vous trouvez les graphismes de vos programmes un peu monotone avec seulement le noir et le blanc, ne partez pas ! Dans ce chapitre vous apprendrez le principe des niveaux de gris et comment l'appliquer dans vos programmes.

3 niveaux de gris

Tout d'abord il faut savoir qu'il existe deux commandes pour les niveaux de gris : celle qui permet d'afficher 3 niveaux de gris, et celle qui permet d'en afficher 4.

Le principe

La méthode consiste à alterner très rapidement les pixels pour faire apparaître à nos yeux une nouvelle nuance de couleur. Pour cela il faut au préalable avoir dessiné les écrans à alterner. Vous l'aurez compris en Axe on utilisera le **buffer** et... le **back-buffer**.



Pour 3 niveaux de gris, il faut compter le blanc, le gris et le noir (tous ne sont que des dégradés du gris).

La pratique

En pratique il faut afficher sur le buffer tout ce qui est en **noir**, et afficher sur le back-buffer ce qui va être en **gris** : le reste sera forcément blanc. Prenons l'exemple de ce sprite :

Pxl	1	2	3	4	5	6	7	8
1	■	■	■	■	■	■	■	■
2	■	■	■	■	■	■	■	■
3	■	■	■	■	■	■	■	■
4	■	■	■	■	■	■	■	■
5	■	■	■	■	■	■	■	■
6	■	■	■	■	■	■	■	■
7	■	■	■	■	■	■	■	■
8	■	■	■	■	■	■	■	■

Maintenant, essayez de convertir ce sprite en suivant les règles dites plus haut, puis de l'afficher en 3 niveaux de gris.



La commande pour afficher 3 niveaux de gris est **DispGraph^r** à utiliser dans une boucle.

Rappel, *DispGraph* se trouve ici :



Et le **r** se trouve ici :



Secret (cliquez pour afficher)

D'abord il faut bien séparer les deux écrans, en premier le buffer :

Pxl	1	2	3	4	5	6	7	8
1	■	■	■	■	■	■	■	■
2	■	■	■	■	■	■	■	■
3	■	■	■	■	■	■	■	■
4	■	■	■	■	■	■	■	■
5	■	■	■	■	■	■	■	■
6	■	■	■	■	■	■	■	■
7	■	■	■	■	■	■	■	■
8	■	■	■	■	■	■	■	■

Le sprite du buffer a pour code :

Code : Axe

: [3C42A581A599423C]

Puis le back-buffer :

Pxl	1	2	3	4	5	6	7	8
1	■	■	■	■	■	■	■	■
2	■	■	■	■	■	■	■	■
3	■	■	■	■	■	■	■	■
4	■	■	■	■	■	■	■	■
5	■	■	■	■	■	■	■	■
6	■	■	■	■	■	■	■	■
7	■	■	■	■	■	■	■	■
8	■	■	■	■	■	■	■	■

Le code est ici :

Code : Axe

: [003C5A7E5A663C00]

Puis on pointe le tout et on affiche avec les commandes déjà vu pour les différents écran :

Code : Axe

```
: [3C42A581A599423C]--Pic0B
: [003C5A7E5A663C00]--Pic1BB
:ClrDrawr
:Pt-On(0,0,Pic0B)
:Pt-On(0,0,Pic1BB)r
:Repeat getKey(15)
:DispGraphr
:End
```

Le résultat :



Exercice

Quelques sprites pour vous entraîner :

Pxl	1	2	3	4	5	6	7	8
1	■			■	■			
2	■		■	■	■	■		
3		■	■	■			■	
4		■	■	■	■	■	■	
5		■	■	■	■	■	■	
6		■	■	■	■	■	■	
7			■	■	■	■	■	
8				■	■			

Pxl	1	2	3	4	5	6	7	8
1	■		■	■	■	■		
2	■		■	■	■	■	■	
3			■	■	■	■		
4		■		■	■			
5		■	■	■	■	■	■	
6			■	■				
7			■			■		
8	■	■			■	■	■	

Pxl	1	2	3	4	5	6	7	8
1		■					■	
2	■		■	■	■	■	■	
3		■	■	■	■	■	■	
4		■	■	■	■	■	■	
5		■					■	
6			■			■		
7			■	■	■		■	
8							■	



Je ne donnerai pas la solution cette fois, testez donc sur votre calculatrice pour voir si vous avez bon. 😊

4 niveaux de gris

Pour 4 niveaux de gris, il faut compter le blanc, le gris 33%, le gris 66% et le noir. Je ne connais pas en détail le temps d'affichage de chaque écran, mais on utilise toujours le buffer et le back-buffer.

Les sprites sont un peu plus difficiles à convertir en hexadécimal, car il faut mettre **le gris 33% dans le back-buffer**, **le gris 66% dans le buffer**, le tout sans oublier **le noir dans les deux** (le reste sera forcément blanc).

Essayons avec ce sprite :

Pxl	1	2	3	4	5	6	7	8
1	■							
2	■							

3									
4			■	■	■				
5	■		■	■	■				■
6	■	■	■	■	■	■	■	■	
7				■	■	■	■	■	
8					■	■	■		

Comme précédemment on sépare le buffer et le back-buffer avant de convertir en hexadécimal.
Le buffer :

Pxl	1	2	3	4	5	6	7	8
1								
2								
3								
4			■	■				
5	■		■	■				
6	■	■	■	■	■	■	■	
7				■	■	■	■	
8					■	■	■	

Le code est donc :

Code : Axe

: [00000060B0EE1EOA]

Pour le back-buffer :

Pxl	1	2	3	4	5	6	7	8
1								
2								
3			■	■	■			
4				■				
5			■					■
6			■					
7					■	■	■	
8					■	■	■	

Ici le code est :

Code : Axe

: [000028102110000A]

La commande pour faire apparaître 4 niveaux de gris est *DispGraphrr* (pareil que *DispGraphr* mais avec deux *r*), sans oublier de placer la commande dans une boucle.

Le code est donc :

Code : Axe

```
: [00000060B0EE1EOA]--Pic0B
:[000028102110000A]--Pic0BB
:ClrDrawrr
:Pt-On(0,0,Pic0B)
:Pt-On(0,0,Pic1BB)r
:Repeat getKey(15)
:DispGraphrr
:End
```

En image :



Le tilemapping avec grayscales

Vous en avez déjà fait en noir et blanc. Avec des niveaux de gris, ça n'est pas vraiment plus compliqué.

Création de la map

Commençons par dessiner les tiles de la map. Comme dans le chapitre sur les datas, nous allons dessiner de l'herbe et une

maison, mais cette fois-ci en quatre niveaux de gris !

L'herbe :

Pxl	1	2	3	4	5	6	7	8
1	Light Gray							
2	Light Gray					Medium Gray		Medium Gray
3	Light Gray					Medium Gray	Dark Gray	
4	Light Gray					Medium Gray		
5	Light Gray					Medium Gray		
6	Medium Gray		Medium Gray			Medium Gray		
7	Light Gray	Medium Gray						
8	Light Gray							

Le code hexadécimal est donc :

[Secret \(cliquez pour afficher\)](#)

[0000000000000000]
[0005020000A04000]

Maison haut gauche :

Pxl	1	2	3	4	5	6	7	8
1	Light Gray							Black
2	Light Gray					Black	Black	Black
3	Light Gray			Black	Black	Medium Gray	Medium Gray	Black
4	Light Gray	Black	Black	Medium Gray	Medium Gray	Medium Gray	Medium Gray	Black
5	Black	Medium Gray	Black					
6	Black	Medium Gray	Black					
7	Black	Medium Gray	Black					
8	Black	Medium Gray	Black					

Le code hexadécimal est donc :

[Secret \(cliquez pour afficher\)](#)

[01071F7FFFFFFF]
[01071961818181]

Maison haut droite :

Pxl	1	2	3	4	5	6	7	8
1	Black							
2	Black	Black	Black	Light Gray				
3	Black	Medium Gray	Medium Gray	Black	Black			
4	Black	Medium Gray	Medium Gray	Medium Gray	Black	Black	Light Gray	
5	Black	Medium Gray	Black					
6	Black	Medium Gray	Black					
7	Black	Medium Gray	Black					
8	Black	Medium Gray	Black					

Le code hexadécimal est donc :

[Secret \(cliquez pour afficher\)](#)

[80EOF8FEFFFFFF]
[80E09886818181]

Maison bas gauche :

Pxl	1	2	3	4	5	6	7	8
1	Black	Medium Gray	Black					
2	Black	Medium Gray	Medium Gray	Medium Gray	Medium Gray	Black	Black	Medium Gray
3	Black	Medium Gray	Medium Gray	Black	Black	Light Gray	Light Gray	Light Gray
4	Black	Black	Black	Light Gray	Light Gray	Light Gray	Light Gray	Light Gray
5	Black	Light Gray	Light Gray	Black	Black	Black	Black	Light Gray
6	Black	Light Gray	Light Gray	Black			Black	Light Gray
7	Black	Light Gray	Light Gray	Black		Medium Gray	Black	Light Gray
8	Black	Light Gray	Light Gray	Black			Black	Light Gray

Le code hexadécimal est donc :

[Secret \(cliquez pour afficher\)](#)

[FFFEF8E09E929692]
[81879FFFFF3F3F3]

Maison bas droite :

Px	1	2	3	4	5	6	7	8
1	■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■
2	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■
3	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■
4	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■
5	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■
6	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■
7	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■
8	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■	■■■■■■■■

Le code hexadécimal est donc :

[Secret \(cliquez pour afficher\)](#)

[FF7F1F0771517101]
[81E1F9FFFFDFFFFF]

On obtient alors :

Code : Axe

```
: [0000000000000000]--Pic1
: [0005020000A04000]
: .herbe
:
: [01071F7FFFFFFFFF]
: [0107196181818181]
: .maison haut gauche
:
: [80E0F8FEFFFFFF]
: [80E09886818181]
: .maison haut droite
:
: [FFFEF8E09E929692]
: [81879FFFFF3F3F3]
: .maison bas gauche
:
: [FF7F1F0771517101]
: [81E1F9FFFFDFFFFF]
: .maison bas droite
```



Désormais, chaque tile prend 16 octets car il y a deux fois 8 octets (pour une tile classique).

Pour afficher les tiles en 4 niveaux de gris, on a juste à utiliser les commandes vues plus haut :

Code : Axe

```
:Pt-On(X,Y,Pic1+0
:Pt-On(X,Y,Pic1+8)r
:.Herbe
:Pt-On(X,Y,Pic1+16
:Pt-On(X,Y,Pic1+24)r
:.Maison haut/gauche
:Pt-On(X,Y,Pic1+32
:Pt-On(X,Y,Pic1+40)r
:.Maison haut/droite
:Pt-On(X,Y,Pic1+48
:Pt-On(X,Y,Pic1+56)r
:.Maison bas/gauche
:Pt-On(X,Y,Pic1+64
:Pt-On(X,Y,Pic1+72)r
:.Maison bas/droite
```

Ainsi, toutes les tiles sur le buffer sont multiples de 16 et toutes les tiles sur le back-buffer sont multiples de 16 additionné à 8.

Par simplification, on a :

Code : Axe

```
:.La variable A vaut la valeur de la tile correspondante
:Pt-On(X,Y,A*16+Pic1
:Pt-On(X,Y,A*16+8+Pic1)r
```

Que l'on peut optimiser par :

Code : Axe

```
:.La variable A vaut la valeur de la tile correspondante
:Pt-On(X,Y,A*16+Pic1-0
:Pt-On(X,Y,0+8)r
```

puisque la tile à afficher sur le back-buffer se trouve à une distance de 8 octets de celle à afficher sur le buffer.

Reprendons la map qui nous avait servi dans le chapitre sur les datas :

Code : Axe

```
: [000000000000]--GDB1
: [000000000000]
: [000000000000]
: [000001200000]
: [000003400000]
: [000000000000]
: [000000000000]
: [000000000000]
```

Bonne nouvelle, elle marchera parfaitement pour notre map en 4 niveaux de gris. 😊

Affichage de la tilemap

Pour l'affichage de la map à l'écran, nous pouvons employer la même méthode vue pour les tiles monochromes :

Code : Axe

```
:For(Y,0,7)
:For(X,0,11)
:nib{Y*6+GDB1*2+X}→A
:Pt-On(X*8,Y*8,A*16+Pic1)
:End
:End
```

Sauf qu'il faut aussi afficher les tiles sur le back-buffer donc on doit écrire :

Code : Axe

```
:For(Y,0,7)
:For(X,0,11)
:nib{Y*6+GDB1*2+X}→A
:Pt-On(X*8,Y*8,A*16+Pic1-0)
:Pt-On(X*8,Y*8,0+8)r
:End
:End
```

Voici maintenant le code final pour afficher notre magnifique map 😊 :

Code : Axe

```
:.TILEMAP
:[0000000000000000]→Pic1
:[0005020000A04000]
:.herbe
:[01071F7FFFFFFFFF]
:[01071961818181]
:.maison haut gauche
:[80E0F8FEFFFFFFFFFF]
:[80E09886818181]
:.maison haut droite
:[FFFEF8E09E929692]
:[81879FFFFFF3F3F3]
:.maison bas gauche
:[FF7F1F0771517101]
:[81E1F9FFFDDFFF]
:.maison bas droite
:
:[000000000000]--GDB1
:[000000000000]
:[000000000000]
:[000001200000]
:[000003400000]
:[000000000000]
:[000000000000]
:
:ClrDrawrr
:.On oublie pas d'effacer le buffer et le back-buffer
:
:For(Y,0,7)
:For(X,0,11)
:nib{Y*6+GDB1*2+X}→A
:Pt-On(X*8,Y*8,A*16+Pic1-0)
:Pt-On(X*8,Y*8,0+8)r
:End
:End
:
:Repeat getKey(15)
:DispGraphrr
:End
```

Voici le résultat en image :



N'est ce pas magnifique ? 😊

Utiliser 4 niveaux de gris dans un jeu (par exemple un RPG) peut parfois ralentir le programme et l'écran devient alors clignotant. Heureusement pour y remédier il existe la commande *Full* permettant d'accélérer la vitesse d'exécution du programme, et miracle,

l'affichage sera de nouveau fluide ! .

 Malheureusement cela ne marche pas sur la TI 83+ dont le processeur est bridé à 6MHz (faute à TI) et ne peut donc pas utiliser le mode *Full*. .

On peut aussi tester si la calculatrice supporte le *full speed mode* fonctionne de cette manière :

Code : Axe

```
:!If Full  
: .Non supporté  
:End  
: .Activé
```

Maintenant toutes les commandes vues dans la deuxième partie pour manipuler les buffers prennent tout leur sens. Dorénavant vous êtes capable de faire des jeux avec des beaux graphismes. .

Optimiser son code

Votre calculatrice n'a que très peu de mémoire, et un programme peut très rapidement prendre des tailles faramineuses. Cette partie est donc très importante : vous apprendrez à optimiser et améliorer vos programmes au maximum. 😊

Généralités

Quelques recommandations

Je commencerai par un avertissement, par une phrase éminemment célèbre chez les programmeurs de tous genres :

Early optimization is the root of all evil.

Ou, en bon français :

Optimiser trop tôt est source de tous les maux.

Je suis assez fier de cette phrase. 🎉

Donc en résumé, programmez d'abord, et optimisez ensuite ce qui a besoin de l'être. Ce propos doit toutefois être nuancé dans le cas des calculatrices : en effet, la RAM étant excessivement limitée, optimiser en codant en est très souvent indispensable. Imaginez si vous aviez à simplifier un très gros programme très mal codé... sur la calculatrice (cela peut prendre des jours croyez-moi). 🤪

Répartir le code source

Afin de clarifier grandement votre code s'il commence à prendre des proportions apeurantes, il est possible de le séparer en plusieurs "sous-programmes". Il suffit de les inclure dans votre programme "principal" comme ceci :

Code : Axe

```
:prgmPROG
```

C'est tout. 😊



Un **programme source** importé ainsi est appelé une **bibliothèque**. Pour que le compilateur le reconnaisse ainsi, il doit avoir la première ligne typique d'un source Axe (aussi appelé header), à la différence que le nom du programme exécutable ne sera jamais compilé :

Code : Axe

```
:.LIB  
:...Votre code
```

Mais ça ne sert à rien niveau taille ? C'est juste pour rendre le fichier principal plus lisible ?

Baaaaah ... oui. Niveau taille, ça ne change rien que vous fassiez :

Code : Axe - prgmMAIN

```
:.EXE  
:Sub(PLP)  
:Return  
:  
:Lbl PLP  
:Disp "PLOP"  
:Return
```

Ou bien :

Code : Axe - prgmMAIN

```
:.EXE  
:Sub(PLP)  
:Return  
:  
:prgmLIB
```

Code : Axe - prgmLIB

```
:.LIB  
:Lbl PLP  
:Disp "PLOP"  
:Return
```

Vous l'aurez compris : on peut ainsi développer nos propres fonctions dans des bibliothèques et les réutiliser dans plusieurs programmes différents. C'est juste une meilleure organisation ! 🎉

Bien entendu, vous pouvez inclure autant de bibliothèques que vous voulez ! Attention, il est toutefois impossible (pas encore possible en tout cas) de faire des bibliothèques "imbriquées", c'est à dire qui incluent elles-mêmes d'autres bibliothèques.

Cette technique n'optimisera donc en rien votre programme, mais vous fera gagner beaucoup de temps.

Pré-calculer des grosses valeurs

Bien, passons désormais à quelques techniques qui pourraient s'appliquer à n'importe quel langage de programmation.

Commençons par le *pré-calcul* de valeurs. C'est très simple : si vous utilisez à plusieurs endroit un même calcul, il est très

souvent avantageux de stocker d'abord le résultat puis de l'utiliser, au lieu de faire le même calcul à chaque fois. Exemple :

Code : Axe - Gâchis de mémoire

```
For (A, 0, 9)
Px1-On (Z/2*Y+B+A, Z/2*Y+C+A)
End
```

Ici l'expression $Z/2*Y$ va être calculée 2 fois dans la commande *Px1-On*, multiplié par 10 tours de boucle cela nous donne 20 calculs inutiles que l'on peut éliminer facilement :

Code : Axe - Utilisation d'une variable temporaire

```
Z*Y/2-r1
For (A, 0, 9)
Px1-On (r1+B+A, r1+C+A)
End
```

Résultat : un gain de temps, et quand l'expression est longue, de place ! 😎

De même, certaines valeurs peuvent être assez longues à calculer, comme une racine carrée ou un cosinus.

Code : Axe - Cosinus de racines carrées

```
For (A, 0, 49)
{L1+A}+cos (√(A))→{L1+A}
End
```

Il est parfois intéressant de les calculer d'avance et de les stocker dans une liste.

Code : Axe - Cosinus de racines carrées

```
.Début du programme
For (A, 0, 49)
cos (√(A))→{L2+A}
End

.Code...

For (A, 0, 49)
{L1+A}+{L2+A}→{L1+A}
End
```

Cela ne vous rajoute pas tellement d'octets, mais la deuxième boucle est bien plus rapide désormais. 🎉

Utiliser le moins de commandes différentes possible

Voici un des moyens les plus efficaces pour réduire drastiquement la taille de vos programmes : utiliser le strict minimum de routines toutes faites.

