



:R4

Langage de Programmation Orienté Humains

Novembre 2006

Rev. Mars 2008, Mai 2008

**<http://www.reda4.org>
pabloreda@gmail.com**

Manuel Cornes - traduction française (Août 2008)

Ont participé :

**Maria Jose Aguirre
Jose Maria Fantasia
Sebastian Desimone
Javier Gil Chica
Dutra da Lacerda**

Ce logiciel est distribué sous la licence CC-GNU GPL

Introduction

La continuelle apparition sur le marché de technologies destinées aux ordinateurs, avec des équipements toujours plus puissants et économiques, a amené les développeurs, pour des raisons dictées par le marché, à préférer des langages ainsi que d'autres outils de développement qui s'éloignent des principes basiques de la programmation, basés sur l'architecture des microprocesseurs, introduisant par là même une complexité et un coût élevé dû à la productivité et à la difficulté du développement, qui se voient compensées par un accroissement des ressources.

A la fin des années 60, Charles Moore développe le langage Forth, basé sur un paradigme de machine à piles différent de la perspective adoptée par la majorité des langages de programmation. Forth oblige à envisager la construction de systèmes plus comme un processus de synthèse que comme un agencement de composants, en s'éloignant de l'idée de construire des couches virtuelles entre l'ordinateur et le programmeur, en considérant que plutôt que de réduire le problème cela le rend plus difficile encore.

Le langage Forth n'est pas très utilisé en développement logiciel, mais il l'est par contre en hardware : les caractéristiques uniques et la brièveté du code généré sont particulièrement utiles pour la programmation des micro-contrôleurs, mais l'industrie logicielle l'ignore pour des raisons difficiles à définir précisément.

Fin 2005, la recherche d'une alternative au langage C et son cortège de problèmes, à savoir les pratiques acceptées (interdiction des variables globales) et la recherche d'un code optimisé (qui implique l'utilisation de variables globales), ont conduit l'auteur à l'étude du langage Forth.

Au départ, l'intention était de créer un langage en ajoutant des capacités au langage Forth, mais après un temps la perspective changea : le chemin était inverse et il s'agissait de trouver quelque chose de plus simple.

:R4 est un langage inspiré par Colorforth, avec un dictionnaire basique de 109 mots que l'on peut agrandir à souhait, une pile de données, une pile d'adresses et 6 préfixes avec des fonctions distinctes.

Ce manuel est la première documentation du langage. On y trouvera une description de son fonctionnement, le comportement de chaque mot, des exemples et une référence rapide du langage.

Manuel de l'Humain pour apprendre :R4

On utilise un ordinateur lorsque l'on suit les règles qu'il pourvoie, par exemple, on appuie sur le bouton sauver pour sauver.

On programme un ordinateur lorsque l'on crée le comportement de l'ordinateur.

De la même manière que l'être humain communique avec un langage, afin d'indiquer à l'ordinateur ce que l'on souhaite qu'il fasse on utilise aussi un langage. Le langage est le mécanisme qui permet d'exprimer le comportement de l'ordinateur.

Le programme que l'on va écrire sera composé d'un ensemble de mots. Un mot, en :R4, est un ensemble de lettres et symboles séparés par des espaces, par exemple + (signe de la somme) est un mot, la distinction entre les signes et les lettres n'est pas relevante en l'occurrence.

:R4 ne fait pas non plus de différences entre les majuscules et les minuscules lorsque l'on définit ou l'on utilise des mots, l'unique exception est l'usage du caractère " (guillemets) qui, utilisé comme préfixe, permet de définir une chaîne de caractères terminée par un autre caractère ".

Il existe un dictionnaire de base, avec un ensemble de mots qui sont les composants du langage.

Programmer en :R4 consiste en la définition de nouveaux mots à partir du dictionnaire de base afin de pouvoir exprimer le programme que l'on souhaite faire.

On va ainsi définir des mots avec des préfixes pour les variables (#données) et le code (:actions)

Si le préfixe es # (dièse) alors le mot est une DONNÉE, qui a un nom (c'est à dire le mot préfixé par #) et un adresse, c'est à dire l'endroit dans la mémoire qui va enregistrer le contenu.

Par exemple :

#position 3

Si le préfixe est : (symbole deux points) alors le mot est une action, qui a un nom, après le préfixe, et qui a de même une adresse où le code exécuté par la machine est stocké lorsque le mot est utilisé.

Par exemple:

:avancer 1 'position +! ;

Préfixes

Les 6 préfixes qui existent dans le langage sont :

^inclure du vocabulaire

^modulo.txt | inclure le fichier modulo.txt

Le préfixe ^ s'utilise pour inclure un fichier source au sein d'un autre, c'est utile pour créer des mots dans un source et les utiliser dans un autre, de cette manière on évite d'avoir à ré-écrire du code qui sera ré-utilisé dans d'autres fichiers sources, ce qui permet la construction de ce que l'on appellera un ensemble de mots ré-utilisables ou encore un vocabulaire spécifique.

Lorsque l'on inclue ainsi un fichier source dans un autre, tous les mots ne sont pas disponibles, seuls ceux qui sont exportés le sont.

: (deux points) définir une action

:nouveau | définir 'nouveau' comme la liste jusqu'au ';'

::nouveau | définir et exporter cette définition

La définition de mots avec ':' (deux points) permet d'assigner la liste des mots à la suite du mot défini au nom défini par le préfixe.

:: (double deux points) définit le mot exporté, il est utilisable dans tout autre fichier source qui l'inclue avec ^.

(dièse) définit une donnée

#mémoire | définit mémoire comme une variable

#:mémoireg | définit et exporte une variable

La définition de variables avec le préfixe # (dièse) assigne les numéros suivants comme contenu, les crochets "[]" définissent des valeurs de 16 bits et les parenthèses des valeurs de 8 bits, enfin le mot)(permet d'indiquer la quantité d'octets qui seront réservés pour la variable définie.

