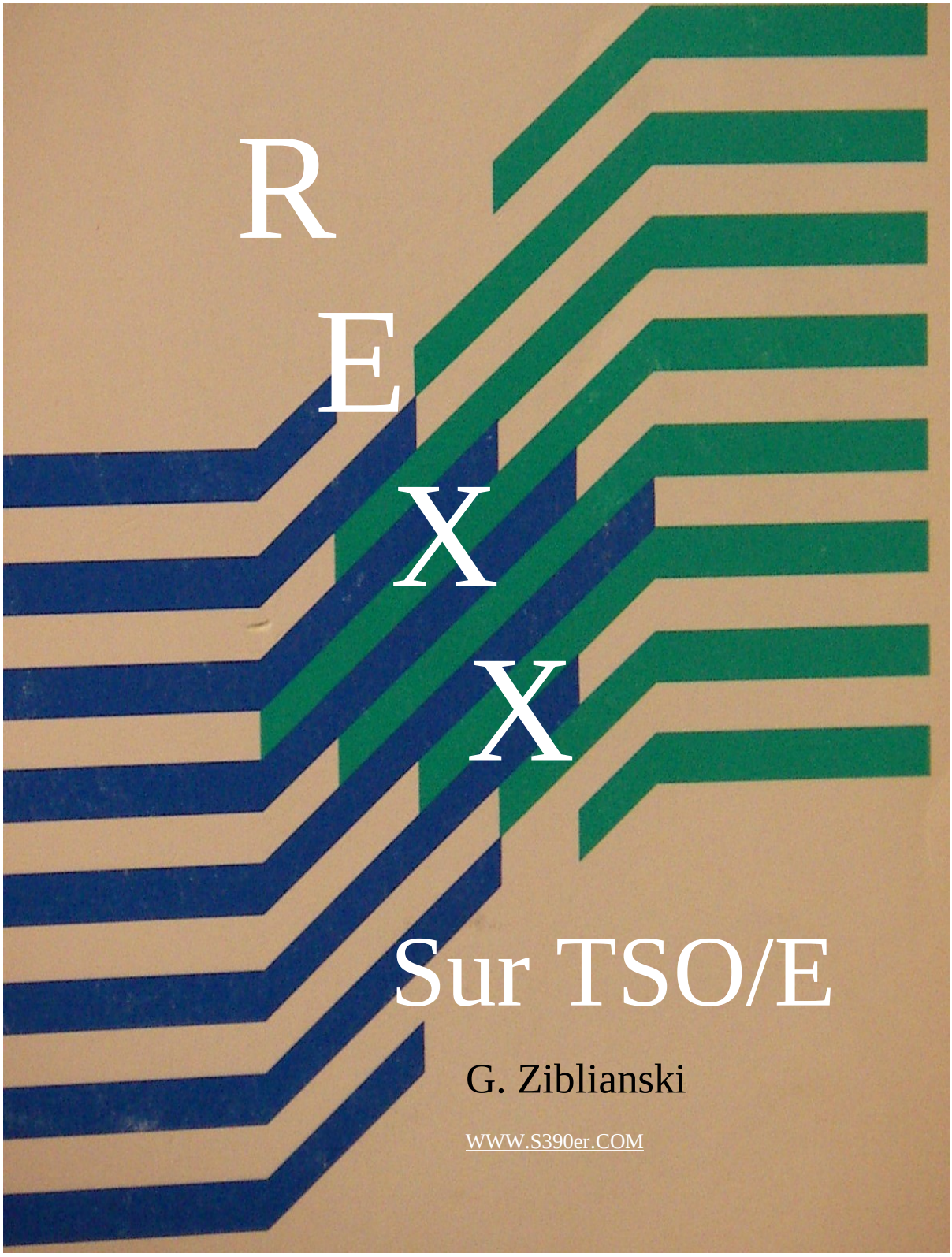
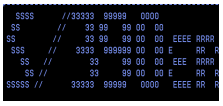
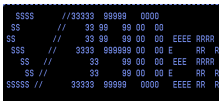


Parution Mai 2006
Publié par S390er



REXX S390er

Edition	première	(Juin 2004)
	Seconde	(novembre 2005)
	Troisième	(Mai 2006)



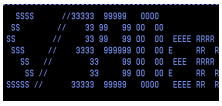
Introduction

Est ce vraiment nécessaire ?