Observons une fonction qui crée des carrés vides de trois pixels de large à un endroit blanc de l'écran :

Code : Axe - PRGM1

```
:Lbl Box
:ClrDraw
:Rect (X, Y, 10, 10)
:RectI (X, Y, 4, 4)
```

En fait ce code est exactement pareil si on utilise la commande *RectI()* deux fois :

Code : Axe - PRGM2

```
:Lbl Box
:ClrDraw
:RectI (X, Y, 10, 10)
:RectI (X, Y, 4, 4)
```

Vous pouvez tester ces deux programmes : font la même chose. Par contre, si vous n'utilisez pas d'autre *Rect()* ailleurs dans le programme, le programme compilé du deuxième code fera 114 octets de moins que celui du premier !



Comment cela se fait-il ? 🌟

En fait, les fonctions que vous utilisez dans votre programme comme *Pt-On()*, *Rect()*, et que sais-je encore, n'existent pas directement en Assembleur : ce sont des fonctions que le compilateur va devoir rajouter au programme. Donc plus vous utilisez de fonctions différentes, plus le programme va s'alourdir, et il est souvent avantageux de recycler une même commande : on peut gagner des centaines voir des milliers d'octets une fois le programme compilé. 😎

Booster les tests

Vous n'êtes pas sans savoir que le code :

Code : Axe

```
:If A=0
```

Se simplifie par :
Code : Axe

```
: !If A
```

Mais savez vous que l'on peut faire encore mieux avec une simple soustraction, ainsi le code :

Code : Axe

```
: If A=B
```

Devient :
Code : Axe

```
: !If A-B
```

Cela marche tout aussi bien, et vous économisez **7 octets d'un coup** 😊

Encore mieux maintenant, vous connaissez l'opérateur logique *and* :

Code : Axe

```
If A=10 and (B=D)
```

On peut le remplacer de la même manière par l'opérateur + qui reviens à un *or* mais de deux octets (On le trouve ici :

2nde
2ND



Code : Axe

```
!If A=10 + (B=D)
```

Ce code économise encore 7 octets, incroyable non ? 😎

Utiliser les puissances de deux

Comme vous le savez, la calculatrice ne manipule que du binaire, et elle sait très bien le faire. C'est à dire que **compter en puissance de 2 est beaucoup plus rapide et léger** dans un programme !

Rappelez-vous, les puissances de deux qui vont être utiles pour manipuler des variables de deux octets :

1	1	1	1	1	1	1	1	-1	1	1	1	1	1	1	1
32768	16384	8192	4096	2048	1024	512	256	-128	64	32	16	8	4	2	1

Privilégier ces valeurs dans vos **calculs réduit et accélère** grandement votre programme. En rouge ce sont les valeurs qui permettent les meilleures optimisations et en orange celles qui optimisent un peu moins, mais toujours plus que le reste.

Prenons l'exemple de ce code : Pause A*30. Si vous avez pris 30 parce que l'effet recherché correspondait à peu près à ce code, essayez alors Pause A*32. Et là miracle, **19 octets d'économisés** ! 😊

Ceci est un exemple très simple, mais sachez que l'on peut également optimiser certains tests comme :

Code : Axe

```
:If A≥2
:End
:If A<2
:End
```

Par une simple division on peut optimiser :

Code : Axe

```
:If A/2
:End
:!If A/2
:End
```

Ainsi 3 octets seront économisés pour chaque condition. 😊

⚠ Je rappelle que ceci ne marche que pour les puissances de deux !

Le registre HL et ses usages

Parlons un peu de calcul, tout d'abord qu'est ce qu'un calcul pour la calculatrice ?

Je vous donne la réponse : en fait, c'est n'importe quelle expression qu'on pourrait stocker dans une variable.

Secret (cliquez pour afficher)

Chaque ligne contient un calcul :

Code : Axe

```
:A+4
:F-- 
:14*2
:B<0
:B and C
:0
:getKey
:getCalc("prgmPLOP")
:port
```

Et je pourrais continuer longtemps, quasiment toutes les commandes de l'Axe sont des calculs !

Maintenant, si je vous dis qu'avant de l'envoyer dans une variable, chaque calcul passe dans une zone temporaire. Par exemple dans A+4, la zone vaut d'abord A, puis vient le +4 à cette zone seulement. Si vous avez compris cela, vous avez tout compris (si si, je vous l'assure).

Explication

Essayons un code tout de suite :

Code : Axe

```
:A  
:+4  
:-A
```

Même pas une erreur de compilation, tout marche, comment est-ce possible ?

C'est grâce à cette zone temporaire ! D'ailleurs elle a un nom cette zone, on parle du **registre HL**.

En réalité vous utilisez déjà les avantages du registre HL dans tous vos programmes :

Secret (cliquez pour afficher)

Par exemple au lieu de faire :

Code : Axe

```
:0→A  
:0→B
```

Vous faites déjà :

Code : Axe

```
:0→A→B
```

Pour ceux qui ont fait du TI-Basic, c'est l'équivalent de *Ans*, sauf qu'il ne s'agit pas d'une "variable" qu'il faut indiquer.

Optimiser à l'intérieur d'une commande

Maintenant, rentrons dans le vif du sujet, pourquoi pas avec une commande *Rect()* pour afficher simplement un carré en haut à gauche dans le buffer :

Code : Axe

```
:Rect (0, 0, 8, 8)
```

Comment gagner pleins d'octets dans un programme :

Code : Axe

```
:Rect (0,,8,,)
```

Et oui, la commande *Rect()* ne regarde pas ce qu'on lui propose, elle regarde ce qu'il y a dans le registre HL. Et il y a pléthore de commandes en Axe qui peuvent ainsi être optimisées !

Optimiser les conditions

Passons à quelque chose de plus technique avec les conditions :

Code : Axe

```
:getKey (12) ?1→V
```

Là encore on gagne des octets :

Code : Axe

```
:getKey (12) ?→V
```

De même lorsqu'on utilise les optimisations vu précédemment :

Code : Axe

```
:A=7?1→V
```

Cela donne :

Code : Axe

```
:A-7??+1→V
```

Aussi incroyable que cela puisse paraître, cela coûte moins d'octets d'ajouter 1 au registre HL quand il est à 0, que de lui donner la valeur 1 !

Optimiser les boucles

Vous connaissez la commande For() qui prend trois arguments, très souvent la variable de la boucle est utilisée en multiplication :

Code : Axe

```
:For (A, 0, 7)
:A*8
:End
```

Sachez qu'il existe une commande For() à **un argument** qui se répète le nombre de fois indiqué en argument, et ce de manière très optimisée :

Code : Axe

```
:0
:For (8)
:-A
:.Code
:A+8
:End
```

L'avantage de cette boucle est qu'elle ne modifie pas le registre HL, donc on peut facilement faire passer HL du début à la fin de la boucle.

Maintenant on pourrait très bien utiliser une boucle *While* qui se répète N fois de cette même manière :

Code : Axe

```
:N
:While
:-1-A
:.Le code à mettre dans la boucle
:A
:End
```

La commande *While* va tester ce qu'il y a dans le registre HL, donc N au début, puis A (qui vaut N-1), et cetera.

Bien sûr ces exemples exploitent abusivement l'avantage du registre HL, mais c'est pour vous montrer jusqu'à quel point on peut agir sur la taille et la vitesse de nos programmes. 😊

Utiliser HL à travers des fonctions

Une dernière chose sur le registre HL, sachez que cela marche aussi quand on appelle une fonction. C'est intéressant notamment si vous n'avez qu'un argument :

Code : Axe

```
:A
:FX ()
:
:Lbl FX
:-r1
:.La variable A est stocké dans r1
```

C'est en combinant toutes ces techniques que le registre HL montre sa puissance. N'hésitez pas à vous entraîner, mais attention à la lisibilité de vos programmes... N'optimisez que lorsque c'est vraiment nécessaire.

Le dilemme entre la taille et la vitesse du programme

Dans la programmation z80, il existe un dilemme entre optimiser un programme pour qu'il soit plus rapide ou (exclusif) pour qu'il soit plus léger.

En Axe Parser la plupart des routines sont optimisées pour réduire au maximum la taille des programmes. Néanmoins lorsque vous programmez des jeux nécessitant de la vitesse, il vous sera nécessaire de connaître les secrets de la rapidité dans un programme ! 😊

Les conditions

Vous croyez avoir déjà fait le tour des conditions, pourtant si on reprend une condition simple vue plus haut :

Code : Axe

```
:!If A-10 + (B-D)
:End
```

Ce code est très léger en octets dans un programme, mais si vous voulez augmenter la vitesse de ce test il vous faudra effectuer cette condition en deux temps, comme ceci :

Code : Axe

```
!If A-10
:!If B-D
:End
:End
```

Dans certains cas, comme ici, la taille du programme est encore plus réduite (4 octets de moins). Mais le réel avantage est que les tests se font plus rapidement, or les conditions sont les outils les plus utilisés dans une boucle, imaginez le **gain de vitesse** si vous optimisez toutes vos conditions ! 😊

Les multiplications et les divisions

En Axe les divisions et les multiplications ont leur propre routine : respectivement 46 octets pour la division et 18 octets pour la multiplication.

 C'est à dire que à chaque fois que je fais une division 46 octets seront ajoutés au programme ? 😊

Non, à partir d'une seule division la routine est ajoutée, c'est donc très rentable pour réduire la taille du programme si vous faites beaucoup de divisions dans un même programme.

Seulement voilà, il existe certaines divisions et multiplications en Axe qui sont optimisées ! 😊

Je vous invite à ouvrir le fichier "Auto Opt.txt" dans le zip de l'Axe Parser, et d'y regarder attentivement les lignes sur la multiplication et la division. Comme il y a moins de divisions, je vais prendre ce dernier exemple :

Citation : Auto Opt.txt

Expression	Nombre d'octets
/VAR	7 + sub_div
/CONST	6 + sub_div
/2	4
/10	3
/128	5
/256	3
/512	5
/32768	5

En fait pour les nombres 2, 10, 128, 256, 512 et 32768 la division prend moins d'octets à être faite que d'appeler la routine de division. Il faut savoir que ces divisions seront également beaucoup plus rapides !

Prenons l'exemple suivant :

Code : Axe - Divisions par 2

```
:A/8
.: s'optimise au détriment de la taille en :
:A/2/2/2
```

Le premier code est deux fois plus léger que le second, mais ce dernier est 15 fois plus rapide.💡



Comment est-ce possible ?

En fait 8 sera considéré comme une constante et le programme fera appeler à la routine de division. Seulement celle-ci prend beaucoup plus de temps à renvoyer la valeur car elle est faite pour prendre le moins de place possible !

C'est exactement le même système pour les multiplications :

Code : Axe

```
:A*96
.: Est beaucoup plus rapide de la manière suivante
:A*32*3
```

C'est également le cas pour les divisions signées :

Code : Axe - Divisions par 2

```
:A//16
.: s'optimise au détriment de la taille en :
:A//2//2//2//2
```

N'hésitez pas à vous aider du fichier "Auto Opt.txt" pour gagner ainsi en rapidité dans vos programmes ! 😊

Autres conseils

Quelques commandes d'optimisation

Il existe des commandes en Axe conçues pour simplifier vos codes, en voici quelques-unes.

Select()

La commande Select() permet de retourner dans le registre HL son premier argument, même si le deuxième argument modifie sa valeur. Prenons ce code en exemple :

Code : Axe

```
:X→A
:X+2→X
```

On peut le simplifier de la manière suivante :

Code : Axe

```
:Select(X,+2→X)→A
```



Select() se trouve ici :



Bien évidemment on peut utiliser toutes les valeurs envoyées dans le registre HL :

Code : Axe

```
:B/2
:Select(, +2→X)→A
```

Organisez vos chaînes de caractères avec stdDev()

Lorsque vous avez besoin de mettre beaucoup de chaînes de caractères dans votre programme, j'imagine que vous utilisez la méthode suivante :

Code : Axe

```
: "Plop world"→Str1
: "Phrase 1"→Str2
: "Phrase 2"→Str3
: .Etc
```

Le problème est que les chaînes de caractères ont un nombre d'octet indéterminé, donc soit vous utilisez la méthode ci-dessus, soit vous comptez les octets un par un (très fastidieux!), soit vous utilisez la commande *stdDev()*!



Cette commande prend deux arguments : le pointeur de la première chaîne de caractères, un nombre N. Elle retourne le pointeur de la Nième chaîne en partant de zéro.

Seulement, pour que cela marche il faut terminer toutes les chaînes de caractères, sauf la première, par des zéros de la manière suivante :

Code : Axe

```
: "Plop world"→Str1
: "Phrase 1"[00]
: "Phrase 2"[00]
:
: Disp stdDev(Str1,1)
: Affiche "Phrase 1"
```

Imaginez dans une boucle affichant un menu ou beaucoup de texte, ici les 8 premières chaînes de caractères :

Code : Axe

```
: For(I,0,7)
: Output(0,I,stdDev(Str1,I)
: End
```

Z-Test() : optimiser les Goto

Dans la plupart de vos programmes, notamment pour les menus, vous avez besoin d'aller aux labels en fonction d'une variable contenant le choix du menu. Votre premier réflexe sera plus ou moins de faire cela :

Code : Axe

```
: A contient le choix du menu
:
: !If A
: Goto LBL1
: Else! If A=1
: Goto LBL2
: Else! If A=2
: Goto LBL3
: End
```

Grâce à la commande *Z-Test()*, tout ceci est fini, voici ce que cela donne :

Code : Axe

```
: Z-Test(A,LBL1,LBL2,LBL3)
```



Si le premier argument, ici A, vaut 0, il va au premier label, s'il vaut 1 il va au deuxième label, s'il vaut 2 au troisième, etc. Et s'il est trop grand (par exemple si A vaut 4), alors il continue dans le programme.

Après si vous voulez faire appel à ces labels comme des fonctions, rien de plus simple :

Code : Axe

```
:Lbl Test
:Z-Test(A,LBL1,LBL2,LBL3,etc)
:Return
```

Spécificités du compilateur

Il existe des optimisations inclassables qui sont beaucoup moins "logiques" : parfois ce sont des bugs ou juste des particularités de l'Axe Parser qui permettent d'optimiser vos programmes.

Les constantes

Dans un calcul, mettre les constantes en dernier dans une expression rend le calcul un peu plus rapide :

Code : Axe - Constantes

```
: préférez
: A+2→A
: Au lieu de
```

```
:2+A→A
```

Placer une expression à l'intérieur d'une commande

Si vous voulez modifier X comme dans le cas suivant :

Code : Axe

```
:Px1-On (X, Y)
:X+r1-X
```

Il existe un moyen de modifier X à l'intérieur d'une commande, tout en utilisant une autre variable pour l'argument suivant :

Code : Axe

```
:Px1-On (X, (+r1-X) Y)
```

La parenthèse sera ignorée par le compilateur, mais X sera bien modifié avant de copier Y dans le registre HL.

Utiliser un pointeur après l'avoir modifié

Avez-vous déjà essayé de donner la même valeur à plusieurs octets pointés par des variables :

Code : Axe

```
:L1→A
:0→{A}→{A+1}
:Disp {A+1}►Dec
:Pause 1800
```

Pourquoi cela affiche 34540 ? 🤔

En fait lorsqu'on modifie un octet pointé par **une variable**, l'adresse de cet octet sera retournée dans HL. A première vue c'est un bug, mais en y réfléchissant bien ça peut servir à optimiser un programme. 😊

Essayez ce code pour stocker les sinus des dix premiers multiples de 8 dans L1 :

Code : Axe

```
:L1→A
:0
:For (10)
:sin (-I)→{A}+1→A
:I+8
:End
```

La fin d'un programme est un Return

Ceci n'est qu'un rappel, mais **la fin d'un programme est un Return**, donc ce n'est pas la peine de mettre un *Return* après la dernière fonction :

Code : Axe

```
:Lbl A
:.Code
:Return
:
:Lbl Z
:.Code
```

Comme vous l'avez sûrement remarqué, il s'agit plus de trucs et astuces que de méthodes globales. Mais c'est ainsi que se joue l'optimisation : dans les détails. N'hésitez pas à expérimenter et partager vos découvertes !

Un dernier conseil : pour être efficace, soyez ordonnés. C'est-à-dire :

- Commencez par penser ce que vous voulez faire, et n'oubliez pas, le crayon et le papier sont les meilleurs amis du bon programmeur.
- L'éditeur sur calculette est assez peu lisible : aérez et commentez vos programmes.
- Sauvegardez régulièrement vos programmes, sur votre ordinateur, en activant la sauvegarde automatique ou en utilisant l'outil de groupage ().

Ça ne rendra pas votre programme plus rapide, mais vous fera gagner énormément de temps. Et le temps, c'est précieux. 😊

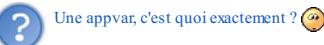
Manipuler les variables de la calculatrice

Si vous en avez marre de ne pas pouvoir sauvegarder vos scores, exploits ou autres lors de l'exécution d'un programme, si vous désespérez de trouver une solution, sachez que votre souffrance sans limite se termine ici et maintenant !!! 😊 Comment ça j'exagère ?

Appvars et programmes

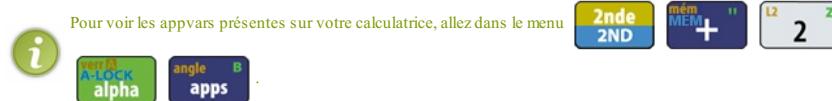
En Axe, les programmes et les appvars se manipulent à peu près pareil. Leur nom peut être de 8 caractères maximum et le premier caractère ne peut pas être un chiffre. Les appvars peuvent contenir des lettres minuscules dans leur nom, mais il est préférable pour les programmes d'avoir un nom en majuscule.

Manipuler une appvar



Une appvar, c'est quoi exactement ? 😊

A l'origine cela vient du terme *application variable*. Ce sont des plages de mémoire faites pour les applications de la calculatrice, mais on peut les utiliser en Axe aussi !



Créer, modifier, archiver, supprimer une appvar

Pour manipuler une appvar, il faut d'abord que celle-ci soit désarchivée impérativement ! Ensuite on utilisera la commande



Code : Autre

```
GetCalc("appvAPPVAR")→Pointeur
```

Où *APPVAR* est le nom de l'appvar et *Pointeur*, le pointeur permettant d'accéder à l'appvar.



Attention, le pointeur n'est pas statique, donc il vous faut utiliser une variable :

Code : Autre

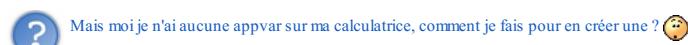
```
GetCalc("appvAPPVAR")→A
```

Cela enregistre l'adresse de l'appvar APPVAR dans la variable A. On peut ensuite accéder aux données dans l'appvar en utilisant A comme un pointeur :

Code : Autre

```
1→{A}
```

Le code ci-dessus enregistre 1 dans le premier octet de l'appvar APPVAR. En fait c'est **exactement pareil que les listes et les datas** ! 😊



Quand l'appvar n'existe pas, on peut facilement la créer. Toujours en utilisant la commande `GetCalc()`, mais cette fois en y ajoutant un deuxième argument qui est la taille de l'appvar :

Code : Autre

```
GetCalc("appvAPPVAR", Taille)→Pointeur
```

Où *Taille* est la taille en octets de l'appvar que l'on veut créer.

Quand on n'a plus besoin de l'appvar, on peut la supprimer, comme ceci :

Code : Autre

```
DelVar "appvAPPVAR"
```

On trouve la commande `DelVar` ici :



Cela marchera aussi bien si l'appvar est archivée ou désarchivée !