#: (dièse deux points) exporte la variable.

' (apostrophe) Ajoute sur la pile l'adresse du mot

'hello | empile l'adresse de hello (déjà défini par ailleurs)

| (ligne verticale) Commentaire

|blabla commentaire, ignore tout texte jusqu'à la fin de la ligne

“ (guillemets) Défini un espace mémoire contenant du texte et empile son adresse

"bla bla" | texte (incluant les espaces), l'adresse de la chaîne de caractères est empilée

Le code et la pile de données

Le code est le script qui s'écrit pour programmer l'ordinateur, en :R4 le code est un fichier texte.

Lorsque l'on écrit un nombre dans le code afin qu'il soit traité par la machine, ce numéro sera ajouté sur la pile (empilé). On appelle pile la séquence de nombres qui permet d'ajouter et d'enlever des nombres du haut de la pile.

On débute avec une pile vide et elle se dessine donc sans aucun nombre.

(

Si un nombre apparaît dans le code, par exemple 3, le dessin de la pile devient le suivant :

(3

Après l'ajout d'un autre numéro, par exemple 2, le dessin de la pile devient :

(3 2

L'exemple suivant écrit dans un fichier source est :

```
test.txt
3 2
Produit la pile :
( 3 2
```

Bien que le dessin de la pile et le code source soient quasi identiques vu que seul manque dans le fichier source le dessin de la base de la pile "(", ce sont deux choses bien distinctes en des endroits bien distincts.

Le code source est écrit par le programmeur et la pile est construite par l'ordinateur au moment où le code est évalué. Le programmeur construit la pile indirectement.

Il est nécessaire d'avoir constamment à l'esprit cette relation entre le code et le comportement de la machine. En somme, on construit un script pour que la machine le réalise, et le code correspondant sera le produit de l'ingéniosité du programmeur pour définir le comportement désiré.

Un nombre dans le code résulte en l'ajout de ce numéro au sommet de la pile, lorsque c'est un mot le dictionnaire intervient.

Le dictionnaire est une liste de mots couplés avec leur signification. Lorsque l'on écrit un mot dans le code, ce mot est recherché dans le dictionnaire et si il est trouvé, la signification correspondante est exécutée. Il faut se souvenir que cette

évaluation ou appel d'un mot ne se produit pas lorsque l'on écrit le code mais lorsque l'ordinateur exécute le code.

Dans le cas où le mot n'existe pas dans le dictionnaire il se produit une erreur, tout mot utilisé doit être défini auparavant.

Au début de l'exécution du code, le dictionnaire contient un petit ensemble de mots basiques qui permettent d'utiliser tout ce que sait faire un ordinateur. Ces mots sont classifiés par fonction: manipulation de la pile, arithmétique, logique, mémoire, etc ...

Mots pour la manipulation de la pile

DROP

Retire la valeur en haut de la pile

3 2 drop

(3

puis que 2 est retiré par le drop.

SWAP

Echange la première et la deuxième valeur en haut de la pile

3 2 swap

(2 3

DUP

Duplique le haut de la pile

3 2 dup

(3 2 2

OVER

Ajoute au somme de la pile le deuxième nombre de la pile

3 2 over

(3 2 3

Remarquez ici que le code et la pile ne sont maintenant plus égaux et on doit dire de plus que l'ordinateur évaluera nos mots ou nombres pas à pas, cela signifie que chaque mot agit sur la pile au moment où il est évalué par la machine et à la fin de l'évaluation de tous les mots on obtient la pile indiquée.

Il est important de connaître à chaque instant l'état de la pile, comme indiqué, chaque mot peut produire un changement, il est pratique d'avoir un manière de la représenter et pour cela on utilise un notation de changement de pile.

Cette notation est la suivante :

| avant -- après

On ajoute le symbole | pour indiquer dans le code qu'à partir de ce caractère et jusqu'à la fin de la ligne tout ce que le programmeur a écrit sera considéré comme un commentaire et donc ignoré par la machine, en d'autres termes il s'agit d'un commentaire pour le programmeur.

Nous écrivons dans la partie 'avant' l'état de la pile AVANT d'exécuter le mot suivi de -- et ensuite l'état de la pile APRÈS l'action du mot.

Les valeurs qui sont sur la pile se représentent avec des lettres et la quantité de lettres dans les sections avant et après va dépendre de l'action du mot.

Le listing complet des mots de manipulation de la pile est le suivant :

```
DUP | a -- a a
DROP | a --
OVER | a b -- a b a
PICK2 | a b c -- a b c a
PICK3 | a b c d -- a b c d a
PICK4 | a b c d e -- a b c d e a
SWAP | a b -- b a
NIP | a b -- b
ROT | a b c -- b c a
2DUP | a b -- a b a b
2DROP | a b --
3DROP | a b c --
4DROP | a b c d --
2OVER | a b c d -- a b c d a b
2SWAP | a b c d -- c d a b
```

On peut dire que la pile est une mémoire de courte durée pour l'ordinateur, et bien que dans un premier temps il s'avère difficile de penser à la pile, elle est la clef de tout le langage aussi ne vous découragez pas si au début son fonctionnement vous paraît complexe, la pratique du langage la rend transparente et son mécanisme devient rapidement limpide.

Exercices

(Solutions à la fin du document)

Avec des nombres, DROP, SWAP, DUP et OVER écrivez le code pour obtenir les piles suivantes en utilisant le moins de nombres possibles.

- 1.- (2 3 2 3 2 3
- 2.- (2 2 2 2 2 2
- 3.- (1 2 3 4 5 6
- 4.- (
- 5.-)

Début de programme

Le code :R4 se sauvegarde dans des fichiers texte, pour exécuter le code :R4 charge le fichier source, le compile et ensuite l'exécute.