Ce petit bouquin a pour but de vous apprendre les bases du REXX sous TSO (autant dire MVS).
Bien des bouquins traitent de REXX en montrant seulement les instructions et autres fonctions et leurs résultats, mais si vous attaquez sous MVS…. d'ailleurs comment fait-on pour écrire un REXX et l'exécuter sur MVS ?
Connaître les instructions et autres fonctions, c'est bien…mais après ?
C'est à cette question que je comptre répondre par le présent ouvrage.

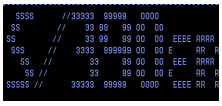
Afin de rendre l'apprentissage plus interressant, je vous présenterai ici les bases pour écrire des EXEC (attaquant différentes interfaces MVS telles que DB2, TSO, ISPF etc) qui vous montreront le VRAI pouvoir de REXX sur MVS.

Amusez vous bien ! !



Edition 2 Novembre 2005

Sommaire :



Histoire du REXX

De 1979 a 1982 un certain Mike Cowlishaw travaille sur un projet personnel (REX, estimé à 4.000 heures de travail), au laboratoire d'IBM près de Winchester en Angleterre et au Watson Research Center à New York (USA).
Ce projet est le developpement d'un langageorienté utilisateurs – dont la syntaxe est facile à comprendre par tous, structure et portable sur différentes plate-forme.
Ainsi selon Mike Cowlishaw, c'est le 20 Mars 1979 que REX est devenu réalité, en effet, Mike c'est réveillé à trois heures du matin ce jour avec une « idée claire de ce qu'il fallait et à la fin de la journée j'envoyais les specifications autour du monde pour commentaires. » Propos recueillis par Lee Peedin VP RexxLA.

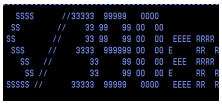
Originellement REX apparait plutot comme un langage de commandes prevu pour remplacer – à terme, L'EXEC 2 de VM/SP, ainsi qu'il est spécifié dans un document interne d'IBM 'a reformed eXecutor – REX. Initial specification' du 29 mars 1979. En effet, dans ce document, Mike Cowlishaw compare les EXEC et EXEC2 à REX, cette tendance sera confirmée par une note a SHARE de Mike Cowlishaw (REX – A Command Programming Language datée du 18 Fevrier 1981), et insiste sur une meilleure lisibilité du langage, et une facilité d'accès pour les utilisateurs en général et pas seulement les informaticiens (dans son livre TRL –voir plus bas, Mike écrit qu'il a limité le nombre d'instructions du langage volontairement), ainsi que la possibilité de structures de contrôle.

REX est distribué en interne chez IBM grâce au système de communication VNET d'IBM, ce qui contribuera à de rapides échanges entre les utilisateurs et le développement.

Ce n'est qu'à la version 2.50 de REX qu'apparaîssent les notions de portabilité et d'indépendance vis à vis du système d'exploitation.
Fin 1982, la version 3 de REX est sortie en interne chez IBM.

Après cette période de developement rapide la version system/370 de Mike Cowlishaw fut intégrée au système VM/SP comme 'System Product Interpreter' de CMS (éditeur VM) en 1983.
Cela explique pourquoi beaucoup de spécialistes REXX viennent du monde VM.

En 1983 (juillet) la version finale de REX est sortie en interne chez IBM, ce sera la dernière version de REX précurseur de REXX.
1985 est l'année de la première Edition de The REXX Language(a practical approach to programming) connu par les développeurs REXX comme 'TRL', par Mike Cowlishaw.
En 1987 IBM choisit REXX comme langage procédural pour les systèmes SAA (systems application architecture).
En clair, REXX sera dorénavant disponible sur MVS, OS400, OS/2 et bien sur VM, c'est ainsi qu'en 1988 IBM sort la version 2 de TSO sur MVS avec le REXX intégré (en plus du langage CLIST).



Histoire du REXX

En 1989 sort le compilateur REXX.
En 1990 2eme édition de TRL par Mike Cowlishaw.
En 1995 REXX s'exécute sous CICS ! ! ! !
En 2000 Nouvel environnement accessible par REXX : DB2 ! ! !
Depuis REXX est sorti sur plusieurs plate-formes (PC, NETWARE, INTERNET) et en plusieurs versions :
Object REXX, NETREXX, etc…

Notions Générales du langage REXX

Un REXX est formé par des 'clauses' terminées par des points-virgules (;) implicitement par les biais de LINE END, explicitement si on code plusieurs ordres sur la même ligne.
En gros, une clause représente un ordre REXX et ne peut dépasser 250 caractères de long..