Heureusement vous pouvez archiver vos appvars et ainsi mettre en sûreté vos données :

Code : Autre

```
Archive "appvAPPVAR"
```

De même vous pouvez les désarchiver :

Code : Autre

```
UnArchive "appvAPPVAR"
```



Les commandes *Archive* et *UnArchive* se trouvent dans le menu



Toutefois, on ne pourra plus sauvegarder des données dans l'appvar tant qu'elle sera archivée.



Cela prendra de plus en plus de temps à s'exécuter si vous n'avez pas fait de *Garbage Collect* récemment. De plus n'abusez pas des fonctions d'archivage et de désarchivage dans un programme, cela le ralentit énormément et augmente le nombre de *Garbage Collect* encore une fois.

Utiliser les fichiers

Pour éviter les *Garbage Collect* incessant, il existe un moyen de lire les données de l'appvar sans avoir à la désarchiver :

Code : Autre

```
GetCalc("appvAPPVAR", Fichier)
```

Où *Fichier* est le symbole d'une fonction (Y_0, \dots, Y_9) que l'on trouve dans le menu



Une fois que l'appvar est copié dans un Fichier, on peut la lire comme un pointeur.

Citation : zéro

Super, je vais pouvoir utiliser la mémoire comme de simples Data !

NON, les fichiers ne peuvent qu'être **Ius** comme des Data, et **seulement** de la manière suivante :

Code : Autre

```
GetCalc("appvAPPVAR", Y0)
{Y0}→A
```

Ici cela sauvegarde le premier octet de l'appvar dans la variable A.

Pour récapituler, laissez tomber l'idée d'utiliser les fichiers avec des commandes comme *DispGraph*(Y_0), *Pt-On*(Y_0) et bien d'autre.

Un autre avantage des fichiers : on n'a pas à s'en soucier, ils seront supprimés automatiquement à la fin du programme. On peut donc utiliser un même Fichier pour lire une appvar puis une autre.

Code : Autre

```
GetCalc("appvAPPVAR", Y0)
GetCalc("appvAPPVAR2", Y0)
```

Ce code copiera l'appvar APPVAR dans le Fichier Y_0 , puis ça copiera l'appvar APPVAR2 dans le Fichier Y_0 . Désormais, lorsqu'on utilise le pointeur Y_0 , on accède aux données de APPVAR2.

Encore une fois, retenez bien que de cette manière, on peut uniquement lire des données ! 😊

Exercice

Pour voir si vous avez bien compris les concepts cités précédemment, rien de mieux qu'un peu d'exercice.

Le but de cet exercice va être de créer une appvar qui contiendra le nombre de fois que le programme a été ouvert et qui l'affiche à l'écran de la TI. Pour cela, vous aurez besoin de ce que nous avons vu ci-dessus ainsi qu'un peu de réflexion.

Voici le résultat !

Secret (cliquez pour afficher)

Tout d'abord, comme on ne sait pas si l'appvar est archivée ou pas, dans le doute il vaut mieux tenter de la désarchiver :

Code : Axe

```
: "appvAPPVAR"→Pic1
:UnArchive Pic1
```

A présent, si l'appvar n'existe pas, nous devons la créer, et dans le même temps l'initialiser à 0 :

Code : Axe

```
: !If Getcalc(Pic1)→P
:Return!If GetCalc("appvAPPVAR", 1)→P
:0→{P}
:End
```

Une appvar d'un octet devrait être suffisant pour contenir le nombre de fois où on a lancé le programme.



N'oubliez pas d'arrêter le programme si l'appvar n'a pas pu être créé ou sinon vous risquez le Ram Cleared.

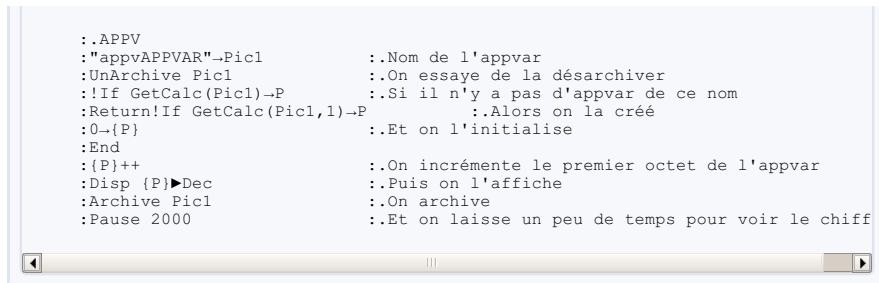
Ensuite, il ne reste plus qu'à incrémenter l'appvar une fois :

Code : Axe

```
: {P}++
```

Pour terminer, on archive l'appvar pour éviter qu'elle soit supprimée par un RAM Cleared, ce qui donne comme code final :

Code : Axe



```

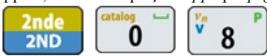
:APPV
:"appvAPPVAR"--Pic1      :.Nom de l'appvar
:UnArchive Pic1           :.On essaye de la désarchiver
:!If GetCalc(Pic1)--P     :.Si il n'y a pas d'appvar de ce nom
:Return!If GetCalc(Pic1,1)--P :.Alors on la crée
:0→{P}                   :.Et on l'initialise
:End
:{P}++
:Disp {P}►Dec            :.On incrémente le premier octet de l'appvar
:Archive Pic1             :.On archive
:Pause 2000               :.Et on laisse un peu de temps pour voir le chiff

```

Manipuler un programme...

...à partir d'un programme en Axe

Pour accéder aux programmes, on procède de la même manière que pour une appVar, mais en remplaçant *appv* par *prgm*. Encore une fois, ne le tapez pas en toutes lettres, sélectionnez-le dans le catalogue :



On a donc, pour créer (ou écraser) un programme *prgmFILE* de 4 octets et le pointer sur F :

Code : Axe

```
:GetCalc("prgmFILE", 4)→F
```

Ainsi toutes les commandes vues pour les appvars marchent pour les programmes exactement de la même manière ! 

Secret (cliquez pour afficher)

Code : Axe

```

:DelVar "prgmFILE"
:Archive "prgmFILE"
:UnArchive "prgmFILE"
:GetCalc("prgmFILE", Y0)
:{Y0}→A

```

... à partir de l'éditeur de programme !

Je vous vois déjà en train de vous ruer sur l'éditeur de programmes pour voir ce qui se passe lorsqu'on met des données dans un programme.



Secret (cliquez pour afficher)

Vous l'aurez remarqué très vite mais l'éditeur de programme convertit tous ses octets en tokens !

Ainsi il est très facile de faire un programme en Axe qui stockera une chaîne de caractères tokens dans un programme éditable :

Code : Axe

```

:.PGM
:"prgmPG"--Str1
:UnArchive Str1
:!If GetCalc(Str1)
:Return!If GetCalc(Str1,500)
:End
:Fill(→P,500,“E29”)
:input
:Copy(→A,P,length(A))

```



^{E29} est le token correspondant à un espace, donc ici votre programme sera entièrement blanc après la chaîne de tokens.

Un petit screen :



Les "vraies" variables et plus encore...

Les "vraies" variables

Accéder aux vraies variables est assez simple à présent.

Tout d'abord, il faut obtenir un pointeur vers une vraie variable :

Code : Autre

```
GetCalc("varA")→Pointeur
```

Où *A* peut être remplacer par n'importe quelles vraies variables, de A à Z ainsi que θ, et *Pointeur* est l'endroit où l'on souhaite que la donnée contenue dans la variable pointe.

 Le token *var* se trouve en faisant  +  . Axe transformera automatiquement le token *u* en *var*.

Ensuite, pour modifier la valeur de la vraie variable, il faut d'abord comprendre deux choses : la première c'est qu'en Axe on ne manipule que des variables de deux octets, alors que les "vraies variables" sont de **9 octets** ; on les appelle **floats**. Heureusement il existe une commande pour convertir un float en un format de 2 octets, c'est tout simplement la commande *float* :



Il suffit de faire :

Code : Autre

```
:GetCalc("varA")→P
:float{P}→A
:1→float{P}
```

Ce code stockera la vraie variable A dans la variable Axe A, puis enregistrera 1 dans la vraie variable A. Bien entendu, on ne peut enregistrer que des entiers allant uniquement de 0 à 65535 dans les vraies variables, puisque l'Axe Parser supporte seulement des nombres de 2 octets.

 Il n'y a pas besoin d'utiliser *r* après la commande *float* si on sauvegarde un nombre de 2 octets. 😊

Encore plus de variables !

En fait toutes les variables de la calculatrices sont manipulables, voici le tableau des tokens à mettre entre parenthèses pour les manipuler :

Exemple	Variable
"prgmABC"	Le programme ABC.
"appvABC"	L'appvar ABC.
"grpABC"	Le groupe ABC (seulement archivé).
"LABC"	La liste ABC.
"L1"	La liste L1
"varA"	La "vraie" variable A
"Str1"	La chaîne de caractères Str1.
"GDB1"	La base de données graphique GDB1.
"Pic1"	L'image Pic1.
"Y1"	La fonction Y1.
"[A]"	La matrice A.

Je fais très vite le tour de ceux qu'on n'a pas encore vus, dans tous les cas le principe reste toujours le même :

- Les groupes ne sont manipulables qu'avec les fichiers.
- Les listes sont composées de nombres réels, pour afficher le nombre de nombres réels dans cette liste :

Code : Axe

```
:GetCalc("L1")→A
:Disp {A-2}►Dec
```

- Les chaînes de caractères (String) sont de simples chaînes de caractère tokens. On peut obtenir la longueur d'une String en lisant les deux octets se trouvant avant celle-ci :

Code : Axe

```
:GetCalc("Str1")→A
:Disp {A-2}►Dec
```

- Les bases de données graphiques font 117 octets.
- Les images (Pic) de base font 756 octets seulement car la ligne du bas est ignorée.
- Les fonctions (Y1, Y2,...) sont également des chaînes de tokens, elles se terminent par un zéro.
- Les matrices ([A], [B]...) sont constituées de nombres réels, la largeur X*Y de la matrice se retrouve dans les deux octets précédant celle-ci :

Code : Axe

```
:GetCalc("[A]")→A
:Disp {A-1}►Dec      ::X
:Disp {A-2}►Dec      ::Y
```

Les variables cachées

Si, dans un programme vous craignez de manquer de variables car elles risquent d'être utilisées par un autre programme, il vous reste une solution : les **variables cachées, ou hackées**.

 QUOI !? On m'a toujours dit que c'était pas bien de hacker ! 😊

Ici, pas de question de pirater quoi ce soit, mais bien d'utiliser des choses qui nous sont littéralement cachées.

Attention cependant : l'utilisation des variables hackées peut conduire à de petits bugs comme un remplissage incompréhensible de la RAM, à des RAM cleared (au pire) ou encore des situations étranges telles que celle-ci :



Ah bah génial, pour l'instant tu nous as présenté que des inconvénients ... Y'a des avantages au moins 🍊 ?

L'avantage, il est de taille, car tenez-vous bien ...

Grâce à cette méthode, vous allez avoir 256 pics, 256 GDB, 256 variables et 256 fois tout autres variables dont le nom n'est pas modifiable, et ce à disposition pour votre programme seulement !!

En gros, vous allez pouvoir manipuler tellement de variables de la calculatrice que vous ne saurez plus quoi en faire. 😲

Le secret des chaînes de caractère

Précédemment, quand on voulait accéder à par exemple *Pic1*, on faisait comme ça :

Code : Axe

```
:GetCalc("Pic1")→A
:.C'est le token Pic1
```

Ici, on pointe en premier argument à *GetCalc()* une chaîne contenant les informations pour trouver la variable de la calculatrice désirée. Il suffit donc d'étudier les octets de cette chaîne.

Je viens de regarder sur ce site, pour Pic1 le token va être [6000]. 😊

C'est une très bonne trouvaille, cela va nous servir par la suite. Seulement il y a un problème, l'Axe Parser rajoute deux octets lors de la compilation : le premier est le zéro de fin de chaîne de caractère ; et le deuxième est rajouté en début de chaîne, c'est **le type de la variable** !

Les différents types de variables

Vous l'aurez compris, on cherche à utiliser les variables de la calculatrice en manipulant leurs octets :

- Le type de la variable (1 octet).
- Le nom de la variable (variant).
- Le zéro de fin de chaîne de caractère.

Premièrement, le type de la variable. Il y en a un paquet, alors voici un tableau pour résumer tout ça :

HEX	Type
00	Nombre réel
01	Liste
02	Matrice
03	Équation
04	Chaîne de caractère
05	Programme
06	Programme protégé
07	Image (Pic)
08	Base de donnée graphique (GDB)
0B	Nouvelle équation
0C	Nombre complexe
0D	Nombre complexe
14	Application
15	Appvar
16	Programme temporaire
17	Groupe



La présence de deux types pour spécifier un complexe vous donne deux fois plus de tokens, donc 512 Complex à disposition.

Cet octet de type va donc renseigner le type de variable à laquelle on va accéder. Il faut ensuite faire suivre cet octet par le nom de la variable.

Le nom des variables

Il y a deux catégories de noms de variables :

- Les noms modifiables (Groupe, programme, programme temporaire, programme protégé, appvar, application et les listes).
- Les noms **non-modifiables** (nombre réel, matrice, équation, chaîne de caractère, image, base de donnée graphique, nouvelle équation et nombre complexe).

Le principe est très simple, pour la première catégorie il suffit de mettre le type suivit du nom :

Code : Axe

```
: [15]"ABC"→Str1  
:.Cela revient au même que  
:"appvABC"→Str1[
```

Pour la seconde, c'est plus compliqué. En fait il faut indiquer le token de la variable comme nom, pour cela je vous conseil de regarder [ici](#).

Essayons avec [Pic1](#) de token [6000] :

Code : Axe

```
: [0760]→Str1  
:Data (0,0)→Pic1X  
:.Cela revient au même que  
:"Pic1"→Str1
```

Les variables hackées



Pourquoi il y a un pointeur *Pic1X* de rajouté ?

En fait en modifiant l'octet pointé par *Pic1X* on peut naviguer aisément [entre les différentes Pics de la calculatrice](#) !

Mais ce n'est pas le seul avantage : cet octet peut prendre 256 valeurs, on a donc accès à **256 Pics** de la calculatrice. Ce sont des variables qui n'existe pas normalement, et on les appelle variables hackées !

Le problème de ces variables est qu'elles ne sont pas reconnues par l'OS, mais elles peuvent très bien être utilisées dans un programme en Axe. Voici l'équivalent de la chaîne de caractère Str100 :

Code : Axe

```
: [04AA]→Str100  
:Data (100,0)
```

Et pareil pour tous les autres types de cette catégorie !



Les groupes et les applications ne se manipulent qu'avec les fichiers.

Donc les deux principaux intérêts des variables hackées pour les noms non-modifiable : pouvoir utiliser 256 variables de chaque type ; et pouvoir naviguer assez facilement entre les variables d'un même type en modifiant un octet.
Grâce à tout cela, vous pourrez créer highscores, achievements et même plugins pour des programmes déjà existants !

Maîtriser les fonctions

Les fonctions sont indispensables à un programme clair et concis. Dans ce chapitre, nous allons manipuler leurs adresses, utiliser la fonction anonyme, et bien plus !
Sachez que ce chapitre est tout à fait facultatif, vous pouvez très bien programmer des choses intéressantes sans le lire... Mais si vous voulez utiliser l'Axe aux maximum de ses possibilités, suivez-moi ! 

Un peu plus sur les arguments

Le pointeurs sont nos amis

Allez, on commence doucement par un petit exercice :
 Ecrivez une fonction qui incrémente une variable passée en argument.
 Il n'y a aucune difficulté... si vous avez bien suivi le chapitre sur les fonctions !

Secret (cliquez pour afficher)

Code : Axe - Fonction outrageusement simple

```
:0→B
:INCR(B)→B
:.B vaut maintenant 1
:
:Lbl INCER
:r1++
:Return
```

Facile, donc. Maintenant, écrivez une fonction qui incrémente deux variables passées en paramètres.

 Mais c'est impossible ! On ne peut renvoyer qu'une unique valeur !

Certes. C'est pour cela qu'il faut utiliser les **adresses** de ces variables.
 Au lieu de passer en argument la valeur d'une variable qui sera seulement copiée dans les variables temporaires *r1*, *r2*, etc, nous envoyons à la fonction l'adresse de la variable. Pour récupérer l'adresse d'une variable, mettons *B*, il suffit de la précédér du symbole "degré" :

Code : Axe - Passage par adresse

```
:0→B→C
:.. On envoie l'adresse de B et de C
:INCR(°B, °C)
:.B et C valent désormais 1
:
:Lbl INCER
:.On modifie la valeur pointée grâce aux crochets
:{r1}r++
:{r2}r++
:Return
```

Et voilà ! 

 Le symbole ° est disponible en faisant  .

Sauvegarder les variables temporaires

Mais ce n'est pas tout : on peut également sauvegarder la valeur des variables temporaires en ajoutant *'r* derrière le nom de la fonction (le même *'r* que celui utilisé pour le back-buffer, dans le menu angle) :

Code : Axe - Sauvegarde des variables temporaires

```
:0→B
:5→rl
:sub(SIX', B)
:.rl vaut toujours 5 !
:
:Lbl SIX
:6→rl
:Return
```

Je n'en ai jamais eu besoin, mais cela peut être utile dans le cas de fonctions imbriquées, si l'on veut éviter d'écraser les arguments de la fonction en appelant une sous-fonction :

Code : Axe - Fonctions imbriquées

```
:UN(1)
:
:Lbl UN
:1→rl
:sub(DEUX', 2)
:.rl vaudra toujours 1 ici
:Return
:
:Lbl DEUX
:2→rl
:Return
```



Cette syntaxe est relativement lente, car elle sauvegarde et restaure les six variables temporaires, elle est donc aussi assez gourmande en mémoire. A utiliser avec modération !

Passons maintenant aux choses sérieuses : les adresses de fonctions !

L'adresse des fonctions

Les fonctions ont une adresse

Depuis la version 1.0.0 de l'Axe Parser, il est désormais possible d'appeler une fonction par son adresse. Et oui, les fonctions sont des données stockées en mémoire, elles ont donc une adresse : pour l'obtenir il suffit de rajouter le petit *L* devant leur nom (celui utilisé pour les listes en TI-Basic).

Ce token est disponible ici : . Dans la suite, je le désignerai par 'L'.

Que peut-on faire avec l'adresse d'une fonction ? L'appeler déjà, avec la syntaxe suivante : *(Adresse)(Argument1, Argument2, etc.)*.

Code : Axe - Appel par adresse

```
: (lFUNC) (8,42)
:.Équivaut strictement à :
:FUNC (8,42)
```



Mais alors, à quoi cela sert, à part rajouter quelques octets inutiles ?

A rien plein de choses ! Vous pouvez vous en servir presque comme n'importe quelle adresse :

Code : Axe - Récupérer la taille d'une routine

```
:Disp lFEND-lF►Dec
:.Affichera la taille en octets
:
:Lbl F
:.Contenu...
:Return
:Lbl FEND
```

Vous pouvez également copier le contenu d'une routine vers une liste, ou vers une autre routine... les possibilités sont intéressantes !