L'exécution débute à l'endroit où se trouve le mot : (deux points), il ne faut pas confondre avec la définition des mots où le : (deux points) est collé (préfixe) au mot qu'il définit.

```
:debut ;
:corps ;
:fin ;
```

```
: debut corps fin ;
```

Si le compilateur trouve une erreur il écrit un fichier "debug.err" dans lequel sont sauvés le code, l'erreur, le numéro de ligne, lorsqu'il existe un fichier debug.txt ce programme l'utilise.

Mots pour calculer

Les mots utilisés pour réaliser les calculs arithmétiques, sont appelés communément des signes mais nous avons déjà vu qu'ici cette distinction n'est pas nécessaire.

+ (signe de la somme)

```
2 3 +
( 5
```

Le mot + (somme) prend les deux premiers numéros de la pile, les additionne et ajoute le résultat au sommet de la pile (après avoir enlevé les deux valeurs antérieures). Le reste des opérations basiques fonctionnent de la même manière avec leur signification correspondante.

+	a b -- c	c = a + b
-	a b -- c	c = a - b
*	a b -- c	c = a * b
/	a b -- c	c = a / b
*/	a b c -- d	d = a * b / c (multiplication 64 bits)

```

>> | a b c -- d   d = (a * b) >> c (multiplication 64 bits)
/MOD | a b -- c d   c = a / b d = a modulo b
MOD | a b -- c     c = a modulo b
NEG | a -- b     b = -a
1+  | a -- b     b = a + 1
1-  | a -- b     b = a - 1
2/  | a -- b     b = a / 2
2*  | a -- b     b = a * 2
<<  | a b -- c   c = a << b déplace les bits à gauche
>>  | a b -- c   c = a >> b a déplace les bits à droite (signe compris)

```

Mots logique

Comme pour les mots arithmétiques, le langage propose un ensemble de mots pour le calcul logique

```

AND | a b -- c   c = a AND b
OR  | a b -- c   c = a OR b
XOR | a b -- c   c = a XOR b
NOT | a -- c     c = NOT a

```

Les mots arithmétiques tout comme les mots logiques effectuent leurs calculs sur la pile de données, on voit donc que tous les calculs s'y effectuent.

Agrandir le dictionnaire

Comme déjà mentionné le dictionnaire est composé d'un ensemble de mots basiques. Programmer en :R4 consiste en la définition de mots et pour cela on utilise le préfixe : (deux points), tout mot débutant par : est une définition ajoutée au dictionnaire.

Le mot :hello présent dans le code ne déclenche pas la recherche de :hello mais plutôt l'ajout au dictionnaire du mot hello (sans les :) et sa définition suit jusqu'au mot ; (point virgule) . Il est très important de se souvenir des espaces. Examinons une définition dans un source :

```
:s3 3 + ;    | a -- b    ( b sera égal à a+3 )
```

La présence de ce fragment de code dans le fichier source ne signifie pas que la pile sera modifiée et contiendra une valeur égale à la valeur du sommet plus 3, mais plutôt que le mot s3 a été ajouté dans le dictionnaire. Le comportement sera exécuté lorsque le mot sera appelé APRÈS avoir été défini.

```
:s3 3 + ;
2 s3
```

Dans ce cas, la somme est calculée, remarquez que bien que le mot s3 n'existe pas dans le dictionnaire basique, une fois défini il peut être utilisé (un mot doit toujours être défini avant son utilisation).

: (deux points), s'utilise en :R4 comme mot et comme préfixe, nous avons déjà vu que de nouveaux mots peuvent être définis en utilisant ce caractère comme préfixe. Au bout du fichier source, on peut définir où commencera l'exécution du programme avec le mot : (deux points) le code vu auparavant sera prêt lorsque l'on aura indiqué où doit commencer le programme.

```
:s3 3 + ;  
: 2 s3 ;  
( 5
```

Les définitions : (deux points) sont des actions auxquelles on affecte un nom, programmer consiste à créer ces noms et leurs définitions. Chaque définition consomme ou produit des nombres qui sont stockés sur la pile, c'est ce pourquoi la notation de pile est utilisée.

Il est important de trouver les noms corrects pour chaque situation.

Exercices

(Solutions à la fin du document)

Définir les mots pour obtenir les changements de pile suivants :

1.- Définir le mot carré avec le comportement suivant :

```
:**2 | a -- a*a
```

2.- Définir la somme des trois nombres au sommet de la pile avec le comportement suivant :

```
| a b c -- a+b+c
```

3.- Supposons que les mots 2drop et 2dup n'existent pas dans notre dictionnaire. Ecrivez une définition pour ces mots en validant votre réponse avec la notation de changement de pile.

4.- Ecrivez un mot qui inverse l'ordre des nombres de la pile

```
| 1 2 3 4 --- 4 3 2 1
```

5.- Ecrivez le mot 3dup qui duplique les 3 nombres au sommet de la pile

| 1 2 3 -- 1 2 3 1 2 3

6.- Définissez -rot comme suit

:-rot | a b c -- c a b

7.- Définissez les équations suivantes en montrant les effets produits sur la pile,

- a) $a^{**2}+b^{**2}$ | a b -- c
- b) $a^{**2}+ab+c$ | c a b -- d
- c) $(a-b)/(a+b)$ | a b - c

8.- Dessinez quelque chose qui vous plait pour oublier tous les calculs précédents.

Les variables du programme

De la même manière que le préfixe : (deux points) sert à définir des mots qui vont réaliser un certain type d'action , le préfixe # (dièse) sert à définir des mots qui vont contenir des données, ces mots sont aussi appelés variables.

#donnée

La ligne ci-dessus définit le mot donnée ou encore la variable donnée comme une position en mémoire qui permet de stocker un nombre, cet emplacement réservé peut-être initialisé avec une valeur comme montré dans ce qui suit :

#donnée 22

On utilise deux mots du dictionnaire pour réaliser l'opération d'écriture dans une variable ou l'opération de lecture de la valeur stockée par une variable de cette manière:

Lire la mémoire

@ | adresse – valeur

Ce mot retourne la valeur stockée dans une adresse mémoire.