Il existe 3 sortes de clauses :

Null clauses (clauses nulles ou vides) :
Il s'agit de lignes laissées a blanc ou de commentaires totalement ignorées par REXX lors de l'exécution, et seul les commentaires seront tracés (si codage).
Une NULL CLAUSE n'est pas une instruction

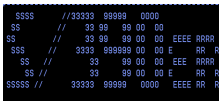
Labels
Une clause ne comportant qu'un nom suivi de deux points est un label. Dans ce cas les deux points agissent comme un point virgule, un label est donc une clause.
Les labels sont utilisés pour identifiées le code cible d'un CALL, SIGNAL, d'un appel de fonction interne.
Les labels sont traités commes des 'null clauses' et peuvent être tracés.

Instruction
Une instruction est constituée de une ou plusieurs clauses décrivant une série d'actions a faire par le 'language processor'. Les instructions peuvent être des Assignations (xx = 'value), des mots clef (voir plus bas), ou des commandes (des instructions passées au système – voir partie MVS).

Ces clauses sont formées de **T**OKENS (jetons) pouvant être :

des chaînes de caractères alphabétiques comprises entre '' ou « « . Leurs contenu sont des constantes et ne sont donc jamais modifiées par REXX.
des chaînes Hexadécimales (0à9 et a à f ou A à F). groupées par pair (sauf pour le 1er qui peut contenir un nombre impair de digit). Ces chaînes sont généralement suivies d'un caractère X ou x pour indiquer que c'est un chaîne Hexadécimale.
Des Symboles. Ce sont des chaînes de caractères alphanumériques non comprises entre '' ou '' '' que REXX manipule ainsi, un symbole peut être une variable un mot clef ou même une option, mais REXX agit dessus.

Si REXX trouve une chaîne de caractère entre quotes, il n'ira pas regarder dedans (sauf Si UN ORDRE CALL précède cette chaîne).
Si rexx trouve une chaîne de caractère sans quotes, C'est un symbole il cherche :
1 à l'exécuter. Et s'il ne trouve aucune commande dans son jeu d'instructions ;
2 à l'initialiser (variable) en Majuscule.



Par défaut les variables portent leurs noms de variables. Ainsi :
SAY ERIC VALERIE retournera la même chose que SAY 'ERIC VALERIE'
Et pourtant le premier say retournera des variables et le deuxième un champs littéral.

Ainsi :	Retournera
/* REXX */ ERIC= 'GROS CON' VALERIE='T ES NUL' SAY ERIC VALERIE SAY 'ERIC VALERIE' EXIT	GROS CON T ES NUL ERIC VALERIE

COMMENTAIRES :

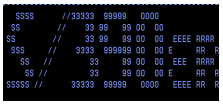
les commentaires sont inclus entre /* et */ et ne peuvent dépasser 250 de long, il est possible d'étendre un commentaire sur plusieurs lignes.

CONTINUATION :
Le caractère de continuation est la vurgule (,).

En REXX, les clauses sont des lignes d'ordres délimitées par ' ;' ou LINE END.
Ces clauses sont faites de TOKENS et de COMMENTAIRES (ceux ci ne sont pas pris en compte.).
Les symboles Entre quotes sont des chaînes et ne seront pas exécutés. Les symboles Non entre quotes seront soit exécutés (si rexx les trouve dans ses instructions) soit initialisés à leur nom.

LES MOTS CLEFS (communs à toutes les implémentations)

Address env	send a command to environment ENV Address ISPEXEC « display panel(panel1) » Attention : l'environnement n'existe pas forcément sur le Système
Arg	Utilisé pour recuperer les arguments passés au REXX . En fait c'est la version 'courte' de *Parse upper ARG*
CALL	appel d'une routine interne ou externe. S'il y a des resultats, ils sont dans la variable *result*
DO	départ de boucle (c.f control structures)
DROP	pour vider une variable
EXIT	sortir (fin) d'un EXEC (programme principal).
IF	Execution conditionelle du code (C ;F control structures).
Iterate	Recommencer (une boucle – c.f control structures).
Leave	Quitter une boucle (c.f control structure).
NOP	ne fait rien.
Parse	Eclater un argument (c.f mise en forme de données).
Procedure	Protège les variables de l'appelant en les rendant invisible à l'appelé. Le paramètre EXPOSE rend les variables de l'appelant spécifiées visibles.
Push	mettre dans le stack (LIFO, c.f stack).
Queue	Mettre dans le stack. (FIFO c.f stack).
Return	retourne le contrôle à l'appelant avec , le cas échéant, un résultat au point d'invocation.
SAY	Affiche le texte qui suit, ou les valeurs des vairiables si elles sont initialisées.
Select	La structure de cas (case structure).
Signal	go to (c.f tracing et debugging)
Trace	Demarre ou arrete la fonction tracing (c.f tracing et debugging)