Par exemple si on cherche à exécuter la routine dans L1, rien de plus facile :

Code : Axe - Récupérer la taille d'une routine

```
:Copy(lF,L1,lFEND-lF)
:.Copie la routine dans L1
:(L1)(r1,r2,...)
:.Exécute la routine à partir de L1
:Return
:
:Lbl F
:.Contenu...
:Return
:Lbl FEND
```

Cette exemple pourrait servir d'optimisation de vitesse d'un jeu, au lieu de chercher la bonne routine à exécuter, on copie dans L1 celle qu'il faudra utiliser au début du tour.

Mais il y en a une que je voudrais approfondir : le passage de fonctions en paramètres.

Une fonction comme argument

Observez ce code :

Code : Axe - Passage de fonction en paramètre

```
:41→B
:FUNC(lINCR, B)→B
:.B vaut désormais 42 !
:
:Lbl FUNC
:(r1)(r2)
:Return
:
:Lbl INCR
:r1++
:Return
```

Ne faites pas cette tête, je sais, cela peut être un peu difficile à comprendre.

Analysons ce code ensemble. On passe en premier argument l'adresse de *INCR*. *r1* contient désormais l'adresse de *INCR*. Puis on utilise la syntaxe avec les parenthèses qui permet d'appeler une fonction par son adresse, en lui passant en paramètre le deuxième argument. *INCR*, comme son nom l'indique, va retourner cet argument incrémenté d'abord vers la fonction *FUNC*, qui va elle-même le retourner vers le programme principal.

Ce code fait donc la même chose que :

Secret (cliquez pour afficher)

Code : Axe - Fonctions imbriquées

```
:41-B
:FUNC(B)-B
:.B vaut désormais 42 !
:
:.Ici FUNC est un intermédiaire assez inutile
:Lbl FUNC
:INCR(r1)
:Return
:
:Lbl INCR
:r1++
:Return
```

Seulement, si on a une autre routine nommée *DECR* par exemple (qui...décrémente !), il suffit de passer son adresse via *FUNC* pour qu'elle l'exécute, sans recopier le code ci-dessus en changeant *INCR(r1)* par *DECR(r1)* ! Puissant, non ? 😊

?

D'accord, d'accord... Mais à quoi ça sert puisque de toute façon, *FUNC* ne fait qu'appeler la fonction dont l'adresse est passée en paramètre ? Autant appeler *INCR* ou *DECR* directement !

Certes, mais *FUNC* peut appliquer d'autres calculs avant d'appeler la fonction donnée en argument, évitant de retaper le même code dans différentes fonctions.

Mais tout le potentiel de cette technique vous sera révélé lorsque nous aurons percé à jour l'identité de ces mystérieuses fonctions *lambda*... 😊

Lambda

Des fonctions bien anonymes

Mais qui sont donc ces mystérieuses fonctions lambda ? La lettre lambda (λ), onzième dans l'alphabet grec, revêt une importance particulière dans les langages dits fonctionnels (Haskell, Caml, etc). Elle permet de déclarer une fonction "anonyme" (je vous jure que le jeu de mot est fortuit 😊).

Et depuis la version stable de l'Axe Parser (1.0.0), l'Axe intègre lui aussi cette fonctionnalité. On déclare donc une fonction anonyme de la façon suivante : $\lambda(expression)$. Voyons tout de suite un exemple :

?

Le token λ se trouve à la place de log : touche  .

Code : Axe - Une fonction anonyme

```
:.Cette fonction toute bête renvoie la somme des deux arguments
: $\lambda(r1+r2)$ 
```

Comme vous le voyez, l'expression doit tenir en une ligne, et peut bien sûr impliquer jusqu'à six variables temporaires (donc six arguments). Le résultat est automatiquement renvoyé.

?

Mais alors, si cette fonction n'a pas de nom, comment l'utiliser ? 😊

Les fonctions anonymes s'utilisent...par leur adresse. 😊

Il suffit de stocker dans n'importe quelle variable et d'utiliser la syntaxe avec les parenthèses !

Besoin d'un exemple ? Voici :

Code : Axe - Utilisation d'une lambda

```
:. $\lambda$ ( renvoie une adresse que l'on stocke dans L
: $\lambda(r1+r2)$ -L
:.On appelle la fonction par son adresse
:(L)(17, 25)-B
:.B vaut maintenant 42 !
```

Comme vous le voyez, les lambdas s'apparentent un peu à des fonctions mathématiques : elles appliquent une transformation ou un test à partir des arguments, puis renvoient un résultat.

?

Il existe une autre manière d'utiliser les lambdas sans stocker le pointeur : $\lambda(r1+r2)(17, 25)$ -B

On peut également employer les lambdas...dans les lambdas. 😊

Code : Axe - Lambda délicieusement imbriquées

```
: $\lambda(r1+(\lambda(r1*r2*(\lambda(r1<r2)(r1, r2)))(r1, r2)))$ (3, 4)
:.Aie mes yeux...
```

Devinez donc ce que cette jolie lambda renverra... 😊

Secret (cliquez pour afficher)

Il suffit dans ce cas de remplacer tous les r1 par 3 et les r2 par 4, ce qui donne :

```
3+ (3*4 * (3<4))
(3<4) est vrai donc cela renverra 1. Il ne reste plus qu'à calculer :
3+(3*4)=15
Pas plus difficile que cela.
```

Ce genre de lambda bien dégoûtante est surnommée "curry" (amateurs de curry, n'y voyez aucune offense).

Pour une poignée de lambdas



Alors, que peut-on en faire, de ces lambdas ?

Beaucoup de choses ! En fait, on s'en servira principalement pour passer une fonction en argument, sans avoir à l'écrire en dur dans le code. Exemple :

Code : Axe - Passage de lambdas en arguments

```
:FUNC (λ(r1+r2), 6, 7)
:.Renverra 42
:FUNC (λ(r1=r2), 6, 7)
:.Renverra 1 car 6 est différent de 7
:
:Lbl FUNC
:(r1) (r2, r3)
```



Attention : dans l'exemple ci-dessus, la valeur de *r1* sera écrasée par celle de *r2* lorsque l'on fait appel à la lambda, qui utilise *r1* dans son expression. Cela risque de poser un problème si l'on essaye d'appeler (*r1*) une deuxième fois : plantage garantit ! Il faut donc soit sauvegarder *r1*, ou mieux la copier tout de suite dans une variable.

Allez, un petit exercice ! Codez-moi une fonction *MAP* qui prend en argument :

- L'adresse d'une fonction (prenant elle-même un seul paramètre).
- Un pointeur (comme *L1* par exemple).
- Un nombre N.

Le but est d'appliquer la fonction aux N octets après le pointeur et de retourner le résultat dans l'octet n°N venant d'être modifié.
Bonne chance !

Correction ! 🎉

Le défi peut sembler un peu ardu, mais simplement essayer consiste en un bon entraînement. Vous n'y arrivez vraiment pas ou vous pensez avoir réussi ?

Alors voici un code possible :

Secret (cliquez pour afficher)

Code : Axe - Une fonction map

```
:Lbl MAP
:r1=L
:r3--
:...
:On sauvegarde d'abord dans L l'adresse contenue dans r1,
:puisque r1 va servir de paramètre dans la lambda passée en argument.
:...
:.On initialise Z au pointeur passé en argument (stocké dans r2).
:For(Z,r2,r2+r3)
:.Puis on parcourt les r3 octets qui suivent, on s'arrête donc à r2+r3.
:
:(L) ({Z})→{Z}
:...
:On applique la fonction dont l'adresse est contenue dans L en lui passant la
:puis on stocke la valeur renvoyée.
:...
:End
:Return
```

Comme la boucle commence à partir de *r2+0*, il ne faut pas oublier de décrémenter *r3* en début de code pour avec les *r3* premiers octets de modifiés !

Si vous ne comprenez pas ce code, pas la peine de poursuivre : allez plutôt faire autre chose, et revenez ici à tête reposée.

Quant à ceux qui ont réussi, bravo, vous venez de créer votre première fonction de mappage ! 😊

Voici quelques utilisations possibles :

Code : Axe - Utilisations de MAP

```
:MAP(λ(0),L1,42)
:.Met les 42 premiers octets de L1 à 0
:
:MAP(λ(r1*3),L2,3)
:.Multiplie par 3 les 3 premiers octets de L2
:
:MAP(λ(Z-L1),L1,100)
:.Remplit L1 avec les entiers de 0 à 99
:
:MAP(λ(r1>42),L1,100)
:.Remplace par 0 tous les nombres strictement inférieurs à 42, par 1 sinon (dans
```

```

:Select(1,-{L1})→{L1+1}
:.L1 et L1+1 valent 1
:MAP(λ({Z-2}+{Z-1}),L1+2,10)
:.La célèbre suite de Fibonacci !
:
:MAP(lFUNC,L3,10)
:.Va appliquer FUNC aux 10 premiers octets de L3

```

Ces routines sont très utilisées dans les langages fonctionnels. Elles sont dites de "haut niveau", pas parce qu'elles sont compliquées, mais parce qu'elles modifient des données de manière abstraite : impossible de dire comment va être modifiée la liste en regardant la définition de *MAP*, il faut pour cela examiner la fonction passée en argument.

Maintenant nous allons découvrir les *folds* et les *filtres*. Que du bonheur ! 😊

Et pour quelques lambdas de plus

Comme vous êtes grand, et que vous avez tout compris, vous allez coder une jolie fonction *FOLD* tout seul. 😊

Le principe est très simple : c'est comme une fonction de mappage, sauf que le résultat de la transformation est enregistré **dans une autre liste**. Il va donc falloir rajouter un quatrième argument : un pointeur sur le début d'une autre liste (*L2* par exemple). C'est parti !

Fini ? Déjà ?

Secret (cliquez pour afficher)

Code : Axe - Fonction fold

```

:Lbl FOLD
:r3--
:r1→L
:r4→r5
:For(Z,r2,r2+r3)
:(L)({Z})→{r5}
:+1→r5
:End
:Return

```

Rien de bien compliqué par rapport à *MAP* : cette fois on envoie le résultat dans l'octet pointé par *r4*, en incrémentant *r5* pour faire "défiler" la deuxième liste.

On va donc l'utiliser comme cela :

Code : Axe - Utilisations de FOLD

```

:FOLD(λ(sin(r1)),L1,50,L2)
:.Va stocker dans L2 le sinus des 50 premiers octets de L1
:
:FOLD(λ(r1),L1,50,L2)
:.Copie les 50 premiers octets de L1 dans L2
:
:FOLD(λ(r1+2),L1,50,L1)
:MAP(λ(r1+2),L1,50)
:.Ces deux appels vont faire la même chose

```

Au tour des filtres. Comme *MAP*, il s'agit de parcourir un certain nombre d'octets dans une première liste, et de stocker dans une autre liste les éléments vérifiant une condition donnée par une fonction passée en paramètre.

Secret (cliquez pour afficher)

Code : Axe - Une fonction filtre

```

:Lbl FILT
:r1→L
:r4→r5
:For(Z,r2,r2+r3)
:!If (L)({Z})
:{Z}→{r5}
:+1→r5
:End
:End
:r5→r4
:Return

```

C'est toujours pareil : on parcourt un certain nombre d'octets dans une première liste, sauf que cette fois on regarde le retour de la fonction passée en paramètre : celle-ci doit donc consister en un test. Si l'élément considéré passe le test, on l'ajoute à la liste filtrée. On renvoie à la fin *r5-1* qui est égal au nombre d'éléments de la liste filtrée afin de savoir où elle s'arrête.

Un petit exercice : à l'aide d'un appel à *MAP* et d'un appel à *FILT*, stockez dans *L1* tous les entiers entre 0 et 99 dont le carré est divisible par 2 et par 3. Il vous faudra donc utiliser une liste intermédiaire. 😊

Correction :

Secret (cliquez pour afficher)**Code : Axe - Liste des entiers entre 0 et 100 dont le carré est un multiple de 2 et de 3**

```
:MAP(λ(Z-L2),L2,100)
:On stocke dans L2 tous les entiers entre 0 et 99
:FILT(λ((r1^2*2+(r1^2*3)),L2,100,L1)
:.Et on filtre !
```

**! If $r1^2*2+(r1^2*3)$ est l'optimisation de If $r1^2*2=0$ and $(r1^2*3=0)$** **Encore, encore !**

Au secours, je crois que je suis devenu accro aux lambdas ! Ça se soigne ?

Ne comptez pas sur moi pour ça ! 😊

Par contre, voilà une petite liste d'exos pour vous entraîner :

- Codez une fonction *ZIP* qui parcourt deux listes et stocke le résultat d'une fonction appliquée aux deux éléments dans une troisième liste ;
- Idem mais avec trois listes !
- Codez une routine *TWHIL* (pour *take while*) qui va copier les éléments d'une première liste vers une deuxième tant qu'ils vérifient la condition imposée par la fonction passée en argument.
- Au lieu d'appliquer la même fonction à une liste d'éléments, appliquez au même élément une liste de fonctions !

**Attention : les lambdas, c'est comme le chocolat ! Il ne faut pas en abuser. Sinon vos programmes vont se complexifier inutilement. Oui, je sais, les lambdas c'est beau et c'est la classe, mais quelques fois ce n'est pas la meilleure solution.**Voilà...C'est tout pour les lambdas. Passons à l'étude d'un autre animal étrange : la *récursivité*...**La récursivité****La récursivité, ça se mange ?**

Qu'est-ce que c'est que ça, la "récursivité" ? Une nouvelle marque de chocolat ?

Euh...non.

Citation : Wikipédia

Du point de vue de la programmation, une fonction récursive est une fonction, au sens informatique de ce terme, qui peut s'appeler elle-même au cours de son exécution ; on parle également de définition récursive ou d'appel récursif de fonction.

C'est donc le fait, pour une fonction, de s'appeler elle-même. Essayons tout de suite !

Code : Axe - Un programme sans fin

```
:FUNC()
:
:Lbl FUNC
:FUNC()
:Return
```

Évidemment, un tel code pose un problème : quand s'arrêtera-t-il ? Les petits malins qui ont essayé ça sur leur calculettes sans réfléchir en ont été pour leurs frais, car ce programme, théoriquement, ne s'arrête jamais ! Je dis bien théoriquement, car en réalité il va juste planter et faire un joli *Ram Cleared*, nous verrons pourquoi par la suite.

Pour ceux qui n'ont pas compris, une petite explication : on lance d'abord *FUNC*, on saute donc à *Lbl FUNC*. Puis la ligne suivante appelle *FUNC*, on revient donc à *Lbl FUNC*, et ainsi de suite. C'est une boucle infinie !

Return of the functionPour casser cette boucle infinie, il va falloir utiliser un *Return* quelque part : c'est la condition d'arrêt.

Si on n'en précise pas, la fonction va s'appeler indéfiniment et finir par faire sauter la *pile*. La *pile* est l'endroit de la mémoire où sont stockés les appels de fonctions et leur position dans le programme, afin de pouvoir y retourner une fois le code de la fonction exécuté. Lorsque l'on sort de la fonction, ces données sont libérées afin d'en accueillir d'autres, on dit qu'elles sont *dépilerées*.

Dans notre cas, on ne sort jamais des fonctions, et chaque appel imbriqué prend un peu plus de mémoire. Comme une calculette lambda (🍪) n'en contient pas beaucoup, c'est très vite le *stack overflow*, ou *dépassement de la pile*.

Que la récursivité soit avec vous !

Les fonctions récursives sont très utilisées en langage fonctionnel (oui je sais je suis chiant avec ça 🤪) car il n'y existe pas de boucles. Même si en langage impératif comme l'Axe leur usage est moins fréquent (surtout avec les possibles problèmes de pile), on peut faire des choses très élégantes avec.

Voyons un exemple très classique : le calcul d'une factorielle. Un petit rappel pour ceux qui n'ont pas suivi en maths : 😊

Pour tout entier n positif :

$n! = 1$ si n est nul.
 $n! = n * (n - 1) * \dots * 2 * 1$ sinon.

 $n!$ signifie évidemment "factorielle de n ".

On remarque un cas particulier dans la définition : $0!$ vaut 1. Notre intuition perspicace nous souffle qu'on pourrait l'utiliser comme condition d'arrêt.

 Mais, et cette fameuse récursivité alors ?

Elle se cache dans la propriété suivante, très simple, qui découle de la définition :

$$n! = n * (n - 1)!$$

Il suffit donc, pour calculer $n!$, de calculer $(n-1)!$, qui va se calculer grâce à $(n-2)!$, etc., jusqu'à $0!$ qui vaut 1. La voilà, notre récursivité !

C'est pas encore clair ? Un code siouplait !

Code : Axe - Factorielle

```
:Lbl FACT
:ReturnIf r1=0
:r1*FACT(r1-1)
:Return
```

On retrouve d'abord notre condition d'arrêt : $FACT(0)$ va retourner 1 car $(r1=0)$ sera vrai. Ensuite on applique juste notre propriété précédente.

Ce code est bien plus élégant, compréhensible et optimisé que son homologue itératif (i.e. avec une boucle).



La fonction vous renverra des valeurs erronées si vous lui passez 9 ou plus en argument, tout simplement parce que $9! = 362880$ et que cela dépasse donc largement la taille de deux octets.

Voilà, voilà, j'espère que vous êtes toujours vivants...

Les lambdas et la récursivité sont des facettes très intéressantes des langages fonctionnels, et les introduire dans un langage impératif tel que l'Axe permet de marier la puissance des deux univers. Amusez-vous bien avec et bonne prog' ! 😊

Maintenant que vous êtes rodés comme jamais contre tout programme en Axe, je vous conseille de lire l'annexe une dernière fois avant de partir pour donner le meilleur de vous-même. 😊

Partie 4 : Annexes

Pour vous simplifier la vie, voici quelques annexes utiles : liste de toutes les commandes en Axe, utilisation de TI-Connect, tableau des erreurs, émulation sur ordinateur...
A venir : un catalogue des fonctions utiles en Axe.

Utilisation de TI-Connect

Le premier obstacle du Zéro en matière de calculatrices est d'échanger des fichiers avec sa calculatrice. Cette annexe a pour but d'apprendre pas à pas comment utiliser le logiciel TI-Connect.

Installer TI-Connect

Ce n'est pas plus compliqué que pour un autre programme, et moins que pour l'installation de vos jeux préférés ! Nous allons voir comment installer un logiciel de transfert TI-PC sur différentes configurations.

 Sous Windows il ne faut pas brancher votre TI avant d'avoir complètement installé Ti-Connect, sinon les drivers ne s'installeront pas correctement.
De plus, ne cherchez pas à installer les drivers manuellement ou d'installer d'autre logiciel que Ti-Connect : 90% des problèmes de transmissions sont dus à ça.

Pour Windows 7 & Windows Vista

Tout d'abord, il faudra se procurer la bonne version de Ti-Connect :

- Si vous avez un ordinateur 32 bits, vous devrez utiliser Ti-Connect 1.6, disponible sur le CD fournit avec la calculatrice, ou bien [téléchargeable ici](#).
- Si vous avez un ordinateur 64 bits, il vous faudra Ti-Connect 1.6.1, qui n'est pas disponible sur votre CD, mais qui est [téléchargeable ici](#).

Il ne vous reste qu'à exécuter le programme pour l'installer, pas besoin d'aide pour ça (il suffit de cliquer sur "suivant" .

Pour Windows XP ou moins

Pour ces versions de windows, installez Ti-Connect 1.6, disponible sur le CD fournit avec la calculatrice, ou bien [téléchargeable ici](#).

Pour Mac

Je ne possède pas d'ordinateur Mac, mais pas de soucis : vous n'avez qu'à installer Ti-Connect pour MacOS, contenu dans votre CD ou bien le télécharger [ici](#). Je vous fais confiance. .