Ecrire à une adresse mémoire

! | valeur adresse --

A partir d'une valeur et une adresse, ce mot écrit la valeur dans la variable correspondante.

Il est utile, arrivés à ce point, de connaître un autre préfixe : ' (apostrophe), il s'utilise pour obtenir l'adresse assignée par le langage à un mot déjà défini, cette adresse s'utilise pour lire et écrire les valeurs des variables.

```
#dato 8'  
dato @ | empile un 8  
2 'dato !  
'dato @ | empile un 2
```

Un mot défini avec # a un fonctionnement distinct d'un mot défini avec un préfixe :. Un mot : est appelé lorsque son nom apparaît dans le code, en revanche, une variable (un mot défini avec #) empile la valeur qu'elle contient lorsque son nom apparaît dans le code.

Ainsi @ n'est pas utile la plupart du temps puisqu'une donnée empilera sa valeur et non son adresse. Les deux lignes suivantes produisent le même résultat, parce que la seconde ligne empile son adresse et obtient ensuite la valeur avec @.

```
donnée | empile la valeur de 'donnée'  
'donnée @ | empile la valeur de 'donnée'
```

Dans les exemples précédents le mot donnée, contenant une valeur, est défini. Dans un but de simplicité on dit que l'on a défini la variable donnée. La valeur de la variable donnée est le nombre que contient la variable et l'adresse de la variable donnée est la position physique en mémoire où est stockée cette information, notez ici que l'adresse est aussi un nombre, mais l'on n'aura jamais besoin de connaître sa valeur autrement qu'à travers son nom (dans ce cas 'donnée).

Le code :

```
0 10 !
```

est valide mais complètement inutile (sauf si la position 10 de la mémoire signifie quelque chose pour l'ordinateur programmé).

Une construction qui se présente très communément consiste à changer la valeur de la variable à partir de sa propre valeur, ainsi, pour ajouter 1 à la valeur de la variable donnée, nous devons écrire :

```
#donnée 0  
:somme1 donnée 1 + 'donnée ! ;
```

Cette définition peut se ré-écrire de cette manière:

```
#donnée 0
:somme1 1 'donnée +! ;
```

Vu que +! incrémente la valeur de l'adresse mémoire correspondante.

Les définitions du dictionnaire ainsi que les variables :R4 se trouvent en mémoire, lorsque l'on obtient l'adresse d'un mot cette adresse est celle assignée au mot par le langage :R4.

Il est important de pratiquer l'usage des variables, des valeurs et des adresses, un dessin aide toujours à comprendre les différences.

Lors de la définition de variables il est possible d'attribuer plusieurs nombres à une variable, par exemple :

```
#liste 10 20 30 40 50
```

Ici, liste empilera le premier nombre, c'est à dire 10. Comment pensez vous que l'on obtiendra le reste des valeurs ?

...c'est bien çà, avec l'adresse de la variable liste.

```
'liste 4 + @
```

Dans ce cas on empilera le second numéro, c'est à dire 20, si l'on remplace le 4 par un 8 on obtiendra le troisième et ainsi successivement. Pour comprendre pourquoi 4 et pas 1 nous devons savoir comment les nombres sont stockés en mémoire.

:R4 définit chaque nombre sur 4 octets par défaut, mais il est possible d'utiliser moins d'octets en mémoire. De la même manière que @ prend 4 octets, c@ prend 1 octet et w@ prend deux octets, on peut aussi utiliser c! qui écrit un octet et w! qui en écrit deux.

Il faut se souvenir constamment de ces quantités car nous travaillerons avec des adresses et elles seront nécessaires pour calculer l'adresse correcte.

On peut indiquer la quantité de bits des numéros en utilisant des crochets et des parenthèses.

```
#de32bits 1 2 3 | nombres de 32 bits
#de16bits [ 1 2 3 ] | nombres de 16 bits
#de8bits ( 1 2 3 ) | nombres de 8 bits
```

La première variable occupe $4*3=12$ octets en mémoire, la seconde occupe 6 octets et la troisième 3 octets.

Il existe une autre construction pour préciser combien d'octets doit contenir une variable, par exemple pour définir un emplacement de 1024 octets soit 1 k-octet :

```
#de1KBYTE )( 1024
```

Les nombres sont stockés sur 4 octets, les caractères sont stockés sur 1 octet.

Une autre précision au sujet des nombres en :R4, il existe deux préfixes qui permettent d'indiquer les nombres en base binaire et hexadécimale, il s'agit de % et \$ respectivement:

Décimal	Binaire	Hexadécimal
0	%0	\$0
2	%10	\$2
10	%1010	\$A
15	%1111	\$F
16	%10000	\$10

La mémoire contient tout ce que l'ordinateur fait, stocke, et montre et il existe aussi un endroit vide dans la mémoire qui permet de travailler, sans cet emplacement vierge la machine ne pourrait pas créer mais seulement montrer.

Le seul mot qui se trouve dans le dictionnaire de base est MEM, ce mot laisse sur la pile l'adresse de début de la mémoire libre, faites ce que vous voulez avec cette mémoire, c'est la votre, elle fait partie de votre machine.

Structures de contrôle

La décision est le mécanisme par lequel on choisit une option ou un chemin. Pour représenter cette décision on doit indiquer quelle est la condition et ensuite le chemin ou mot sera appelé en fonction de cette condition.

La condition s'évalue, devinez où... oui, sur la pile de données, il y a 4 mots qui pour cela.

- 0? Saute si le haut de la pile est 0
- 1? Saute si le haut de la pile n'est pas 0
- +? Saute si le haut de la pile est positif
- ? Saute si le haut de la pile est négatif

Ces mot utilisent la valeur en haut de la pile pour comparer, sans la modifier.