Ecrire un programme REXX en TSO

Tout le monde peut faire ses premiers pas de REXX sous TSO (OS/390 V1R3) et on commence généralement par un programme qui affiche "Hello world!".
Il est nécessaire d'avoir un PDS pour faire ces exercices, c'est le seul 'pre-requisite'.
Nous allons assumer que vous connaissez l'option 3.4 de ISPF

Aller dans votre PDS (librairies) créez un membre essai (Surtout ne choisissez pas TEST car c'est une commande TSO) et tapez ce code :

```
/* REXX */                                          This program says "Hello world!"
 say "Hello world!"
```

sortez et sauvegardez (généralement PF3) puis taper EX sur la ligne de votre REXX

Hello world! S'affiche
Vous venez d'exécuter votre premier REXX (on dit aussi EXEC)
NOTES :
Les Execs REXX commencent toujours par /* REXX */, Sauf si la recherche automatique de SYSEXEC est activée
Vu la simplicité du codage, nous commencerons toujours nos EXEC par /* REXX */.
Nous allons attaquer les variables par le chapitre arithmétique

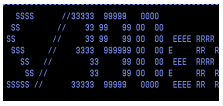
VARIABLES

- **arithmétique**

REXX a une vaste collection d'opérateurs pour exécuter des opérations Nous n'allons pas tous les voir ... mais les principaux, cela nous permettra d'aborder les variables en REXX car il y a quelques différences subtiles comparé aux autres langages.
Ecrivez ce code dans un membre ARITH

```
 /* REXX */
say 'give a number'
 pull a
 b=a*a
 c=1/a
 d=3+a
 e=2**(a-1)
 say 'Results are:' a b c d e
```

Sortir par PF3 et refaire la manipulation indiquée au debut, voici un exemple (page suivante) :



Ecrire un programme REXX

_____ ALLOCX	01.01 98/11/25 13:32	17	17	0	INALBLX
_ex_____ ARITH *RC=0	01.00 99/01/20 09:17	8	8	0	INALBLX
_____ CFTFORM	01.11 99/01/06 17:45	46	92	0	INALBLX

GIVE A NUMBER

5

Results are: 5 25 0.2 8 16

les résultats affichés sont le chiffre original (5), son carré (25), sa réciproque (0.2), le chiffre original +3 (8) et deux à la puissance 1 - le chiffre (16).

Cet exemple montre :

- Dans cet exemple a, b, c, d, e sont des variables ici elles sont facilement reconnaissables grâce à l'opérateur '=' Et c'est la valeur de la variable (à droite de l'opérateur) qui est utilisée dans les opérations où la variable est appelée
- input: l'instruction PULL permet de récupérer une valeur dans une variable soit par le terminal

soit par la 'queue ' ATTENTION l'instruction PULL ne récupère pas BLIGATOIREMENT du terminal

- arithmétique: les symboles usuels (+ - * /) ainsi que ** (à la puissance) ont été utilisé pour effectuer les calculs
- Les parenthèses ont été utilisées pour grouper les expressions… en fait, il s'agit d'influencer l'ordre de précédence des opérateurs.
- la dernière ligne affiche les résultats : noter que si les variables ne sont pas initialisées les mots

A B C D E seront affichés (voir point 4). Noter aussi l'absence de caractère de concaténation.

Le blanc entre l'expression (une expression peut-être littérale, ou même évaluée) et les variables est en fait considéré par l'interpréter (REXX) comme une concaténation avec blanc.

'Results are:' a b c d e

S'affichent 6 composants : le constant littéral et les 5 variables.

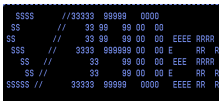
Un constant littéral est toute séquence de caractère commençant et terminant par : ' ou ''

Dans ce programme il sera possible de changer la précision décimale de l'interpreteur par l'ordre

numérique digit XX ou XX est un chiffre

- **Variables non typées**

dans l'exemple déjà donné, il est possible de donner un nom (par exemple). REXX (l'interpreteur) ne détectera pas d'erreur, mais les opérations exécutées sur ces variables se planterons. (on ne peut effectuer des calculs sur des noms) en erreur de syntaxe.