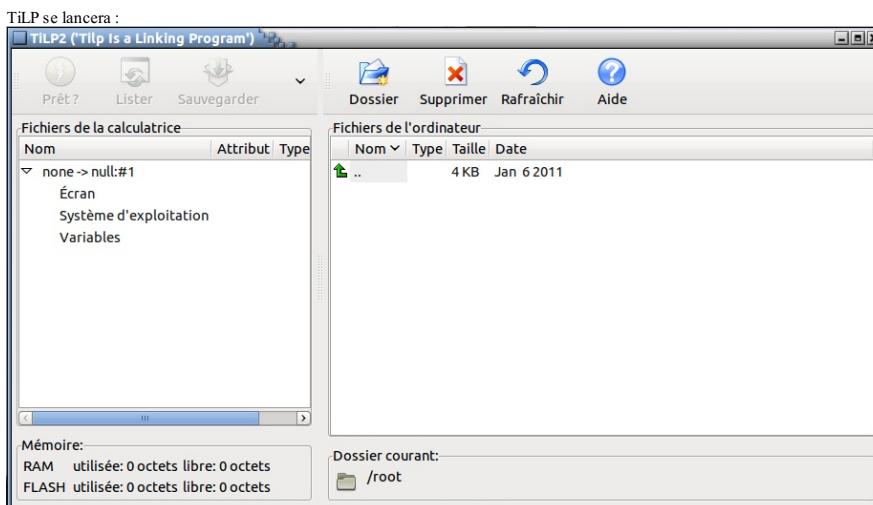
Pour GNU/Linux

C'est là que les choses se compliquent : Ti-Connect n'est pas compatible avec les systèmes Linux. Et pas question de l'utiliser avec Wine les transferts ne pourront pas s'effectuer. 
Nous allons donc utiliser un logiciel de transfert alternatif : TiLP. Vous pouvez l'obtenir ici [TiLP 2](#). En plus de cela, vous pourriez avoir besoin d'installer gfm, qui permet de créer et d'édition les fichiers TiGroup. .

Maintenant, allumez votre calculatrice, branchez la sur un port USB et tapez dans un terminal :

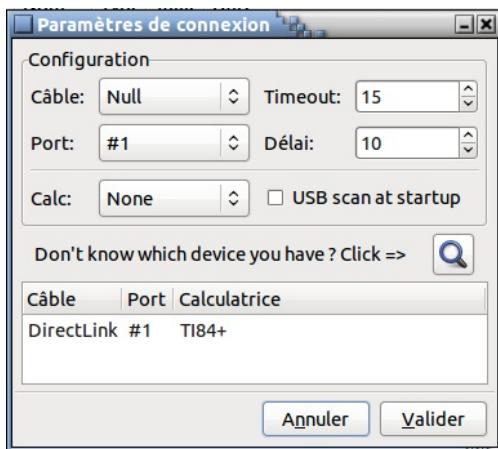
Code : Console

```
sudo tilp
```



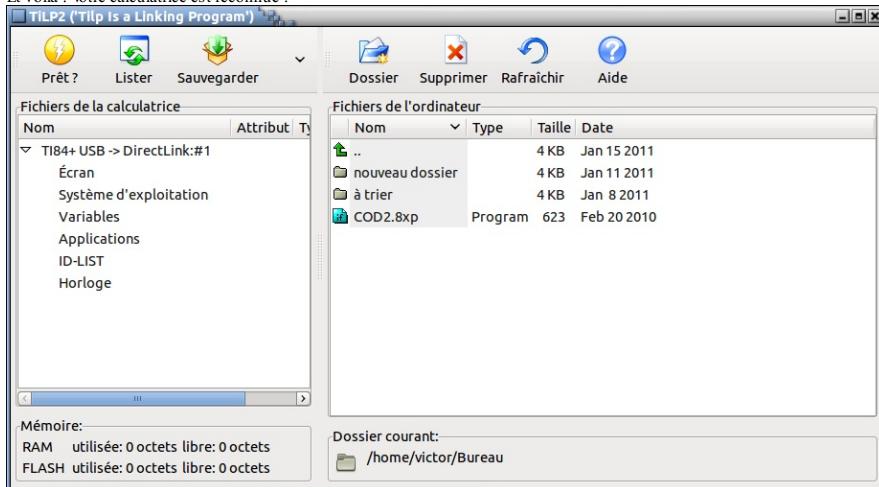
Ne vous en faites pas si, graphiquement, votre fenêtre diffère un peu de la mienne, tout dépend de votre thème. .

Mais votre calculatrice ne s'affiche pas ! Pour cela faites un clic droit sur *none -> null:#1*, cliquez sur *paramètres* et vous obtiendrez ceci :



Cochez alors *USB scan at start-up* et double cliquez sur le nom de votre calculatrice, dans la zone du bas (la ti84+ pour ma part).

Et voilà ! Votre calculatrice est reconnue !



Pour envoyer des programmes, faites des glissez vos programme de la fenêtre de droite à celle de gauche (sur *Variables* pour les programmes en basic). 😊

Brancher sa calculatrice

Ce n'est pas tout d'avoir installé un logiciel de transfert, il faut aussi relier la TI et l'ordinateur par câble. 😊

Nous allons voir ici les différents câbles qui existent ainsi que leur branchement.

Brancher sa TI-83+

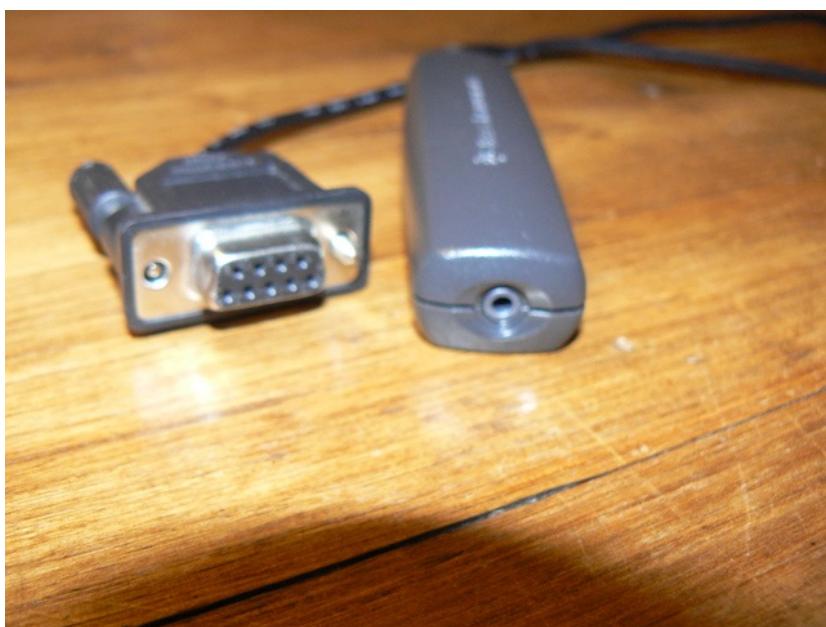
La TI-83+ peut être connecté à l'ordinateur via 2 types de câbles.

câble black

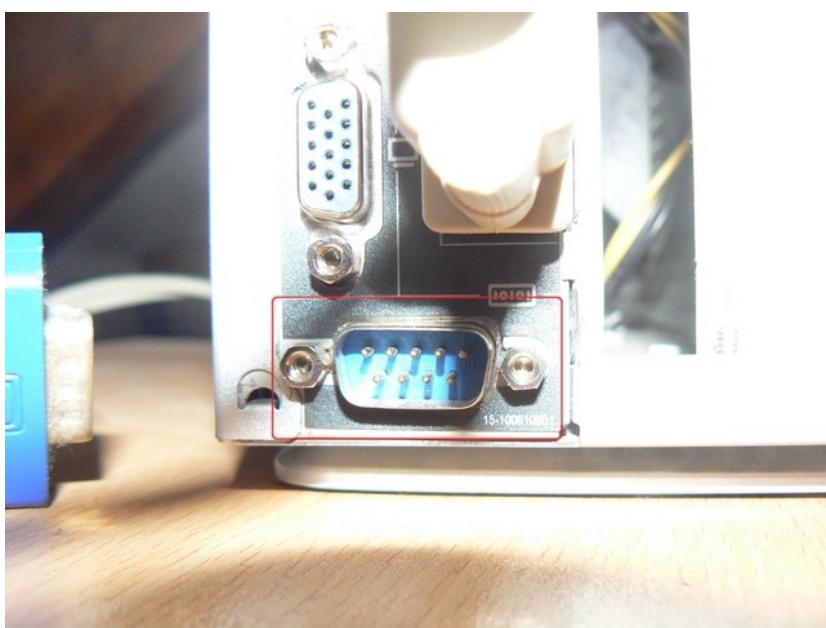
C'est le plus vieux des câbles. Il se branche sur le port I/O de votre calculatrice et sur une des grosses prise à l'arrière de votre PC (pas de risque de vous tromper sur le branchement : si la forme et la taille de la prise et du câble sont les mêmes c'est bon !).



à droite le câble TI-TI et à gauche l'adaptateur PC-câble. Il faut brancher les 2 ensembles pour avoir son câble PC-TI



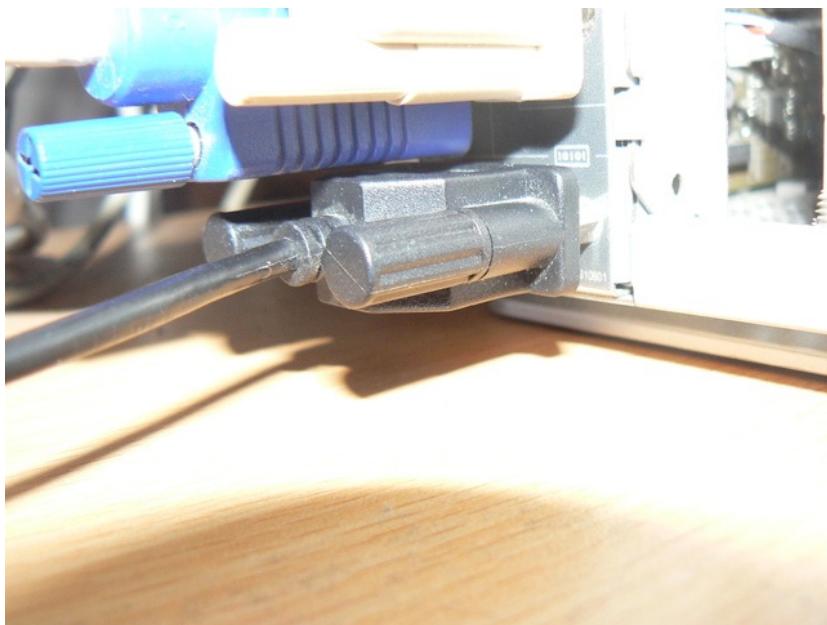
les deux bouts du câble adaptateur.



voici la prise sur lequel vous devez brancher le câble sur votre PC.



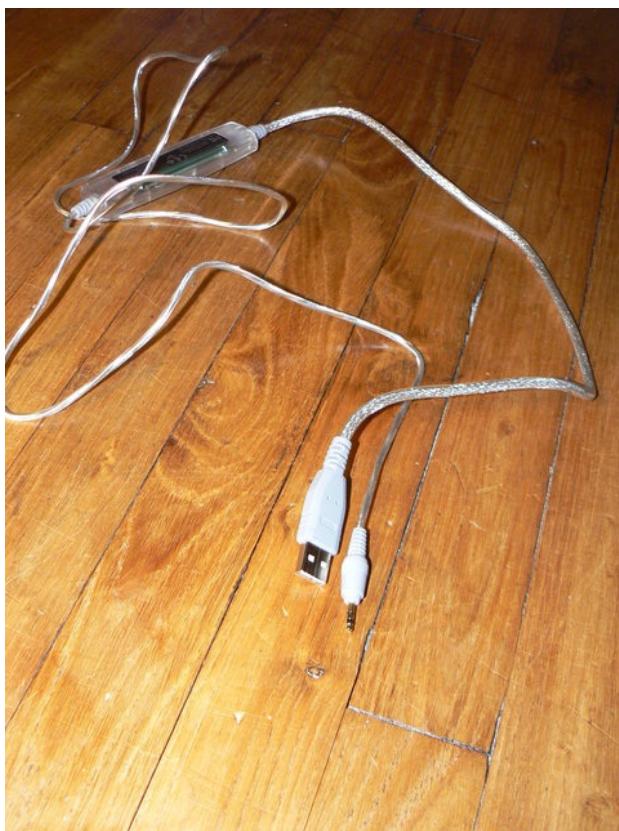
la TI branchée.



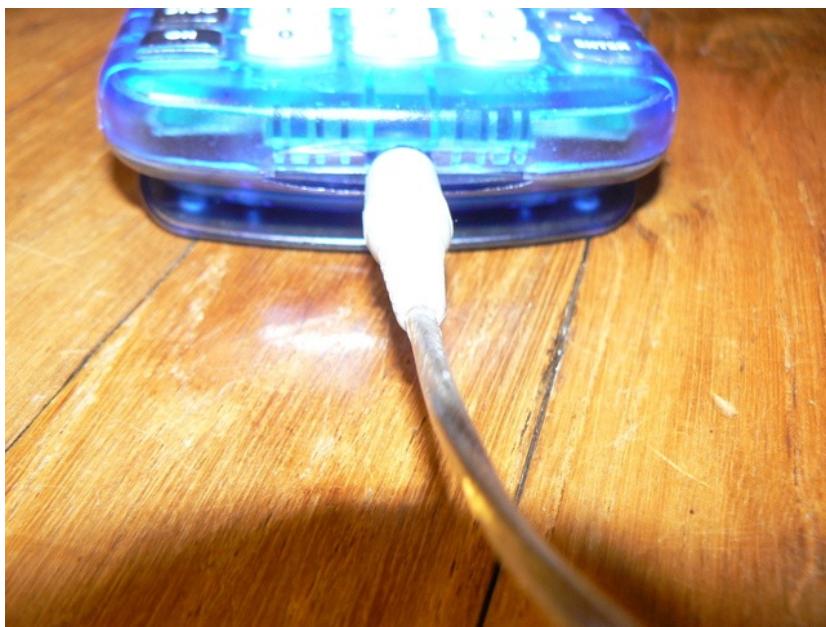
le câble est branché au PC, c'est parti pour les transferts ! 🎉

silver cable

C'est une version modernisée du câble précédent, on le branche d'un côté au port I/O de la ti83+ et de l'autre à un port USB de l'ordinateur.



Contrairement au câble black, ce câble est en une seule partie.



Le mini(jack branché sur le port I/O de la calculatrice.



Le port USB de branché sur votre ordinateur, et le tour est joué. 😊

brancher sa TI-84+

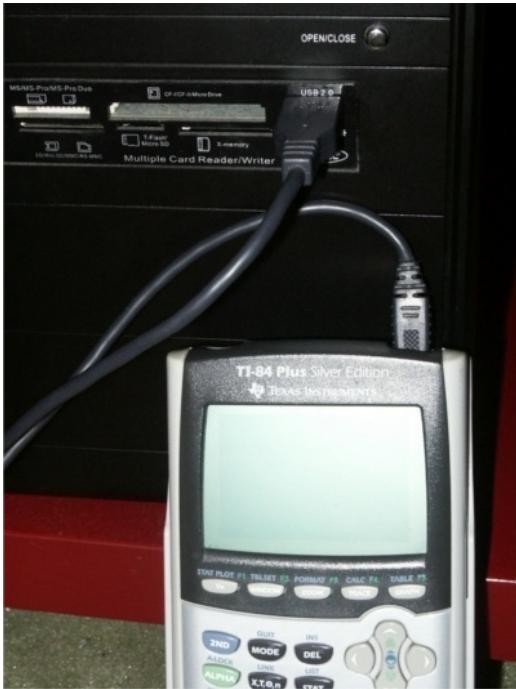
C'est encore plus simple : il vous faudra juste un câble mini-USB - USB normal. Normalement ce câble vous est fourni avec la TI. Si vous ne l'avez pas, n'importe quel câble générique de ce genre suffira. En général c'est ce genre de câble qui est fourni avec les appareils photos numériques ou certains PDA. 😊



le câble. En haut la partie mini-USB à brancher sur le port mini-USB de votre ti84. En bas la partie USB normale à brancher sur l'ordinateur.



La ti84+SE branchée.



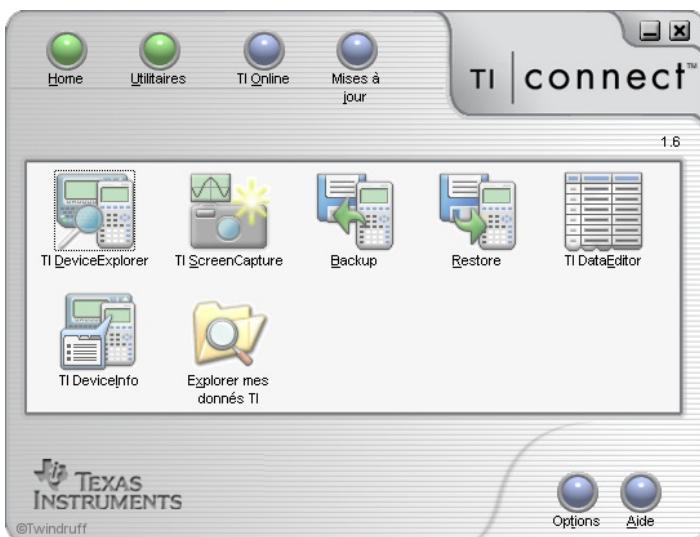
Maintenant que tout est installé et branché, vous pouvez enfin envoyer plein de programmes sur votre TI ! 😊

Transférer ses programmes

Enfin, la partie que vous attendiez tous ! Comment transférer des programmes sur sa TI. Vous allez voir, c'est très simple.

Le transfert

Ouvrez Ti Connect, vous devriez voir cette fenêtre apparaître :



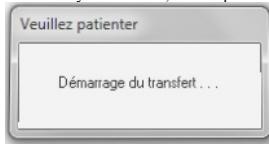
Détaillons un peu tout ces icônes :

- *TI DeviceExplorer* : la partie du programme la plus intéressante : c'est elle qui fait les transferts de données entre la TI et le PC.
- *TI ScreenCapture* : comme son nom l'indique, c'est un utilitaire qui permet de faire des captures d'écran de votre TI.
- *Backup* : très utile ! Ça permet de sauvegarder l'intégralité des données de votre TI. N'hésitez pas à faire des backups souvent ! 😊
- *Restore* : le frère de Backup : il permet de transférer le contenu d'un backup sur votre TI. Pratique pour récupérer ses données après un crash.
- *TI DataEditor* : pas très utile, il permet de créer ou d'édition des listes, matrices, etc...
- *TI Device Info* : il permet d'avoir des infos de base de sa TI : l'OS, l'état des piles (c'est approximatif), etc...
- *Explorer mes données TI* : il ouvre le dossier "Ti Connect" de votre ordinateur.

A présent, lancez TI DeviceExplorer. Il va chercher votre TI sur les différents ports de votre ordinateur. Si vous avez installé correctement le logiciel et si vous êtes bien sage, votre TI sera détectée 😊 :



Sélectionnez donc votre machine, un fenêtre s'ouvrira, vous y verrez tout ce que contient votre TI. Pour envoyer un fichier, rien de plus simple : un simple glisser-déposer sur la fenêtre en question !