Un second group de mots comparent les deux numéros en haut de la pile en consommant le premier

=?	a b -- a	a = b ?
<>?	a b -- a	a <> b ?
>?	a b -- a	a > b ?
<?	a b -- a	a < b ?
<=?	a b -- a	a <= b ?
>=?	a b -- a	a >= b ?
and?	a b — a	a ET b a les bits à 1 ?
nand?	a b – a	a ET b a les bits à 0 ?

Le saut conditionnel

Le saut dont il est question ici s'indique avec des parenthèses, n'oubliez pas pour autant que les parenthèses sont aussi des mots, voyons un exemple :

0? (drop)

Cette ligne dans le code retire le sommet de la pile s'il est égal à 0, sinon elle ne fera rien puisqu'elle continuera sans exécuter le drop. Cette construction s'appelle IF, puisqu'elle représente en fait un SI conditionnel.

Un autre type de saut conditionnel permet d'indiquer les deux actions possibles, lorsque la condition est vraie et lorsqu'elle ne l'est pas.

+? (dup)(drop)

Il s'agit de penser à des blocs de mots placés entre des parenthèses, même si en réalité les parenthèses ouvertes "(" , fermé-ouvert ")(" et fermées ")" sont trois mots, dans l'exemple précédent si le haut de la pile contient un numéro négatif le saut s'effectue vers le mot suivant le ")(" , si positif l'exécution continue jusqu'au ")(" et ensuite saute jusqu'au caractère ")".

Ainsi, si le haut de la pile est positif, la valeur est dupliquée et sinon la valeur est retirée de la pile.

Les blocs de mots doivent toujours être refermés correctement, laisser un bloc ouvert est une erreur qui sera indiquée par le langage.

Répéter toujours

Pour répéter l'exécution d'une liste de mots, ces mots doivent être compris au sein de parenthèses, par exemple :

```
:nettoyer | ... définition de nettoyer
;
```

```
:salir | .. définition de salir
```

;

: (salir nettoyer) ;

A l'exécution de ce programme, les deux mots salir et nettoyer seront répétés sans fin.

Répéter tant que

Le nom de cette construction est directement issu de son comportement

Il est plus utile d'utiliser une condition d'arrêt que de boucler sans fin, pour cette définition, notez ici que la position de la condition définit le type de construction :

:printn | n -- imprime le contenu du haut de la pile et le consomme ;

: 5 (1 ?)(1 – dup printn) drop ; |imprime 4 3 2 1 0

Cette dernière ligne se lit :

Empiler 5 et tant que la valeur du haut de la pile n'est pas 0, répéter soustraire 1 à cette valeur et l'imprimer à l'écran, il est nécessaire de la dupliquer avant de l'imprimer parce que l'impression consomme la valeur.

Comme vous le voyez, il est nécessaire de garder à l'esprit l'état de la pile à chaque répétition, les opérations de conditions qui consomment des valeurs de la pile sont généralement plus faciles à lire:

: 0 (5 <?)(1 + dup printn) drop ; |imprime 1 2 3 4 5

C'est à dire: empiler 0 et tant que la valeur est inférieure à 5, lui ajouter 1 et l'imprimer (nous savons déjà que nous devons dupliquer avant d'imprimer).

Répéter jusqu'à

: 5 (1 – dup printn 0?) drop ; | imprime 4 3 2 1 0

Cette construction se répètera jusqu'à ce que le haut de la pile soit nul.

Ces structures de contrôle doivent être correctement positionnées, c'est à dire qu'il doit exister la même quantité de parenthèses ouvrantes que de parenthèses fermantes, de plus ";" (deux points) au sein d'une paire de parenthèses n'achèvent par la définition mais l'exécution du mot.

La pile R

Lorsque l'on appelle un mot, l'adresse à laquelle l'exécution du code doit reprendre après la fin de l'exécution de ce mot est empilée sur la pile R.

Pour utiliser la pile R on peut utiliser ces mots :

RDROP	--	r: a --
R>	a --	r: -- a
>R	-- a	r: a -
R	-- a	r: a -- a
R+	v --	r: a -- a+v
R!+	v --	r: a -- a+4
R@+	-- v	r: a -- a+4

La pile R gère le flux d'exécution du programme, ainsi dans l'exemple suivant :

```
:un "un" print rdrop ;
:deux un "pas de passage" print ;

:quefait? deux ;
```

"pas de passage" ne s'affiche jamais car rdrop retire une adresse de la pile R.

Le mot EXEC (défini plus tard) peut aussi se définir en manipulant la pile R :

```
:exec 0? ( drop ; ) >r ;
```

Il est possible et utile d'utiliser cette pile comme une pile auxiliaire pour données, pour garder des résultats intermédiaires ou (saltear calculos ??).

Ici la pile R fonctionne comme un emplacement où libérer la pile.

```
:hlink | x1 y1 x2 y2 -
over >r op r> over line line ;
```

La pile R peut s'utiliser pour parcourir des portions de mémoire en écriture (R!+) ou encore en lecture (R@+).

Ici la mémoire qui commence à 'debut' est parcourue jusqu'à ce qu'un 0 soit rencontré, en copiant chaque numéro à partir de l'adresse passée en paramètre sur la pile.

```
:recorre | a --
>r debut ( @+ 1? )( r!+ ) r!+ rdrop ;
```

Vecteurs

On appelle vecteur un variable qui contient une adresse mémoire, mais à cette adresse se trouve du code et non des données, ce qui permet de changer sa signification à tout moment et apporte ainsi un important outil de construction. Voyons maintenant un exemple:

```
:sum1 1 + ;
:sum2 2 + ;
#sum
:define1 'sum1 'sum ! ;
:define2 'sum2 'sum ! ;
: 3 define1 sum exec define2 sum exec ;
```

(6

Il y a une manière de définir des comportements sans leurs affecter de noms qui sont appelés des mots anonymes, ces définitions s'utilisent lorsque l'on définit un comportement à utiliser qu'une seule fois et qui s'appelle alors à travers son adresse, notez ici que cette adresse sera appelée avec le mot EXEC, que ce soit immédiatement ou dans une autre définition.