Ecrire un programme REXX

REXX ne vérifie pas la validité des opérations sur les variables ceci afin d'être plus souple et de ne pas 'ennuyer' l'utilisateur avec des détails de type de déclarations de variables comme dans d'autres langages (ref plus haut), cela entraîne que si l'on fait une instruction say sur une variable après l'avoir 'vidée' le nom de la variable s'affichera.

Concaténation : On a déjà abordé ce sujet pour une concaténation avec espace.

/* REXX */	Ce rexx affichera :
SAY 'il fait' ! !'beau'	**il faitbeau**
say 'il fait' 'beau'	**il fait beau**
Exit	

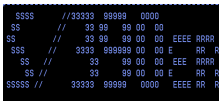
utilisation de ! ! ou || selon les sites. Il y a concaténation sans blanc
utilisation d'un blanc (ou espace) concaténation avec espace.

On a vu les variables et la concaténation, les variables utilisées jusque là n'utilisaient qu'une lettre à titre d'exemple.
Une variables peut faire jusque 250 caractères de long et peut inclure des chiffres et les 6 caractères "@#$!?_". La fonction SYMBOL permet de savoir si elle a été initialisée correctement.

Tableaux a plusieurs dimensions

Une notion importante dans les variables est LES VARIABLES COMPOSEES. En anglais ARRAY PROCESSING. Ce système permet de donner plusieurs valeurs à une variables l'une après l'autre.

/* REXX */	et voilà l'affichage
bk.1 = 'le rexx by da kine'	
bk.2 = 'le rexx by GABE'	
bk.3 = 'le rexx by yen amarre'	
bk.4 = 'le rexx by jean epleinlcu'	
bk.5 = 'le rexx by who else'	
bk.6 = 'le rexx by mike cowlishaw'	
say bk.6 'est 6ieme arg'	**le rexx by mike cowlishaw est 6ieme arg**
say bk.2 'est deuxieme arg'	**le rexx by GABE est le 2eme arg**



Ecrire un programme REXX

Il est possible d'améliorer l'exec page précédente comme suit :
rajouter 2 instructions

- say 'donner un numéro de bouquin'
- pull n

Modifier l'instruction say déjà présente : SAY bk.6 'est 6eme arg' par : SAY BK .N 'est'
N ! !'ieme bouquin'
Retirer la deuxième instruction SAY
Exécutez le : EX devant le nom de membre.

Donnez numéro de bouquin
6
le rexx by mike cowlishaw est 6ieme bouquin
****

note : La variable BK.N est dite composée (Compound variable).
BK. Est la première partie et est appelé RACINE (STEM en anglais).
N est l'extension.

Il est possible de combiner 2 noms de variables, REXX ne limitant pas les variables (elles peuvent être numérique ou alphabétiques) rien n'empêche d'avoir une variables composée dont la racine et l'extension soient des caractères.

Exemple un répertoire téléphonique avec nom et prénom :

```
/* REXX */
say 'donnez prénom et nom'
pull pren nom
say nom pren
03.20.60.82.38 *
ERIC.BOULADOUX='* 8238 * 03.20.60.82.38 *'
ERIC.MARMOUZE= '* 7690 * 03.20.60.76.90 *'
MIKE.CORMIER= '* 7071 * 01.43.12.70.71 *'
say pren ' ' nom value(pren'.'nom)
Exit
```

donnez prenom et nom
Eric bouladoux
ERIC BOULADOUX * 8238 *
***

Ainsi, les variables de chaque coté du point doivent être résolues.
Déjà il est possible de développer des utilitaires, ici une liste téléphonique, quoique il a fallu utiliser une fonction que l'on a pas encore vu.
Autre point que l'on verra plus tard, l'instruction ARG, elle permet de passer des arguments directement à notre programme, mais il est necessaire que le REXX puisse etre appelé directement depuis le champs commande d'ISPF, necessitant une manipulation spéciale que nous verrons plus loin.
Nous venons de voir les variables, un des éléments importants de REXX. et d'autres langages.
Le chapitre suivant vous présente les structures de contrôle (IF/ELSE, WHEN, et boucle).
Les tableaux à plusieurs dimensions sont fort utiles dans la construction de boucle.