Les fichiers transférables à votre calculatrice sont du type:

- .82* si vous avez une TI-82

- .83* si vous avez une TI-83 ou une TI-82.stats (fr comprise)
- .8x* si vous avez une TI-83+ ou TI-84+ (SE comprise)

Les problèmes fréquents

Il est fort probable que vous rencontriez lors de vos premiers transferts. 😊



Votre calculatrice est constituée d'une mémoire vive et d'une mémoire flash. Les programmes doivent être désarchivés (dans la mémoire vive) lorsqu'on les exécute, mais on peut les garder dans la mémoire flash. Les applications (comme l'Axe parser) sont obligatoirement enregistrées dans la mémoire flash.

Dès l'achat, TI a implémenté des applications... mais toutes ne sont pas très utile 😊. Du coup il faut aller dans le menu de la mémoire sur la calculatrice pour supprimer celles qui ne vous servent pas :



Cette erreur est très courante, elle vient du fait que le nom de votre programme (ou autre fichier de la calculatrice) est incorrecte, le plus souvent en minuscule. Il vous suffit de le renommer en **majuscule**.

i Je compléterai cette partie en fonction des problèmes principaux que je vous me signalerez (par message personnel par exemple).

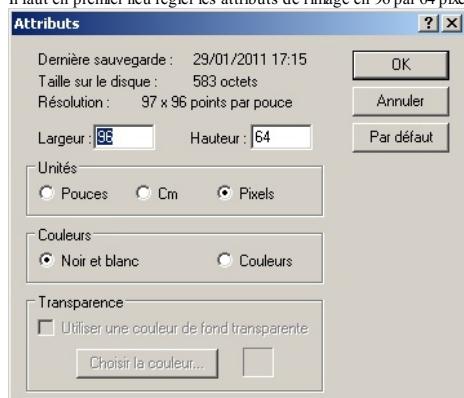
Autres outils

Créer une image pour sa TI.

Vous voulez faire une image sur votre ordinateur et l'envoyer ensuite sur votre TI au format .8xi (le format des images de la calculatrice) ? Rien de plus simple.

Tout d'abord ouvrez votre éditeur d'image préféré. Ici nous utiliserons Microsoft Paint.

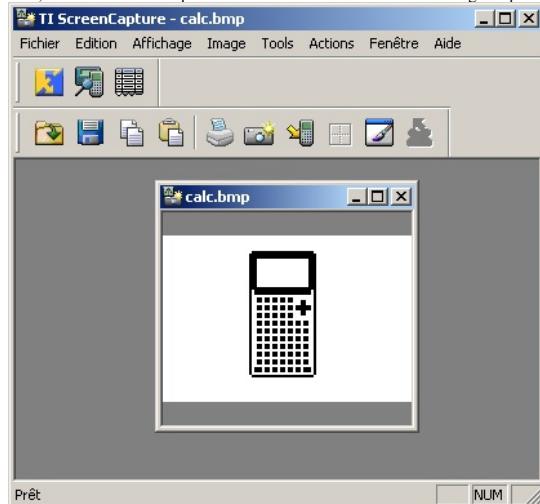
Il faut en premier lieu régler les attributs de l'image en 96 par 64 pixels (la dimension de votre écran de TI) :



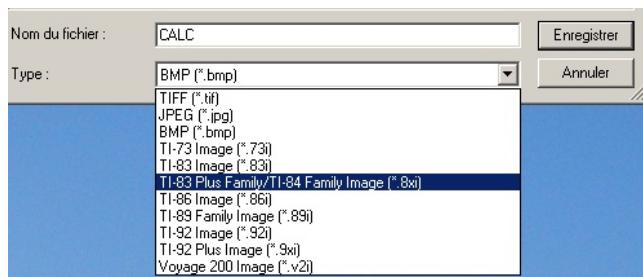
Puis dessinez à votre guise. 😊

Après cela, enregistrez votre image en format bmp monochrome.

Puis, ouvrez TI Screen capture dans Ti Connect et ouvrez votre image bmp :



Faites ensuite "enregistrer sous" et choisissez le format .8xi :



Votre image pour TI est maintenant prête ! 



Il vous suffit d'envoyer vos images comme vu plus haut pour un programme.

Tableau des erreurs et émulation



Arg mon programme n'arrive pas à compiler depuis tout à l'heure, il me signale une erreur, mais je n'y comprends rien !

Pas de problème, cette annexe est là pour vous aider, elle contient toutes les erreurs possibles lors de la compilation.

J'ai également rajouté dans ce chapitre du vocabulaire et quelques explications à propos des émulateurs.

Les erreurs

BAD CONSTANT	Un nombre non constant a été trouvé là où une constante était attendue.
CANNOT USE HERE	La syntaxe est correcte, mais ne peut pas être utilisée dans ce cas.
DECLARE FIRST	La constante n'a pas été déclarée avant utilisation.
DUPLICATE SYMBOL	Le pointeur statique ou le label existe déjà.
ELSE ONLY FOR IF	Un Else ne s'utilise qu'après un If...
INVALID AXIOM	Vous utilisez une librairie <i>Axiom</i> manquante ou corrompue.
INVALID HEX	Le nombre hexadécimal est invalide.
INVALID NUMBER	Le nombre est trop grand ou trop petit.
INVALID TOKEN	Le token utilisé n'est pas supporté par l'Axe.
INVALAID FILE USE	Vous avez mal utilisé un type de fichier.
LABEL MISSING	Le label appelé est inexistant.
LOW BATTERY	Vous ne pouvez pas archiver un fichier si vos piles sont faibles.
MISSING AN END	Une condition ou une boucle n'a pas été terminée par un End.
MISSING PROGRAM	Le programme que vous essayez d'inclure n'existe pas où n'est pas un programme en Axe.
MUST END COMMENT	Un commentaire multi-lignes n'a pas été fermé.
NAME LENGTH	Le nom est trop long pour un label ou un pointeur statique.
NO NESTED LIBS	Les librairies imbriquées ne sont pas (encore) supportées.
OS VAR MISSING	L'objet d'OS que vous essayez d'absorber n'existe pas.
OUT OF MEMORY	Il n'y a plus de place dans la RAM.
PARENTHESIS	Il faut fermer vos parenthèses.
SAME OUTPUT NAME	Le nom de votre exécutable est le même que celui de votre programme source.
TOO MANY AXIOMS	Seulement cinq <i>Axioms</i> sont autorisés par programme.
TOO MANY BLOCKS	Il y a trop de boucles ou de conditions imbriquées.
TOO MANY ENDS	Il y a des End inutiles.
TOO MANY BLOCKS	Il y a trop de boucles ou de conditions imbriquées.
TOO MANY SYMBOLS	Il y a trop de pointeurs statiques ou de labels.
TOO MUCH NESTING	Il y a trop de parenthèses imbriquées.
UNDEFINED	Le pointeur statique n'a pas été déclaré.
UNDOCUMENTED	L' <i>Axiom</i> utilise une instruction inconnue.
UNKNOWN ERROR	Le parser a rencontré un problème inconnu. Reportez l'erreur immédiatement !
WRONG # OF ARGS	Vous avez donné le mauvais nombre d'arguments.

Je tiens à rappeler que lorsque vous avez une erreur de compilation et que votre programme source est désarchivé, il vous suffit d'appuyer sur n'importe quelle touche sauf pour éditer votre programme à l'endroit précis où se situe l'erreur.

Utiliser un émulateur

Tout d'abord, pour programmer en Axe, il vous faut de la mémoire flash, et certains émulateur ne l'émulent pas.

Je vous proposerai ici d'utiliser 2 émulateurs :

- **Wabbitemu** : Il a l'avantage d'avoir beaucoup d'options très utiles, mais peut-être qu'il n'est pas assez stable.
- **Tilem** : reproduit fidèlement les calculatrices et est compatible linux, il a cependant moins d'options que wabbitemu.

Je présenterai seulement wabbitemu car les deux se manipulent à peu près de la même manière.



Il faut que vous sachiez utiliser TI-Connect pour continuer la lecture de cette annexe (voir utilisation de TI-Connect).

Téléchargement et installation

Choisissez votre émulateur et cliquez sur les liens plus haut pour arriver à la page de chacun.



Wabbitemu pour Mac existe : [ici](#).

Vous aurez donc une icône *download* ou *télécharger* sur la page, cliquez dessus, puis cliquez sur *I agree* (wabbitemu seulement).

Lors du premier lancement de wabbitemu.exe, une petite fenêtre s'affiche :



Si vous possédez déjà une ROM, cliquez sur *Browse*, sélectionnez la à partir de vos fichiers et passez à la partie suivante.

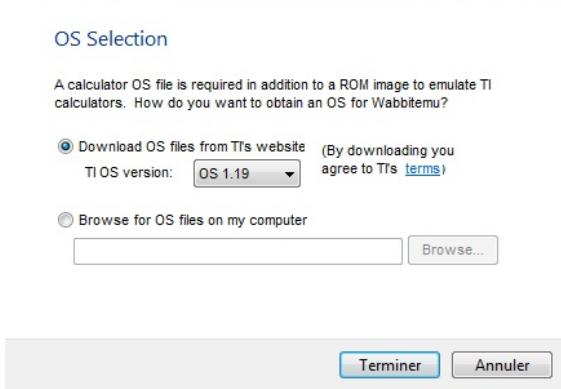
Si vous ne possédez pas de ROM, je vais vous apprendre à en créer une en cochant la case *Create a ROM image using open source software*

Sélectionnez votre calculatrice préférée parmi la liste, et cliquez sur suivant :



Les calculatrices TI-83+ et TI-84+ sont considérées comme des TI-83+.

Ensuite il vous propose un OS, cliquez sur terminer :

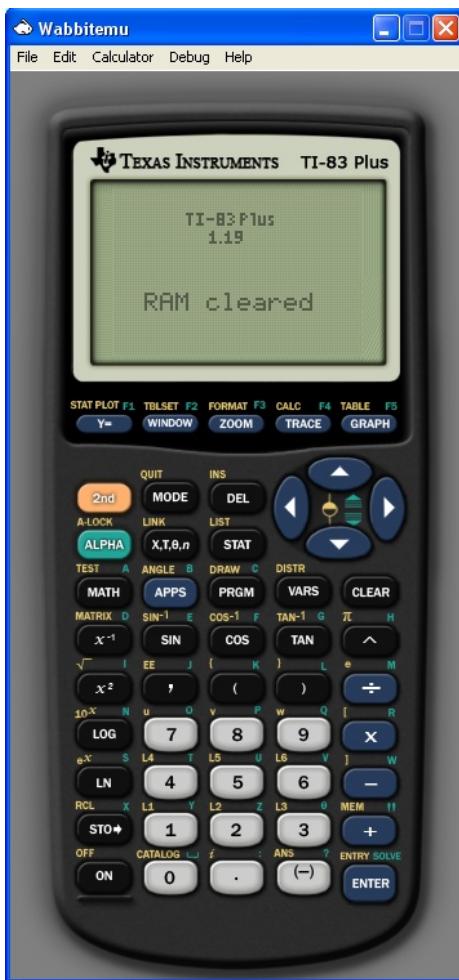


Il vous demandera alors où enregistrer votre ROM, puis l'émulateur démarrera avec votre calculatrice émulée. 😊

Manipulation l'émulateur

Mais il n'y a que l'écran d'affiché, comment je peux manipuler la calculatrice ? 😕

Il suffit de faire clic droit sur l'écran ==> *Calculator* ==> *Enable Skin* :



Rassurez vous, vous avez fait le plus dur. Maintenant il ne vous reste plus qu'à manipuler votre émulateur :

- Pour envoyer un fichier compatible (voir liste) sur votre calculatrice, faites un cliquer-glisser de votre fichier ou *clique droit ==> File ==> open* et indiquez l'emplacement.
- Pour prendre un screenshot animé, faites *clic droit ==> File ==> Record Gif* puis indiquez l'emplacement où vous souhaitez l'enregistrer. Votre screenshot sera enregistré dans une image *.gif.
- Il y a des raccourcis clavier pour les touches de la calculatrice, la liste est [ici](#).
- Pour envoyer un fichier de l'émulateur sur le PC, faites *clique droit ==> calculator ==> variables* et un cliquer-glisser suffit pour envoyer les fichiers.
- Si votre émulateur ne répond plus : *clique droit ==> Debug ==> Reset*.
- Pour sauvegarder votre ROM, avec toutes les données qu'elle contient (programmes, applications, etc.), faites *clique droit ==> File ==> save states*.
- Pour l'accès aux options, faites *clique droit ==> calculator ==> option*.
- Et plein d'autres encore que je vous laisse découvrir ! 😊

Peut être que programmer sur des émulateurs ne vous tente pas, mais au moins vous saurez comment prendre des screenshots avec vos programmes.

Liste des commandes

Voici le tableau de toutes les commandes de l'Axe Parser v1.1.2. Elles sont réparties selon leur contexte d'utilisation ou leur utilité.

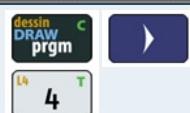
- En rouge toutes les expressions de calcul (variables, nombre fixe, pointeur statique, etc). Exemple : `A+2+Str1→B`
- En orange les constantes, nombre hexadécimaux, tokens et caractères ASCII. Exemple : Buff(`SIZE,CONST`)
- En bleu gris les labels. Exemple : Lbl `HEY`
- En bleu le reste. Exemple : `prgmTEST`

Système

Commandes	Touches correspondantes	Description
-		Les espaces sont ignorés dans la plupart des situations. Ils aident en général à l'organisation et à la compréhension du code.
:		Les deux petits points servent à sauter une ligne. Ils sont mis automatiquement lorsque l'on appuie sur la touche entrer .
;		Le point désigne une ligne de commentaire, sauf sur la première ligne où il indique le nom du programme.
...		Les points de suspension indiquent le début ou la fin de commentaires sur plusieurs lignes.
...If CONST		Commentaire conditionnel : commente les lignes suivantes (ou termine le commentaire) si la constante est égale à 0.
...!If CONST		Commentaire conditionnel : commente les lignes suivantes (ou termine le commentaire) si la constante n'est pas 0.
...Else		À mettre après un ...If CONST ou ...!If CONST : si le bloc précédent était commenté, le suivant ne sera pas commenté, et vice-versa.
DiagnosticOn		Active l'indicateur de calculs en cours (le défilement des pixels en haut à droite). Le programme affichera Done à la fin de l'exécution. trouver <code>DiagnosticOn</code> .
DiagnosticOff		Désactive l'indicateur de calculs en cours. Le programme n'affichera pas Done à la fin de l'exécution. trouver <code>DiagnosticOff</code> .
Full		Le mode pleine vitesse est activé sur les calculatrices les plus récentes, rendant le programme 3 fois plus rapide. Renvoie 0 s'il n'est pas supporté.
Normal		Le mode pleine vitesse est désactivé.
Pause TEMPS		Pause pour un TEMPS donné : une seconde en mode normal est un temps de 1800, et de 4500 en full speed mode. Explication détaillée .
getKey		Renvoie la dernière touche appuyée ou 0 si aucune touche n'est pressée. C'est comme le getkey en BASIC, mais avec des codes touches différents. Explication détaillée .
getKey^r		Attend qu'une touche soit pressée et renvoie le code correspondant. Les codes sont différents de ceux d'un getkey normal. Pour le r :

<code>getKey(TOUCHE)</code>		Renvoie 1 si la TOUCHE indiquée est pressée, sinon renvoie 0. Le code de la touche peut être une constante ou une variable. Et les parenthèses respectivement : { () }
<code>getKey(0)</code>		Renvoie 1 si n'importe quel touche du clavier est pressé, ou sinon renvoie 0.
<code>Asm(HEX)</code>		Exécuter un code assembleur écrit en hexadécimal à cet endroit.
<code>prgmNAME</code>		La liste des programmes se trouve ici : Le code du programme externe est inséré lors de la compilation comme si il faisait partie du programme principal. (Similaires à l'"include" en C++)
<code>#Axiom(NAME)</code> Commande de base : AsmComp()		La librairie assembleur Axiom peut être utilisée dans le programme. Il n'y a pas besoin de mettre de guillemet pour le nom de l'appvars visé.
<code>#Icon(HEX)</code> Commande de base : identity()		Indique au compilateur de remplacer l'icône par défaut par celle indiqué entre les parenthèses. Celle-ci doit être constituée de 64 caractères hexadécimaux.
<code>#Realloc()</code> <code>#Realloc(PTR)</code> Commande de base : Real()		Modifie le buffer des variables (A, B, C,...) pour le pointer à un nouvel endroit (54 octets nécessaires). Si aucun argument n'est précisé, l'emplacement par défaut est utilisé.

Ecran et mémoire tampon

Commandes	Touches correspondantes	Description
<code>DispGraph</code> <code>DispGraph(BUF)</code>		Affiche le buffer ou tout autre buffer spécifique sur l'écran. Explication détaillée.
<code>DispGraphClrDraw</code> <code>DispGraphClrDraw(BUF)</code>		Fait la même chose que <code>DispGraph:ClrDraw</code> mais en plus rapide.
<code>DispGraph^r</code> <code>DispGraph(BUF1,BUF2)^r</code>		Affiche successivement le buffer et le back-buffer ou d'autres buffers spécifiques sur l'écran. On voit apparaître 3 niveaux de gris. Explication détaillée.
<code>DispGraphClrDraw^r</code> <code>DispGraphClrDraw(BUF)^r</code>		Fait la même chose que <code>DispGraph^r:ClrDraw^r</code> mais en plus rapide.
		

<i>DispGraph rr</i> <i>DispGraph(BUF1,BUF2) rr</i>		Affiche successivement le buffer et le back-buffer ou tout autres buffer spécifiques sur l'écran. On voit apparaître 4 niveaux de gris à l'écran.
<i>DispGraphClrDraw rr</i> <i>DispGraphClrDraw(BUF1,BUF2) r</i>		Fait la même chose que <i>DispGraph rr</i> mais en plus rapide. Et pour le r :
<i>ClrHome</i>		Efface l'écran et remet la position du curseur en haut à gauche.
<i>ClrDraw</i> <i>ClrDraw r</i> <i>ClrDraw(BUF)</i>		Efface en blanc respectivement le buffer, le back-buffer ou tout autre buffer spécifique. Et pour le r :
<i>ClrDraw rr</i>		Efface en blanc le buffer et le back-buffer.
<i>DrawInv</i> <i>DrawInv r</i> <i>DrawInv(BUF)</i>		Inverse les pixels respectivement du buffer, back-buffer ou de tout autre buffer spécifique. Et pour le r :
<i>StoreGDB</i>		Copie le vrai écran sur le buffer.
<i>StorePic</i>		Copie le buffer sur le back-buffer.
<i>RecallPic</i>		Copie le back-buffer sur le buffer.
<i>Horizontal +</i> <i>Horizontal + r</i> <i>Horizontal +(BUF)</i>		Le buffer, le back-buffer, ou tout autre buffer spécifique est décalé vers la droite de 1 pixel. Les pixels blancs sont décalés également. Et pour le r :
<i>Horizontal -</i>		Le buffer, le back-buffer, ou tout autre buffer

Horizontal - r Horizontal -(BUF)		spécifique est décalé vers la gauche de 1 pixel. Les pixels blancs sont décalés également.
Vertical + Vertical + r Vertical +(BUF)		Le buffer, le back-buffer, ou tout autre buffer spécifique est décalé vers le bas de 1 pixel. Les pixels blancs sont décalés également.
Vertical - Vertical - r Vertical -(BUF)		Le buffer, le back-buffer, ou tout autre buffer spécifique est décalé vers le haut de 1 pixel. Les pixels blancs sont décalés également.
Shade()		Renvoie le contraste actuel de la calculatrice (entre 0 le plus clair et 63 le plus foncé).
Shade(EXP)		Change le contraste. 0 est le le plus clair, et 63 le plus sombre.