Les crochets permettent de construire ces définitions.

[exit ;] >esc< | comportement assigné à la touche ESC

[(crochet ouvrant) comme] (crochet fermant) sont des mots, le premier indiquant que le code commençant juste après ne s'exécutera pas immédiatement, le deuxième finalisant le code et empilant l'adresse du code ainsi défini.

Punto Fijo

:R4 usa como alternativa a los numeros enteros los numeros decimales almacenados en punto fijo
Para esto transforma la notacion de dos numeros con un punto en un entero.
Se utiliza 16.16 o se 16 bits para entero, 16 bits para decimal.

asi

2.0 es \$20000

1.0 es \$10000

0.5 es \$7fff

La suma y la resta no necesitan ajuste.

2.0 1.5 + 0.03 -

la multiplicacion necesita correrse 16 bits a la derecha

```
.* 16 *>> ;
```

la division necesita 16 bits a la izquierda en el primer operando

```
:/ swap 16 << swap / ;
```

ahora es posible + - * /

```
3.2 0.02 * . 3.1 / .
```

```
|-----  
-----
```

Le système

Le groupe de mots suivants permet de manipuler certains aspects du système, les trois premiers mots obtiennent de l'information et les deux suivants contrôlent la machine virtuelle.

MSEC | -- a

Empile les millisecondes actuelles du système

TIME | -- h m s

Empile l'heure, les minutes et les secondes du système

DATE | -- d m a

Empile le jour, le mois et l'année du système

END | --

Arrête la machine virtuelle, lorsque :R4 s'exécutera sur une machine réelle ce mot éteindra l'ordinateur, pour l'instant, il permet de retourner au système d'exploitation sur lequel s'exécute la machine virtuelle.

RUN | d -

Charge le code indiqué par le nom de fichier et l'exécute. Cela permet d'enchaîner les programmes.

L'écran

L'écran de l'ordinateur est une matrice de points lumineux pouvant chacun s'éclairer d'une couleur différente, chaque petit point s'appelle pixel.

La largeur et la hauteur de l'écran varient en fonction des besoins de l'utilisateur du programme :R4.

Pour obtenir plus de puissance de calcul on utilisera un écran plus petit puisque

la machine nécessitera alors moins d'efforts pour la dessiner.

SW	-- w	Largeur de l'écran en pixels (ScreenWidth)
SH	-- h	Hauteur de l'écran en pixels (ScreenHeight)
CLS	--	Efface l'écran avec la couleur de fond
REDRAW	--	Copie l'écran virtuel sur l'écran réel
FRAMEV	-- a	Adresse du début de la mémoire vidéo
UPDATE	--	Rafraîchit les évènements interne du système d'exploitation

Dessin

Pour dessiner à l'écran on utilise les mots suivants. Les coordonnées sont passées par l'intermédiaire de la pile et les nombres correspondent à des positions physiques à l'écran, il ne s'agit pas d'utiliser des constantes en l'occurrence mais plutôt des variables calculées en fonction des dimensions de l'écran afin que le dessin soit indépendant de la résolution.

OP	x y --	Point d'origine
CP	x y --	Point de contrôle d'une courbe
LINE	x y --	Tracer une ligne
CURVE	x y --	Tracer une courbe
PLINE	x y --	Tracer une ligne de polygone
PCURVE	x y --	Tracer une courbe de polygone
POLI	--	Tracer un polygone
PAPER	a --	Choisir la couleur de fond
INK	a --	Choisir la couleur de dessin
INK@	-- a	Empiler la couleur de dessin actuelle
ALPHA	a --	Définir la transparence de la couleur actuelle

Interaction

Pour l'interaction avec l'utilisateur on assigne des actions aux touches du clavier ainsi qu'à la souris.

XYMOUSE	-- x y	Coordonnées de la souris
BMOUSE	-- b	Etat de la souris
KEY	-- s	Dernière touche enfoncée

IPEN!	v --	Evènement de la souris
IKEY!	v --	Evènement du clavier

Fichiers

La manipulation de la mémoire externe avec :R4 se réalise avec les mots suivants:

DIR	"" –	Change de répertoire actuel
FILE	nro -- ""	Obtient le nom de fichier à partir de son numéro
LOAD	's "" -- 'h	Charge un fichier en mémoire
SAVE	's c "" –	Ecrit une portion de mémoire dans un fichier

Jusqu'ici ont été couverts les mécanismes basiques du langage, les mots décrits dans ce qui suit complètent ce dictionnaire de base.

Extension du vocabulaire

Le dictionnaire de base est construit sur la machine virtuel qui exécute le langage, pour faciliter la programmation on crée des bibliothèques ou vocabulaires spécifiques pour chaque aspect.

Bibliothèques

reda4.txt – système
gui.txt – boutons, écran
fontv8.txt – fonte de caractères vectorielles
sprites.txt – dessins multicolores vectoriels

Installation ?

:R4 ne nécessite pas de procédure d'installation particulière, il s'agit simplement de décompresser le fichier .ZIP (en n'importe quel point du disque) téléchargé du site reda4.org et d'exécuter ensuite le programme `reda4.exe`

Comment démarre :R4 ?

Lorsque l'on appelle `reda4.exe`, ce programme charge le fichier source `main.txt` et l'exécute. S'il vous plaît, modifiez ce fichier pour ajouter, enlever ou encore modifier les programmes, n'ayez pas peur de casser quoi que ce soit, il suffit de garder une copie du fichier original si vous souhaitez restaurer l'état initial.

Observez qu'au sein d'un programme on peut en appeler un autre en utilisant:

```
"autre.txt" RUN
```

notez que `"autre.txt"` est le nom du fichier source que le programme souhaite appeler, le nouveau programme commencera à s'exécuter ou bien indiquera à l'écran l'erreur rencontrée avec le numéro de ligne et une description du problème.

Le code source de chaque programme est le fichier texte dans le répertoire précédemment décompressé ... il suffit de l'ouvrir et de l'éditer avec NOTEPAD par exemple.