Structure de controle

Nous avons vu un terme précédemment, la boucle, celle ci est ce que l'on appelle une 'control structure' tout comme if then else (conditionnels)

IF then else conditionnels

créez le REXX suivant :

```
/* REXX */
say 'donnes moi un chiffre'
    pull a
    if a<10 then say a "is less than 50"
    else say a "is not less than 50"
        exit
```

affiche :

donnes moi un chiffre
5
5 is less than 50
***

Dans cet exemple, a<10 then say ... est une expression conditionnelle, elle ressemble à toute autre expression à tel point qu'une expression numérique ordinaire est interchangeable avec une conditionnelle.

Le programme s'exécute comme il a été tapé..

IF instruction Vrai ? 1 ou 0
else si if faux (0) alors exécute else

```
4 *-* pull a
5
  >>>   "5"
    5 *-* if a<10
      >V>   "5"
      >L>   "10"
      >O>   "1"
      *-*  then
      *-*  say a "is less than 50"
      >V>    "5"
      >L>    "is less than 50"
      >O>    "5 is less than 50"
5 is less than 50
***
```

C'est un résultat d'exécution grâce à l'instruction TRACE I

l'instruction IF est évaluée à un (1) or ici elle est vraie donc 1 = vrai 0 = faux.

POINT important, Le REXX se lit de gauche à droite (c'est écrit par des anglo-saxons mais quand même...) le REXX s'exécute de gauche à droite... (comme il se lit) c'est important pour la précédence des opérateurs,

ainsi le jeux d'instruction :
N=6
SAY N* 100/50'%' sont des rejets
Affichera 12% sont des rejets, on en deduit que les operations arithmetiques ont une plus grande
priorité que la concatenation.
Et surprise pour les anglophobes, IF veut dire SI...et ELSE veut dire SINON.
De là il n'y a rien de plus facile à comprendre....

OPERATEURS :

= (égal à) < (plus petit que) > (plus grand que) <= (plus petit que ou égal à)
>= (plus grand ou égal) <> (plus grand ou plus petit) ^= (non-égal) ^> (pas plus grand)
^< (pas plus petit)

les opérateurs comparent non seulement les chiffres mais aussi les littéraux en enlevant les blancs de début et de fin **è** ' eric ' = 'eric'

OPERATEURS stricts (Ceux la n'enlèvent pas les blancs des arguments => ' eric ' = 'eric' est faux (0)) :
= = (égal) << (plus petit) >> (plus grand) <<= (plus petit ou égal) >>= (plus grand ou égal)
^== (non égal) ^>> (pas plus grand) ^<< (pas plus petit)

LES OPERATEURS BOOLEENS :

& (et), ! (ou) and && ou exclusif). Ces opérateurs peuvent eux aussi être inversé comme précédemment..

Boucles

REXX a une collection complète d'instruction pour exécuter des loops (boucles).

a. Boucles comptées (Counted loops)

les instructions spécifiées dans une boucle comptée sont exécutés le nombre de fois spécifiée

```
/* REXX */                                  Say "Hello" ten times
do 10
  say "Hello"
end
```

Une variation de la précédente :.

```
/* REXX */  il faudra presser la clef interrupt pour l'arrêter
do forever
  nop
end
```

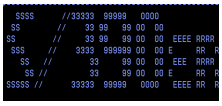
b. Boucle de contrôle

Ce type de boucle rappelle la 'counted loop'. Ici une variable est utilisée comme compteur et devient une 'Variable de contrôle'. Elle peut compter (s'incrementer) de 1 à n valeur c'est à vous de le spécifié..

```
/* REXX */                                  compte jusqu'à 20
do c=1 to 20
  say c
end

 /* REXX */                                 affiche tous les multiples de 2.3 jusqu'à 20
do m=0 to 20 by 2.3
  say m
end

/* REXX */                                          Affiche les 6 premiers multiple de 5.7
do m=0 for 6 by 5.7
  say m
end
```



Structures de controle

Boucles

```
/* REXX */                    Print all the natural numbers
      do n=0
        say n
      end n
```

Le caractère ‘n’ est ici optionnel. Placée à la fin cette variable sera vérifiée pour voir si elle est égale à la variable du début (son contenu).

c. Conditional loops

Un jeu d’instructions peut etre repete jusqu’a ce qu’une condition ‘VRAIE’ soit rencontree :

```
/* REXX */                                  I won't take no for an answer
do until answer \= "NO"
  pull answer
end
```