Blocs de contrôle

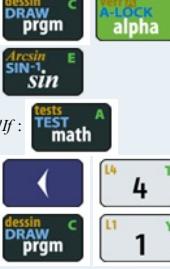
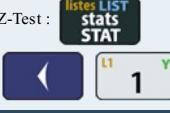
Commandes	Touches correspondantes	Description
Code : Axe End!If EXP		Dans une boucle, si la condition est fausse, ce code se comporte comme un <i>End</i> classique. Si l'expression est vraie, le programme sort de la boucle
Code : Axe EndIf EXP		Dans une boucle, si l'expression est vraie, ce code se comporte comme un <i>End</i> classique. Si l'expression est fausse, le programme sort de la boucle
Code : Axe Else!If EXP		Utilisé avant ou à la place d'un <i>Else</i> : si l'expression est fausse, le code suivant sera exécutée, sinon il est sauté.

Code : Axe <pre>ElseIf EXP</pre>	 <p>If :</p> <p>Utilisé avant ou à la place d'un <i>Else</i> : si l'expression est vraie, le code suivant est exécuté, sinon il est sauté.</p>
Code : Axe <pre>If EXP code End</pre>	 <p>If :</p> <p>Si l'expression est vraie, le code sera exécuté.</p>
Code : Axe <pre>If EXP code 1 Else code 2 End</pre>	 <p>If :</p> <p>Si l'expression EXP est vraie, le code 1 sera exécuté. Sinon, c'est le code 2 uniquement qui le sera.</p>
Code : Axe <pre>!If EXP code End</pre>	 <p>If :</p> <p>Si l'expression EXP est fausse, le code sera exécuté.</p>
Code : Axe <pre>!If EXP code 1 Else code 2 End</pre>	 <p>If :</p> <p>Si l'expression EXP est fausse, le code 1 sera exécuté. Sinon, seul le code 2 sera exécuté.</p>
Code : Axe <pre>While EXP code End</pre>	 <p>While :</p> <p>Tant que l'expression EXP est vraie le code sera exécuté. Il ne le sera plus dès que EXP sera devenu fausse.</p>
Code : Axe <pre>Repeat :</pre>	 <p>Repeat :</p> <p>Cette instruction est presque identique à <i>While</i>. La seule</p>

<pre>repeat EXP code End</pre>		<p>La différence est que le code ne sera exécuté uniquement si l'expression EXP est fausse.</p>
<pre>Code : Axe For (VAR, EXP1, EXP2) code End</pre>		<p>La variable VAR est initialisé par l'expression 1 (EXP1). Le code sera exécuté un certain nombre de fois, à chaque fois VAR sera incrémenté (augmenté) de 1. La boucle s'arrête quand la valeur de VAR est supérieure à celle de EXP2.</p>
<pre>Code : Axe For (EXP) code1 End</pre>		<p>Répète code1 exactement EXP fois.</p>
<pre>Code : Axe DS<(VAR, MAX) code1 End</pre>		<p>La variable VAR est décrémentée de 1. Si elle vaut 0, le Code1 est exécuté puis la variable est réinitialisée à sa valeur maximale MAX, sinon le Code1 est ignoré.</p>

Labels et fonctions

Commandes	Touches correspondantes	Description
<i>Lbl LBL</i>		<p>Créé le label LBL à la position actuelle.</p>
<i>Goto LBL</i>		<p>Saute jusqu'au label LBL.</p>
<i>Sub(LBL)</i>		<p>Appeille la fonction. Toutes les fonctions doivent finir par un <i>Return</i>.</p>
<i>Sub(LBL,r1,r2...)</i>		<p>Appeille la fonction en envoyant jusqu'à six paramètres qui seront stockés dans r1, r2, r3, r4, r5 et r6.</p>
<i>Sub(LBL,r,r1,r2...)</i>		<p>Fait pareil qu'au dessus, à la différence que les variables r1, r2, r3, r4, r5 et r6 sont sauvegardées avant l'exécution de la fonction, et qu'elles sont restaurées après.</p>

<i>LBL(r1,r2...)</i>		Équivalent à <i>Sub(LBL,r1,r2...)</i> avec une syntaxe plus courte.
<i>(EXP)(r1,r2...)</i>		Appelle une fonction définie par une expression entre parenthèses en y passant les arguments entre les deuxièmes parenthèses. Équivalent à <i>Sub(LBL,r1,r2...)</i> : Lbl <i>LBL</i> : <i>EXP</i> .
<i>Return</i>		Termine une fonction. Si ce n'est pas dans une fonction, le programme s'arrêtera.
<i>Return^r</i>		Et pour le <i>r</i> : Quitte le programme d'urgence.
<i>LLBL</i>		Renvoie l'adresse du label.
<i>ReturnIf EXP</i>		Revient uniquement si l'expression est vraie.
<i>Return!If EXP</i>		Revient uniquement si l'expression est fausse.
<i>λ(EXP)</i> Commande de base : <i>log</i>	$\lambda : \frac{10^x}{\log}$	Renvoie l'adresse d'une fonction anonyme. Pour plus d'explications, cf. le chapitre "Maîtriser les fonctions" de la partie 3.
<i>Z-Test(EXP,LBL1,LBL2...)</i>	<i>Z-Test</i> : 	Résous l'expression. Si le résultat est 0, le programme saute au premier label <i>LBL1</i> , s'il est 1, le programme saute au deuxième label <i>LBL2</i> , etc. Si le résultat est plus grand que le nombre de labels, le programme saute la commande.

Math (base)

Commandes	Touches correspondantes	Description
<i>VAR</i>	Il faut appuyer sur  pour	Renvoie la valeur de la variable. Les majuscules de A à Z et théta (Θ) sont des variables.
<i>VAR^r</i>	Pour le <i>r</i> : 	Renvoie les 8 premiers bits de la variable (de 0 à 255).
<i>°VAR</i>	Pour le $^\circ$: 	Renvoie l'adresse de la variable dans la mémoire.
<i>EXP→VAR</i>	$\rightarrow : \frac{\text{rappel } X}{\text{sto+}}$	Stocke l'expression <i>EXP</i> dans la variable <i>VAR</i> .
<i>EXP→VAR^r</i>	$\rightarrow : \frac{\text{rappel } X}{\text{sto+}} \text{ et } r : \frac{2\text{nde}}{2\text{ND}}$ 	Stocke l'expression <i>EXP</i> dans les 8 premiers bits de la variable.
		

'CHAR'		Convertit un caractère ASCII en un entier.
-EXP		Renvoie l'opposé de l'expression. C'est le signe négatif, pas le signe moins!
EXP1+EXP2 EXP1-EXP2		Expression2 est additionnée ou soustraite de expression1.
EXP++ EXP--		L'expression (variable ou adresse) est augmentée ou diminuée de 1.
EXP1*EXP2 EXP1/EXP2 EXP1^EXP2		EXP1 est multipliée, divisée, ou le modulo de EXP2.
EXP ²		L'expression est multipliée par elle-même.
EXP1=EXP2 EXP1<EXP2 EXP1<EXP2 EXP1≤EXP2 EXP1>EXP2 EXP1≥EXP2	Les symboles tests sont disponibles dans le menu : tests A math	Renvoie 1 si l'inégalité est vraie ou 0 si elle est fausse. C'est une comparaison non signée.
EXP1 or EXP2 EXP1 and EXP2 EXP1 xor EXP2	Les opérateurs logiques sont disponibles dans le menu : tests A math	Renvoie l'opération logique sur les 8 bits inférieurs des expressions. Une parenthèse doit parfois être utilisée sur le second argument lorsqu'il est utilisé.
abs(EXP)		Renvoie la valeur absolue de l'expression.
√(EXP)		Renvoie la racine carrée de l'expression.
sin(EXP)		Renvoie le sinus de l'expression. Une Période est de [0,256] et la valeur renvoyée varie de -127 à 127.
cos(EXP)		Renvoie le cosinus de l'expression. Une Période est de [0,256] et la valeur renvoyée varie de -127 à 127.
tan ⁻¹ (DX,DY)		Renvoie l'angle d'un point de coordonnées (DX;DY) par rapport à l'axe X. Une période est de [0;256] et DX et DY doivent appartenir à [-512;512]
e^(EXP1)		Renvoie 2 à la puissance de l'expression.
ln(EXP1)		Renvoie le logarithme base 2 de l'expression.
min(EXP1,EXP2)		Renvoie le minimum de 2 expressions.
max(EXP1,EXP2)		Renvoie le maximum de 2 expressions.
rand		Renvoie un nombre à 16 bits aléatoire (de 0 à 65535).
COND?EXP	Pour le ? : rép ANS ? (-)	Si la condition est vraie, l'expression est évaluée.
COND?EXP1,EXP2	Pour le ? : rép ANS ? (-)	Si la condition est vraie, l'expression 1 est évaluée. Si elle est fausse, c'est la 2 qui est évaluée.
COND??EXP	Pour le ? : rép ANS ? (-)	Si la condition est fausse, l'expression est évaluée.

COND? ?EXP1,EXP2	Pour le ? :  	Si la condition est fausse, l'expression 1 est évaluée. Si elle est vraie, c'est l'expression 2 qui est évaluée.
-----------------------------------	---	--

Math (avancé)

Commandes	Touches correspondantes	Description
E^{HEX}	Le E :    	Convertit un nombre hexadécimal en un nombre entier décimal.
b^{BIN}	  	Convertit un nombre binaire en un nombre entier décimal. Il faut avoir activé les minuscules.
T^{TOKEN}	      	Convertit un Token (un ou deux octets) en un nombre entier décimal.
INT.DEC	 	Convertit le <u>nombre constant</u> décimal d'un octet entier INT et de trois chiffres décimaux DEC en un nombre de deux octet. Cela ne marche qu'avec 3 chiffres après la virgule maximum (entre 0 et 999). Attention, le nombre décimal est tronqué et non arrondi !
$\text{EXP1} << \text{EXP2}$ $\text{EXP1} \ll \text{EXP2}$ $\text{EXP1} >> \text{EXP2}$ $\text{EXP1} \gg \text{EXP2}$	Les symboles tests sont disponibles dans le menu :  	Ce sont les comparaisons signés pour les nombres pouvant être soit positif, soit négatif. Retourne 1 si c'est vrai, 0 si c'est faux.
$\text{EXP1} ** \text{EXP2}$	  	C'est la multiplication signé d'un nombre décimal.
$\text{EXP1} *^ \text{EXP2}$	   	Renvoie 1 si EXP1 est plus petit que EXP2 , marche pour les nombres de 2 octets.
$\text{EXP1} // \text{EXP2}$	 	Effectue une division, mais marche pour les nombres signés.
$\text{EXP1} \cdot \text{EXP2}$ $\text{EXP1} + \text{EXP2}$ $\text{EXP1} \square \text{EXP2}$	   	Renvoie le résultat sur 2 octets respectivement d'un "et", "ou inclusif" et "ou exclusif" appliqué sur les bits des deux expressions.
$\text{not}(\text{EXP})$	    	Renvoie l'inverse de tous les bits d'un nombre d'un octet.
$\text{not}(\text{EXP})^r$	    	Renvoie l'inverse de tous les bits d'un nombre de deux octet. Et pour le r :

$\text{EXP1} \times \text{EXP2}$		Renvoie le bit de poids EXP2 de EXP1 . EXP1 doit faire 1 octet. A noter qu'en Axe, le bit de poids le plus fort est noté 0, et celui de poids le plus faible est numéroté 7.
$\text{EXP1} \times \text{ee} \text{EXP2}$		Idem que ci-dessus, EXP1 faisant ici 2 octets.
$\sqrt{(\text{EXP})^r}$		Et pour le r : Renvoie la racine d'un nombre décimal.
EXP^{-1}		Renvoie l'inverse d'un nombre décimal.

Dessin

Commandes	Touches correspondantes	Description
$\text{Pixel-On}(\text{X}, \text{Y})$ $\text{Pixel-On}(\text{X}, \text{Y})^r$ $\text{Pixel-On}(\text{X}, \text{Y}, \text{BUF})$		Un pixel devient noir à l'emplacement (X, Y) respectivement du buffer, du back-buffer ou de tout autre buffer spécifique.
$\text{Pixel-Off}(\text{X}, \text{Y})$ $\text{Pixel-Off}(\text{X}, \text{Y})^r$ $\text{Pixel-Off}(\text{X}, \text{Y}, \text{BUF})$		Un pixel devient blanc à l'emplacement (X, Y) respectivement du buffer, du back-buffer ou de tout autre buffer spécifique.

$Pxl-Change(X,Y)$ $Pxl-Change(X,Y) \text{ } r$ $Pxl-Change(X,Y,BUF)$	<p>Et pour le r :</p>	<p>Un pixel est inversé à l'emplacement (X,Y) respectivement du buffer, du back-buffer ou de tout autre buffer spécifique.</p>
$pxl-Test(X,Y)$ $pxl-Test(X,Y) \text{ } r$ $pxl-Test(X,Y,BUF)$	<p>Et pour le r :</p>	<p>Un pixel est testé à l'emplacement (X,Y) respectivement du buffer, du back-buffer ou de tout autre buffer spécifique. La commande renvoie 1 si il est noir ou 0 si il est blanc.</p>
$Line(X1,Y1,X2,Y2)$ $Line(X1,Y1,X2,Y2) \text{ } r$ $Line(X1,Y1,X2,Y2,BUF)$	<p>Et pour le r :</p>	<p>Dessine une ligne noir partant du point $(X1,Y1)$ et allant jusqu'au point $(X2,Y2)$ respectivement dans le buffer, le back-buffer ou de tout autre buffer spécifique.</p>
$Rect(X,Y,W,H)$ $Rect(X,Y,W,H) \text{ } r$ $Rect(X,Y,W,H,BUF)$	<p>Et pour le r :</p>	<p>Dessine un rectangle plein dont l'angle haut-gauche est le point (X,Y). W est la largeur (axe X) et H la hauteur (axe Y). (Respectivement dans le buffer, le back-buffer ou de tout autre buffer spécifique)</p>
$RectI(X,Y,W,H)$ $RectI(X,Y,W,H) \text{ } r$ $RectI(X,Y,W,H,BUF)$	<p>Et pour le r :</p>	<p>Inverse les pixels d'un rectangle plein dont l'angle haut-gauche est le point (X,Y). W est la largeur (axe X) et H la hauteur (axe Y). (Respectivement dans le buffer, le back-buffer ou de tout autre buffer spécifique)</p>

	L3 6 3	
Circle(X,Y,R) Circle(X,Y,R) r Circle(X,Y,R,BUF)	2nde 2ND dessin C prgm W R Q W 9 Et pour le r : 2nde 2ND angle B apps L3 6 3	Dessine un cercle de centre (X,Y) et dont le rayon est de R (en pixels) dans le buffer.

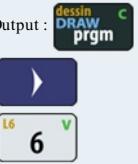
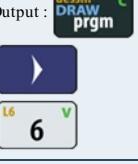
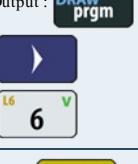
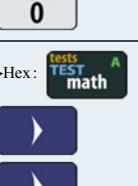
Sprites

Commandes	Touches correspondantes	Description
Pt-On(X,Y,PIC) Pt-On(X,Y,PIC) r Pt-On(X,Y,PIC,BUF)	Pt-On : 2nde 2ND dessin C prgm L1 Y 1 Et pour le r : 2nde 2ND angle B apps L3 6 3	Le sprite 8*8 pixels PIC est dessiné respectivement sur le buffer, sur le back-buffer, ou sur le buffer BUF spécifié, aux coordonnées (X,Y), mais n'efface pas la zone en dessous de lui.
Pt-Off(X,Y,PIC) Pt-Off(X,Y,PIC) r Pt-Off(X,Y,PIC,BUF)	Pt-Off : 2nde 2ND dessin C prgm L2 Z 2 Et pour le r : 2nde 2ND angle B apps L3 6 3	Le sprite 8*8 pixels PIC est dessiné respectivement sur le buffer, sur le back-buffer, ou sur le buffer BUF spécifié aux coordonnées (X,Y), et efface la zone en dessous de lui au préalable.
Pt-Change(X,Y,PIC) Pt-Change(X,Y,PIC) r Pt-Change(X,Y,PIC,BUF)	Pt-Change : 2nde 2ND dessin C prgm L3 6 3 Et pour le r : 2nde 2ND angle B apps L3 6 3	Le sprite 8*8 pixels PIC est dessiné respectivement sur le buffer, sur le back-buffer, ou sur le buffer BUF spécifié aux coordonnées (X,Y), en inversant ses pixels (le noir devient blanc et vice-versa).
Pt-Mask(X,Y,PIC) Commande de base : Plot1()	2nde 2ND graph stats F1 f(x) Y= L1 Y 1	Le sprite PIC 8*8 pixels en niveau de gris (2 layers) est dessiné sur les 2 buffers aux coordonnées (X,Y). Les zones vides (pixels éteints) sur les deux layers sont transparentes et les autres combinaisons génèrent 3 niveaux de gris.
Pt-Mask(X,Y,PIC) r Commande de base : Plot1()	2nde 2ND graph stats F1 f(x) Y= L1 Y 1	Le sprite PIC 8*8 pixels en niveau de gris (2 layers) est dessiné sur le buffer aux coordonnées (X,Y). Les zones vides (pixels éteints) sur les deux layers sont transparentes et les autres combinaisons sont noir, blanc et inversé.
pt-Get(X,Y) pt-Get(X,Y) r pt-Get(X,Y,BUF) Commande de base : plot2()	2nde 2ND graph stats F1 f(x) Y= L2 Z 2	Renvoie un pointeur temporaire sur une copie du sprite 8*8 situé respectivement aux coordonnées (X,Y) du buffer, du back-buffer, ou du buffer BUF spécifié.
pt-Get(X,Y,BUF,TEMP) Commande de base : plot2()	2nde 2ND graph stats F1 f(x) Y= L2 Z 2	Copie le sprite 8*8 situé aux coordonnées (X,Y) du buffer BUF vers le buffer temporaire TEMP de 8 octets. La fonction retourne TEMP.
Bitmap(X,Y,BMP) Bitmap(X,Y,BMP) r Bitmap(X,Y,BMP,BUF) Bitmap(X,Y,BMP,BUF,MODE) Commande de base : Tangent()	2nde 2ND dessin C prgm L5 U 5	Dessine une bitmap respectivement sur le buffer, le back-buffer, ou le buffer BUF spécifié à l'emplacement X,Y. Le paramètre BMP doit être une PIC déclarée de cette façon : :Data (hauteur, largeur)→Ptr :[spriteEnHexa] :Bitmap (X,Y,Ptr) L'argument MODE peut être 0 (les pixels seront affichés)

		comme avec <i>Pt-On</i>) ou 1 (comme <i>Pt-Change</i>). 0 est utilisé par défaut.
<i>rotC(PIC)</i> Commande de base : ShadeNorm()		Exerce une rotation de 90° à droite (sens des aiguilles d'une montre) à une copie du sprite, et renvoie un pointeur sur cette copie.
<i>rotCC(PIC)</i> Commande de base : Shade_t()		Exerce une rotation de 90° à gauche (sens inverse des aiguilles d'une montre) à une copie du sprite, et renvoie un pointeur sur cette copie.
<i>flipV(PIC)</i> Commande de base : ShadeX2()		Exerce un effet de miroir vertical sur une copie du sprite, et retourne un pointeur sur cette copie.
<i>flipH(PIC)</i> Commande de base : ShadeF()		Exerce un effet de miroir horizontal sur une copie du sprite, et retourne un pointeur sur cette copie.