Derniers mots

Le souhait de l'auteur est de rencontrer des gens qui utilisent, améliorent et augmentent la quantité de programmes pour :R4.

Ne vous découragez pas si au départ vous ne savez pas par où commencer un programme, c'est normal, en FORTH comme en :R4 la brièveté du code oblige à réfléchir pleinement à tous les détails.

Si un groupe de mots se répète beaucoup, faites un autre mot et voyez comme le code se réduit, quelques fois quelques mots disparaissent, prenez l'habitude d'effacer autant de code que vous en écrivez.

N'utilisez que des nombres entiers, si vous avez besoin de fractions multipliez l'unité afin de la diviser en parties, n'utilisez pas de nombres flottants, cela cumulerait des imprécisions dans la représentation de vos nombres.

Essayer toujours de réduire la taille du code, comprendre peu de mots est toujours plus simple que d'en comprendre beaucoup, avec en prime une exécution plus rapide bien entendu.

Comme l'a dit quelqu'un, la simplicité est le point d'arrivée, pas le point de départ.

Programmer en :R4

Il n'y a pas si longtemps 64 ko représentaient beaucoup de mémoire, les magazines listaient des programmes que l'on pouvait entrer au clavier et sauvegarder ensuite sur des supports cassette, la sauvegarde sur cassette n'était pas vraiment fun mais la saisie si, car on pouvait apprendre pendant la saisie.

Etudiez l'exemple suivant, Les numéros sont utilisés pour référencer les lignes et ne doivent pas être copiés.

```
[1]^reda4.txt
[2]^gui.txt
[3]:debut
[4]   'exit >esc<
[5] show cls
[6]   16 16 screen blanco
[7]   "Hello world" print ;
[8]
[9]: debut ;
```

Les lignes 1 et 2 permettent d'inclure des mots pour les graphiques, l'animation le clavier et les fontes de caractères, ces lignes s'utilisent presque toujours.

La ligne 3 définit le mot debut qui est appelé au début du programme à la ligne 9. La ligne 4 assigne l'action de sortir du programme à la pression de la touche ESC.

La ligne 5 utilise SHOW pour indiquer les mots qui dessinent l'écran jusqu'au caractère ; (point virgule) de la ligne 7, le premier mot, CLS, permet d'effacer l'écran.

La ligne 6 définit la taille des lettres qui seront dessinées, screen permet ainsi de spécifier le nombre de colonnes et de lignes qui serviront à partager la surface de l'écran, ensuite on spécifie la couleur du dessin à venir (blanco = blanc en espagnol).

La ligne 7 affiche à l'écran le texte compris entre les apostrophes.

Exemple d'animation en :R4

Voyons un exemple d'animation d'une balle rebondissante:

```
[1]^reda4.txt
[2]^gui.txt
[3]#balle $cc004 $7493FE85 $6BCE50D7 $22DC7 $93D250D7
[4]$8B73FE87 $9435B557 $1CF37 $6C31B3C7 $7493FE87 0
[5]#xb 0 #yb 0 #vx 8 #vy 0 #ay 2
[6]
[7]:toc
[8]    vx neg 'vx ! ;
[9]
[10]:tic
[11]   vy neg 'vy ! ;
[12]
[14]:ecran
[14]   'exit >esc<
[15]   show cls
[16]       100 100 dim xb yb pos 'balle sprite
[17]       xb vx + sw >? ( toc ) 0 <? ( toc ) 'xb !
[18]       ay 'vy +!
[19]       yb vy + sh >? ( tic drop sh ) 'yb ! ;
[20]
[21]: ecran ;
```

Les lignes 1 et 2 permettent d'inclure les extensions.

Les lignes 3 et 4 définissent un espace de 11 nombres de 32 bits, ces numéros représentent le dessin de la balle, ce type de dessin peut être réalisé avec le programme RMATION et l'on peut ensuite utiliser le fichier dibujos.txt (dibujos = dessins en espagnol) généré par ce programme.

La ligne 5 définit les variables qui vont être utilisées , XB et YB sont les

coordonnées qui sont utilisées en ligne 16 comme position du dessin à l'écran (le mot sprite dans le jargon des anciens jeux vidéos). VX VY représentent la vitesse de chaque coordonnée et AY est l'accélération verticale, à envisager un peu comme la gravité.

Les lignes 7 et 8 définissent TOC qui permet de changer le signe de VX (quand il y a un choc contre les bords)

Les lignes 10 et 11 définissent TIC qui permet de changer le signe de VY (lorsqu'il y a un choc contre le 'sol')

Remarquez que la ligne 21 est l'endroit où commence le programme comme indiqué avec un caractère : suivi d'un espace avec à la suite le mot défini entre les lignes 14 et 19.

La ligne 14 assigne l'action de la pression de la touche ESC au comportement de sortie de la routine de dessin de l'écran.

Remarquez que l'assignation d'actions aux touches du clavier est directe, il n'y a pas de comportement plus rapide et simple que le fait d'assigner à chaque événement externe une action.

La ligne 15 utilise SHOW pour indiquer le début des mots utilisés pour dessiner l'écran, ce dessin se réalise 30 fois par seconde en répétant les mots qui se trouvent entre SHOW et la fin de la définition. CLS efface l'écran, retirez CLS du programme et voyez ce que cela change.

La ligne 16 définit en premier les dimensions du dessin, changez les 100 avec le mot DIM (qui indique une dimension) et voyez ce qui se passe. Ensuite la position est indiquée par les variables XB et YB comme nous l'avons déjà vu, avant le mot POS (pour position) et ensuite le dessin défini en ligne 3 est affiché avec le mot SPRITE.