Elles peuvent etre executees pour autant qu’une condition soit vraie

```
 /* REXX */                                 on repete a tant qu’il ny a pas d’erreur
do while error=0
  pull a
  if a="ERROR" then error=1
  else say a
end
```

Notez qu’ici la variable zero doit etre a 0 (pas d’erreur) pour demarrer sinon le jeu d’instructions ne sera pas éecuté
Dans l’exemple C, les instructions seront au moins execute une fois. En effet les expression d’un « UNTIL » sont évaluees àla fin de la boucle, alors que dans un « WHILE » elles sont evaluées au début de la boucle.

d. Controlled conditional loops

il est possible de combiner a ou b avec c, comme ceci ::

```
      /* REXX */                            won't take no for an answer unless it is typed 3
times
      do 3 until answer \= "NO"
        pull answer
      end
```

Structures de controle

Boucles

Ou comme ceci :

```
 /* REXX */                                    input ten answers, but stop when empty string is entered
  do n=1 to 10 until ans==""
     pull ans
     a.n=ans
  end
```

les instructions "iterate" et "leave" vous permettent de continuer ou de quitter une boucle

```
/* REXX */                                    input ten answers, but stop when empty string is entered
 do n=1 to 10
    pull a.n
    if a.n=="" then leave                                  on quitte la boucle en cours
 end
```

```
/* REXX */                                          print all integers from 1-10 except 3
 do n=1 to 10
    if n=3 then iterate          3 ne sera pas affiche, l'ordre iterate arrivant avant le SAY
    say n
 end
```

il est possible de controler l'execution ou la sortie d'une boucle par la variable qui la definie :

```
  /* Print pairs (i,j) where 1 <= i,j <= 5, except (2,j) if j>=3 */
 do i=1 to 5                                  /* la variable de la bouce est I */
    do j=1 to 5
       if i=2 & j=3 then iterate i            /* "leave j" serait ok,
         say "("i","j")"                      ou leave tout seul          */
       end
 end
```

SELECT (test case)

Permet de choisir une condition parmi plusieurs choix possible. Une clause SELECT se termine TOUJOURS par END

```
/* REXX */
ARG K
Select
When k=’1’ then say k=1
When k=’2’ then say k=2
When k=’3’ then say k=3
Otherwise say ‘number too big’
end
```

SELECT remplace les instructions IF imbriquées

```
/* REXX */
If K=’1’ then say k=1
 Else if k=’2’ then say k=2
  Else if k=’3’ then say k=3
   Else    say ‘number too big’
```

Les structures de controle permettent de controler l’execution d’instructions base sur des conditions ou cas particuliers.
Elles sont d’une importance capitale dans toute programmation, mais seul REXX les rend aussi facile d’utilisation.

Mise en forme de données

Parsing opération qui consiste à éclater l'argument passer en plusieurs variables (Note la casse est gardée) instructions ARG et/ou PARSE.

Avant d'eclater des arguments il est nécessaire de les 'recevoir', pour ce faire il existe plusieurs instructions :
PULL (lis dans le stack), ARG (recois directement les arguments passés a l'appel).
L'instruction PARSE manipulera les arguments passés à loisir.

```
/* REXX   */                    eclate l'argument en 4 variables
parse arg a.1 a.2 a.3 a.4       ici ARG est un sous paramètre de parse
do i=1 to 4
   say "Argument" i "was:" a.i
end
```

Pour l'exécuter, il est nécessaire de taper son nom puis "alpha beta gamma delta" dans la ligne de commande :

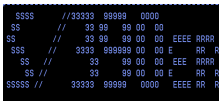
ex arguments alpha beta gamma delta
le programme affiche

Argument 1 was: alpha
Argument 2 was: beta
Argument 3 was: gamma
Argument 4 was: delta

Le programme a séparé la chaîne passée en quatre arguments. La chaîne a été découpée au niveau des blancs en entrée. Si vous oubliez un argument alors, la dernière ligne affichera **Argument 4 was:**
Si vous entrez plus de 4 arguments alors la quatrième ligne affiche les arguments supplémentaires.
Aussi si vous entrez des blancs avant chaque argument, seul le dernier argument affichera SES blancs. On appelle ça "tokenisation".

Il n'y a pas que les arguments passés au moment de l'appel qui peuvent être tokenisés, toutes les variables peuvent être tokenisee. Par exemple remplacez ARG par PULL dans le pgm précédent. Le pgm attend une entrée clavier qui sera tokenisée.
Note, la casse des paramètres passes dépend de l'instruction utilisée pour les récupérer, ainsi ARG recevra TOUT en Majuscule, alors que PARSE ARG gardera la casse.