Texte

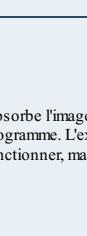
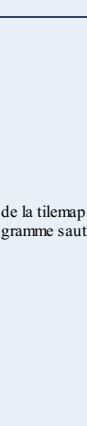
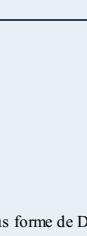
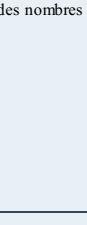
Commandes	Touches correspondantes	Description
<i>Disp PTR</i>	Disp : 	Affiche la chaîne pointée à l'emplacement du curseur, et déplace le curseur à la fin de la chaîne. Si le texte atteint le bord de l'écran, il ira à la ligne suivante.
<i>Disp EXP▶Dec</i>	Disp : 	Affiche l'expression comme un nombre décimal à l'emplacement du curseur, et avance le curseur de 5 espaces.
<i>Disp EXP▶Char</i> Commande de base : ▶Frac	Disp : 	Affiche l'expression comme un caractère ASCII à l'emplacement du curseur et avance le curseur d'un espace.
<i>Disp EXP▶Tok</i> Commande de base : ▶DMS	Disp : 	Affiche le token (1 ou 2 octets) correspondant à EXP, et bouge le curseur de la taille en lettres du token.
<i>Disp □</i>	Disp : 	Bouge le curseur à la ligne suivante.

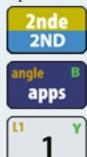
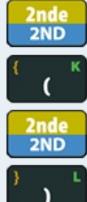
<i>Output(X)</i>	Output : dessin DRAW prgm 	Bouge le curseur à la position X divisé par 256, X modulo 256.
<i>Output(X,Y)</i>	Output : dessin DRAW prgm 	Bouge le curseur à la position (X,Y) .
<i>Output(X,Y,...)</i>	Output : dessin DRAW prgm 	Bouge le curseur à la position (X,Y) et y affiche comme texte les arguments suivants.
<i>Text EXP</i>	Text : 2nde 2ND dessin DRAW prgm 	Écris EXP sous forme de texte (affecté par les options du texte) à l'emplacement actuel du curseur texte (différent du curseur classique, voir la commande Fix pour les options du texte).
<i>Text EXP▶Dec</i>	Text : 2nde 2ND dessin DRAW prgm 	Écris EXP sous forme de nombre(s) décim(al/aux) (affecté(s) par les options du texte) à l'emplacement actuel du curseur texte (différent du curseur classique, voir la commande Fix pour les options du texte).
<i>Text EXP▶Char</i> Commande de base : ▶Frac	Text : 2nde 2ND dessin DRAW prgm 	Écris EXP sous forme de caractère(s) ASCII (affecté(s) par les options du texte) à l'emplacement actuel du curseur texte (différent du curseur classique, voir la commande Fix pour les options du texte).
<i>Text PTR▶Tok</i> Commande de base : ▶DMS	Text : 2nde 2ND dessin DRAW prgm 	Écris le nombre pointé par PTR sous forme de token(s) (affecté(s) par les options du texte) à l'emplacement actuel du curseur texte (différent du curseur classique, voir la commande Fix pour les options du texte). Passer directement un nombre en argument ne fournira pas le résultat attendu.
<i>Text(X)</i>	Text(: 2nde 2ND dessin DRAW prgm 	Bouge le curseur texte à l'emplacement $(X \text{ modulo } 256, X \text{ divisé par } 256)$.
<i>Text(X,Y)</i>	Text(: 2nde 2ND dessin DRAW prgm 	Bouge le curseur à l'emplacement (X,Y) .
<i>Text(X,Y,...)</i>	Text(: 2nde 2ND dessin DRAW prgm 	Bouge le curseur à l'emplacement (X,Y) et y affiche le reste des arguments sous forme de texte.
<i>EXP▶Hex</i> Commande de base : ▶Rect	►Hex: TEST math 	Résout l'expression, convertit le résultat en hexadécimal et renvoie un pointeur sur cette valeur hexadécimale.

		
Fix 0		Active la petite taille de police. La calculatrice ne remet pas la taille normale en fin de programme !
Fix 1		et 2 fois Active la grande taille de police (c'est celle utilisée par défaut).
Fix 2		et 3 fois Active la coloration normale (blanc) de la police. La calculatrice ne remet pas la couleur normale en fin de programme !
Fix 3		et 4 fois Active la coloration inversée (noir, c'est la coloration par défaut).
Fix 4		et 5 fois Le texte est écrit directement sur l'écran. La calculatrice ne change pas d'écran à la fin du programme !
Fix 5		et 6 fois Le texte est écrit sur le buffer (écran d'écriture par défaut de la calculatrice).
Fix 6		et 7 fois Avec 7 fois, il active le scrolling si on essaye d'afficher du texte tout en bas en dehors de l'écran. La calculatrice ne désactive pas automatiquement le scrolling à la fin d'un programme !
Fix 7		et 8 fois Désactive le scrolling automatique (par défaut).
		

Fix 8	 	Désactive le mode Minuscules.
Fix 9		Active le mode Minuscule.

Data et stockage

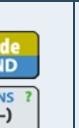
Commandes	Touches correspondantes	Description
"STR"		Ajoute la chaîne de caractère dans la mémoire du programme.
[HEX]		Ajoute le nombre hexadécimal dans la mémoire du programme.
[OSVAR]		Absorbe l'image, l'appVar, le programme ou la chaîne dans la mémoire du programme. L'exécutable n'aura pas besoin de cette variable pour fonctionner, mais elle devra exister pour compiler.
[PICVAR ^r]		Absorbe l'image de la tilemap dans la mémoire du programme. Au bout de 12 tiles, le programme saute une ligne. L'exécutible n'aura pas besoin de la Pic.
Data(^{NUM1,NUM2} ^r ,...) Commande de base : ΔList()	Data :  Et pour le r : 	Ajoute les octets spécifiés sous forme de Data dans le programme. Les nombres terminés par ^r sont ajoutés comme des nombres 16 bits.
	Buff : 	

<code>Buff(SIZE)</code> <code>Buff(SIZE,CONST)</code> Commande de base : <code>det()</code>		Créé un buffer de <code>SIZE</code> octets dans la mémoire du programme, rempli par la valeur <code>CONST</code> ou 0 si non spécifiée.
<code>CONST→°NAME</code>	 Et pour le ° : 	Remplace le pointeur statique ° <code>NAME</code> par la constante <code>CONST</code> lors de la compilation.
<code>EXP→NAME</code>		Stocke l'expression <code>EXP</code> dans la variable <code>NAME</code> . <code>NAME</code> doit être défini avec la commande ci-dessus en indiquant son pointeur en constante.
<code>NAME</code>	Toutes les variables	Retourne la variable <code>NAME</code> .
<code>L₁</code> <code>L₂</code> <code>L₃</code> <code>L₄</code> <code>L₅</code> <code>L₆</code>	 à	Retourne un pointeur sur un endroit libre de la mémoire. <code>L₁</code> = 714 octets (saveSScreen) Volatilité: BASSE <code>L₂</code> = 531 octets (statVars) Volatilité: BASSE (ne pas utiliser cette zone quand vous utilisez les interruptions) <code>L₃</code> = 768 octets (appBackUpScreen) Volatilité: MOYENNE (c'est le back-buffer) <code>L₄</code> = 256 octets (tempSwapArea) Volatilité: MOYENNE (ne pas utiliser en même temps que les commandes Archive et Unarchive) <code>L₅</code> = 128 octets (textShadow) Volatilité: MOYENNE (ne pas utiliser en même temps que "Disp", "Output" et "ClrHome") <code>L₆</code> = 768 octets (plotSScreen) Volatilité: HAUTE (c'est le buffer)
<code>{EXP}</code>		Retourne le premier octet de l'expression pointée (de 0 à 255).
<code>{EXP}^r</code>	 Et pour le ^r : 	Retourne les deux octets de l'expression pointée (de 0 à 65535).
<code>{EXP}^{rr}</code>	 Et pour le ^{rr} : 	Retourne les deux octets de l'expression pointée, mais inversés. Exemple, si <code>EXP</code> pointe sur 1111111000000000, cette instruction renverra 00000001111111.
		<small>Le premier octet de <code>EXP1</code> est stocké dans l'octet pointé par <code>EXP2</code>.</small>

<code>EXP1→{EXP2}^r</code>		Le premier octet de <code>EXP1</code> est stocké dans l'octet pointé par <code>EXP2</code> .
<code>EXP1→{EXP2}^rr</code>	<p>Et pour le <code>r</code> :</p>	Les deux octets de <code>EXP1</code> sont stockés dans les deux octets pointés par <code>EXP2</code> .
<code>EXP1→{EXP2}^rrr</code>	<p>Et pour le <code>r</code> :</p>	Les deux octets de <code>EXP1</code> sont stockés dans les deux octets pointés par <code>EXP2</code> , mais inversés. Donc, le premier octet de <code>EXP1</code> est stocké dans le deuxième pointé par <code>EXP2</code> et le deuxième octet de <code>EXP1</code> dans le premier pointé par <code>EXP2</code> .
<code>sign{EXP}</code> Commande de base : int()		Retourne le premier octet pointé par <code>EXP</code> (il sera de -128 à 127).
<code>nib{EXP}</code> Commande de base : ipart()		Renvoie le <code>EXP</code> ième quartet de la RAM (demi-octet, entre 0000 et 1111 ou entre 0 et 15). Comme ce sont des quartets, il y en a deux fois plus que des octets, passez donc en argument les pointeurs multipliés par deux.
<code>nib{EXP}^r</code> Commande de base : ipart()	<p>Et pour le <code>r</code> :</p>	Renvoie le <code>EXP</code> ième quartet de la RAM (demi-octet, entre 0000 et 1111 ou entre 0 et 15). Comme ce sont des quartets, il y en a deux fois plus que des octets, passez donc en argument PTR*2.
<code>EXP1→nib{EXP2}</code> Commande de base : ipart()		Stocke <code>EXP1</code> dans le <code>EXP2</code> ième quartet de la RAM. Même remarque sur les quartets.
<code>Fill(PTR,SIZE)</code> <small>Fill(PTR,SIZE,NUM)</small>		Remplit les <code>SIZE</code> octets qui suivent le pointeur <code>PTR</code> par l'octet pointé à cet emplacement. Si <code>NUM</code> est spécifié, remplit <code>SIZE</code> octets à partir de

<code>Put(PTR,SIZE,NUM)</code>		PTR par la valeur NUM.
<code>Copy(PTR1,PTR2,SIZE)</code> Commande de base : <code>conj()</code>		Copie les SIZE premiers octets de PTR1 sont copiés au début des octets pointés par PTR2 (avec SIZE différent de 0).
<code>Copy(PTR1,PTR2,SIZE)</code> Commande de base : <code>conj()</code>		Copie les SIZE derniers octets de PTR1 au début des octets pointés par PTR2 (avec SIZE différent de 0) Et pour le r : 2nde 2ND angle B apps L3 θ 3
<code>Exch(PTR1,PTR2,SIZE)</code> Commande de base : <code>expr()</code>		Les SIZE octets au début de PTR1 sont échangés avec les SIZE octets à partir de PTR2 (avec SIZE différent de 0) et 4 fois puis précéd résol entrer

Variables externes

Commandes	Touches correspondantes	Description
<code>Ans</code>		Retourne la valeur de la variable d'OS Ans comme entier. Ans n'est en aucun cas modifié par un programme Axe sauf avec la commande ci-dessous. Renvoie une erreur si la valeur est hors limites.
<code>EXP→Ans</code>		Stocke l'expression dans la variable d'OS Ans.
<code>GetCalc(PTR)</code>		Renvoie un pointeur sur la variable d'OS (Pic, appvar etc) dont le nom est pointé, ou 0 s'il est archivé ou s'il n'a pas été trouvé.
<code>GetCalc(PTR,FILE)</code>		Créé un fichier nommé FILE avec le contenu de la variable d'OS (Pic, appvar etc) dont le nom est pointé, et ce même s'il est archivé. Retourne 0 si la variable n'a pas été trouvée ou n'importe quelle autre valeur en cas de succès.
<code>GetCalc(PTR,SIZE)</code>		Créé une variable d'OS avec le nom pointé et la taille spécifiée. Retourne l'adresse de départ de la variable d'OS ou 0 s'il n'y a pas assez de RAM libre. Écrase tout autre variable d'OS déjà appelée de ce nom, même si elle est archivée.
		

<i>Archive PTR</i>		Essayes d'archiver la variable dont le nom est pointé. Renvoie 1 en cas d'échec et 0 en cas de succès. Si la mémoire Flash disponible est insuffisante, renvoie ERR:MEM.
<i>Unarchive PTR</i>		Essayes de dés-archiver la variable d'OS dont le nom est pointé. Renvoie 1 si l'opération a échouée et 0 si elle a réussi. S'il n'y a plus assez de place pour dés-archiver, renvoie une ERR:MEM.
<i>DelVar PTR</i>		Suprimes la variable d'OS dont le nom est pointé. Rien ne se passe si la variable n'est pas trouvée. Ne pas utiliser pour supprimer les variables de programme !
<i>input</i>		Attend l'entrée d'une chaîne et l'appui sur Entrée. Renvoie un pointeur sur cette chaîne. C'est une chaîne de tokens, pas de caractères !
<i>float{PTR}</i> Commande de base : fPart()		Convertit le nombre flottant pointé en un entier. Les flottants pèsent 9 octets.
<i>EXP→float{PTR}</i> Commande de base : fPart()		Convertit l'expression en un flottant et la stocke à l'adresse pointée. Les flottants pèsent 9 octets.

Interruptions

Commandes	Touches correspondantes	Description
<i>FnInt(LBL,FREQ)</i>		Initialise les interruptions avec la fonction LBL, à une vitesse correspondant à FREQ : 0 (le plus rapide), 2, 4 ou 6 (le plus lent).
<i>FnOn</i>		Active les interruptions.
<i>FnOff</i>		Désactive les interruptions.
<i>Stop</i>		Suspend l'exécution du programme jusqu'à la prochaine interruption. Si les interruptions ne sont pas activées et que vous appelez quand même cette fonction, la calculatrice freeze !
<i>LnReg</i>		Réinitialise le mode d'interruption par défaut de la calculatrice. Cette fonction doit OBLIGATOIREEMENT être appelée en fin de programme si les interruptions sont utilisées.

--	--	--

Port de liaison

Commandes	Touches correspondantes	Description
<i>port</i> Commande de base : <i>ClrTable</i>		Renvoie l'état actuel du port de liaison traduit par un nombre entre 0 et 3 (4 états possibles).
<i>EXP→port</i> Commande de base : <i>ClrTable</i>		Assigné le statut <i>EXP</i> (de 0 à 3) au port liaison. Le programme doit obligatoirement se terminer avec le port au statut 0 si modifié !
<i>Freq(<i>ONDE,TEMPS</i>)</i> Commande de base : <i>SinReg</i>		Un son est joué à partir du port jack. L'onde doit être comprise entre 1-255 (inversion proportionnelle à la fréquence) et peut être calculé comme suit : 32768/fréquenceNote. Le temps est compté en microseconde. Ainsi, pour jouer un La pendant 500 microsecondes on peut écrire <i>Freq(32768/440,500)</i> .
<i>Virgule :</i>		
<i>Send(<i>BYTE,TIME</i>)</i>		Essayé d'envoyer un octet (comme une variable statique) par le port de liaison pendant le temps spécifié. L'octet est envoyé jusqu'à ce que l'autre calculatrice le reçoive ou jusqu'à ce que le temps (de l'ordre des microsecondes) soit écoulé. Retourne 1 si l'envoi est un succès ou 0 si le temps est écoulé.
<i>Get</i>		Vérifie si l'autre calculatrice essayé d'envoyer quelque chose. Retourne l'octet reçu ou -1 si rien n'est envoyé. Aucune attente n'est faite, donc il est conseillé d'utiliser cette commande dans une boucle.

Autres

Commandes	Touches correspondantes	Description
<i>Select(<i>EXP1,EXP2</i>)</i>		Renvoie la valeur de l'expression 1 avant d'avoir reçu les modifications de l'expression 2. Utile surtout pour l'optimisation.
<i>length(<i>PTR</i>)</i>		Renvoie le nombre d'octets depuis le début du pointeur jusqu'à la prochaine data 0.
<i>inData(<i>OCTET,PTR</i>)</i> Commande de base : <i>inString()</i>		Recherche <i>OCTET</i> dans la data pointée. S'il est trouvé, sa position dans la data est renvoyée (en partant de 1), s'il n'est pas trouvé, 0 est renvoyé.
<i>Equ▶String(<i>STR1,STR2</i>)</i>		Vérifie si <i>STR1</i> et <i>STR2</i> (deux chaînes terminées par 0) sont égales. Si oui, renvoie 0, si non renvoie n'importe quelle autre valeur.

<code>SortD(PTR,SIZE)</code>		Trie SIZE octets (de 1 à 256) à partir de l'adresse pointée du plus grand au plus petit.
<code>cumSum(PTR,SIZE)</code>		Calcule la somme cumulée de SIZE octets à partir de l'adresse pointée.

J'essaierai de mettre à jour ce tableau régulièrement en fonction des mises à jour (version actuelle : v1.1.2).

Vous ne connaîtrez peut être jamais cette annexe par cœur, donc n'hésitez pas à la consulter régulièrement.

Ce cours n'est évidemment pas terminé. Pour suivre l'évolution du plan, je vous conseille d'aller jeter un coup d'œil [ici](#). De plus, j'essaierai de la mettre à jour au fur et à mesure des nouvelles version de l'Axe Parser.

Je vous demanderai juste de laisser un petit commentaire pour dire ce que vous pensez de ce tutoriel, les améliorations possibles, les points que vous voudriez voir plus clairement, combien de temps vous avez mis pour lire chaque chapitre...

Voici quelques liens pour faire partager vos projets ou poser des questions :

- Chez [omnimaga](#), la plus grande partie du [forum](#) est anglophone, mais [une partie est réservée pour les français](#).
- Le forum [espace TI](#) organise régulièrement des concours pour que chacun puisse tester ses capacités, et il est essentiellement constitué de programmeurs en Axe.
- Ou tout simplement le [forum du site du zéro](#) 😊 !

Et pour finir je voudrais je dois quelques remerciements :

- A conflict pour avoir validé mon tutoriel.
- A wagwan pour l'aide qu'il apporte dans ce tutoriel.
- A Citor de TI-Planet pour la conception des fameuses touches de calculatrices : [en version intégrale ici](#).
- A gaos pour [la correction](#) les conseils à propos de [mon orthographe](#) ma syntaxe d'écriture. 🍑
- Et bien sûr au forum [omnimaga](#) pour avoir créé et soutenu le projet Axe Parser !