La ligne 17 calcule le mouvement horizontal, vu pas à pas, voici ce qui se passe:

```
xb    | empile la valeur de XB
vx    | empile vx
+     | ajoute les deux
sw    | empile la largeur de l'écran en pixels
>?   | la valeur issue de l'addition est elle supérieure à la largeur de l'écran?
(     | si oui, alors il faut exécuter le code entre les ( )
toc   | rebond ( défini avant)
)     |
0     | empiler un 0
<?   | la valeur issue de l'addition est elle inférieure à 0 ?
(     | si oui, alors il faut exécuter le code entre les ( )
toc   | rebond
)     |
'xb   | empiler l'adresse de xb
!     | Sauvegarder dans la variable xb la somme calculée plus haut
```

La ligne 18 ajoute AY à VY

La ligne 19 est similaire à la ligne 17 mais elle ne teste que sol, remarquez que dans ce cas on remplace la valeur ajoutée si elle est supérieure au sol.

Pour essayer:

Agregue entre la 7 y la 8 la siguiente linea

Ajoutez entre les lignes 7 et 8 la ligne suivante :

rand 4 << 4 or 'balle !

Qu'arrive-t-il à la balle ?

Solutions

Exercices 1

```
2 3 over over over over
2 dup dup dup dup dup
1 2 3 4 5 6
```

????????????????

Exercices 2

```
:**2 dup * ;
:suma3 + + ;
:2drop drop drop ;
:2dup over over ;
:inv4 1 2 3 4 swap 2swap swap ;
:3dup dup 2over rot ;
:-rot rot rot ;
:resa **2 swap **2 + ;
:resb over + * + ;
:resc 2dup - -rot + / ;
```

Reference base :R4

Prefixes					
^modulo.txt			include modulo.txt file		
:new			define new as the list until the ;		
::newg			define and export this definition		
#memory			define memory as a variable		
#:memoryg			define and export the variable		
'hello			push the address of hello (previously defined)		
blabla...			comment, ignore until end of line		
"bla bla"			text (include spaces), push address		
Control					
(...)			Repeat WORDS in between parentheses		
?? (.v.)(.f.)			Condition		
(.f. ??)			Repeat until condition (run at least once)		
(.. ??)(.v.)			Repeat while condition		
[...]			Anonymous definition(push its address)		
EXEC	d --		Call the address on the stack		
Conditionals					
0?	--	1?	--	Is the top of the stack null/(not null) ?	
+?	--	-?	--	Is the top of the stack positive/negative ?	
=?	ab - a	<?>	ab - a	a = b ? a <> b ?	
>?	ab - a	<?	ab - a	a > b ? a < b ?	
<=?	ab - a	>=?	ab - a	a <= b ? a >= b ?	
and?	ab - a	nand?	ab -- a	Test bits	
Data stack					
DUP	a -- aa	ROT	abc -- bca		
DROP	a --	2DUP	ab -- abab		
OVER	ab -- aba	2DROP	ab --		
PICK2	abc -- abca	3DROP	abc --		
PICK3	abcd -- abcda	4DROP	abcd --		
PICK4	abcde - abcdea	2OVER	abcd -- abcdeb		
SWAP	ab -- ba	2SWAP	abcd -- cdab		
NIP	ab - b				
Logic					
AND	ab -- c	c = a AND b	XOR	ab -- c	c = a XOR b
OR	ab -- c	c = a OR b	NOT	a -- b	b = NOT a
Arithmetic					
+	ab -- c	c=a+b	NEG	a -- b	b= -a
-	ab -- c	c=a-b	ABS	a -- b	b= a
*	ab -- c	c=a*b	1+	a -- b	b=a+1
/	ab -- c	c=a/b	1-	a -- b	b=a-1
*/	abc -- d	d=a*b/c	2/	a -- b	b=a/2
/MOD	ab -- c d	c=a/b	2*	a -- b	b=a*2
MOD	ab - d	d=a modulus b	<<	ab -- c	c=a<<b
*>>	abc - d	d=(a*b)>>c	>>	ab -- c	b=a>>b

Memory					
@	a - b	b=32(a)	@+	d - d+4 v	
C@	a -- b	b=8 (a)	C@+	d -- d+1 byte(v)	
W@	a -- b	b=16(a)	W@+	d -- d+2 word(v)	
!	vd --	32(d) =v	!+	vd -- d+4	
C!	vd --	8(d) =v	C!+	vd -- d+1	
W!	vd --	16(d)=v	W!+	vd -- d+2	
+	vd --	32(d)=32(d) + v	W+!	vd -- 16(d)=16(d) + v	
C+!	vd --	8(d)=8(d) + v	MEM	-- d free memory	
Return stack					
>R	a --	R: --a	R<	-- a	R:a --
R	-- a	R:a--a	R+	v --	R:a - a+v
R@+	-- v	R:a--a+4	R!+	v --	R:a -- a+4
			RDROP	--	R:a --
System					
MSEC	-- a	Milliseconds of the system			
TIME	-- h m s	Hour, minutes and seconds			
DATE	-- d m a	Day, month and year			
RESET	--	Exit from :r4, finalize and shutdown virtual machine			
RUN	d --	Load, compile and execute the file which name is in d			
Screen					
SW	-- w	Push screen width on the stack			
SH	-- h	Push screen height on the stack			
CLS	--	Erase screen			
REDRAW	--	Draw real screen with virtual one			
FRAMEV	-- a	Push start of virtual screen on the stack			
UPDATE	--	Update internal SO events			
Drawing					
OP	xy --	Point of origin			
CP	xy --	Control point for the curve			
LINE	xy --	Draw line			
CURVE	xy --	Draw curve			
PLINE	xy --	Draw polygon line			
PCURVE	xy --	Draw polygon curve			
POLI	--	Draw polygon			
Color					
PAPER	a --	Set background color			
INK	a --	Set current drawing color			
INK@	-- a	Push current drawing color on the stack			
ALPHA	a --	Set alpha (0-255)			
External storage					
DIR	"" --	Change current directory			
FILE	n -- ""	Get the name given the number			
LOAD	a"" -- b	Load a file in memory			
SAVE	ac"" --	Write a memory chunk			