Les deux utilisations les plus utiles de PARSE sont PARSE VAR et PARSE VALUE qui vous permettent de séparer des donnes arbitraires renvoyées par le pgm ainsi :



Mise en forme de données

```
/* REXX */
     d=date()
     parse var d  day month year
     parse value time() with hour ':' min ':' sec
```

La dernière ligne ci dessus montre une autre manière de séparer les données : elles sont séparées au niveau d'un caractère ' :'.

Testez ces trois lignes pour voir le résultat :

```
/* rexx */
parse value time() with hh ':' MM ':' ss ':'
 say hh ' heure ' MM ' minutes ' SS 'secondes '
```

ici, time sera eclatée au niveau des 2 point (:) .

parse arg first "beta" second "delta"

Ici tout ce qui est avant beta sera dans first et ce qui est entre beta et delta sera dans second si il y a quelque chose après delta ce sera perdu.
Si Beta n'apparait pas dans le string alors tout va dans first et second est vide.
Si Beta seul est code dans le string alors tout ira dans second.

On peut répartir ce qui apparaît entre 2 arguments, ainsi, parse arg "alpha" first second "delta", placera tout ce qui apparaît entre ALPHA et BETA dans FIRST et SECOND.

Un point entre deux variables, parse pull a . c . e jettera la valeur intermédiaire (dans notre cas la deuxième et quatrième valeurs passées seront jetées).
Il est préférable de coder un point après le dernier argument, afin de s'assurer de la validité des arguments passés :

parse pull first second third .

Et cela permet de s'assurer qu'aucun des arguments ne contient de 'blancs' seul le dernier argument pouvant en contenir et ici le dernier est un point.
Enfin, il est possible de délimiter des variables par des champs numériques :

```
/* REXX */
say 'test numeric delimiter enter string '
pull string
parse var string premier 6 second 15 trois 25
say premier 'deux : ' second 'trois : ' trois
```

execution :

```
test numeric delimiter enter string
input =>full monty terrasse test et hop
résultat :
FULL  deux :  MONTY TER trois :  RASSE TEST
```

Premier contient tout jusqu'à la position 5 incluse, second contient depuis la position 6 jusque 14 incluse et trois 15 à 24 incluses

PARSE EXTERNAL restreint L'INPUT AU CLAVIER (voir chapitre sur le stack).

Sous TSO, il est possible de connaître certaines infos concernant la version du REXX utilise ainsi que l'environnement et le REXX appelant en utilisant PARSE :
parse source env ici tout le contenu de source est envoye dans la variable ENV

parse version sysrx versrx rxinst_date ici des renseignements concernant le produit REXX, type de Rexx, version, date d'installation.

Lecture et ecriture de fichiers

Ceci étant une doc TSO-REXX, il m'apparait important de parler de l'ecriture et lecture de fichiers.
Comme en JCL, il est nécessaire d'allouer les fichiers à utiliser, ceci ce fait par la commande **TSO** ALLOC :

```
 "ALLOC FI(EXOOUT) DA('"USERID()".test.exo') new catalog Space(5,2) TRACK ",
"   RECFM(F B) " BLKSIZE(29720) LRECL(80) REUSE".
```

Cette allocation influe sur l'EXECIO, ainsi une alloc de MOD permet de rajouter des enregistrements a la fin du fichier

a. lecture d'un fichier :

« execio *number* diSkR DDNAME *pos* (finis »
Number le nombre de lignes a lire, ou * pour lire tout le fichier
Pos la position de la ligne à lire ou a partir de laquelle on lit. Ce chiffre est facultatif.

exemple : lecture totale d'un fichier fixe existant

```
/* REXX */
 "ALLOC FI(EXOOUT) DA('HH594D.EXOOUT.EX01') shr reuse"
 "execio * diskr exoout (finis"
 do I=1 to queued()
 pull whatever
 say whatever
 end
```

Notez :
Le fichier étant alloue préalablement et rempli, point n'est besoin de lui spécifier les attributs du dit fichier. Dans cet exemple, le fichier contient des numéros de téléphone et des villes :

```
****** ***************************** Top of Data **********************
000001  eric bouladoux          26.20.28.92          lux
000002  rene van broken         03.82.75.62.59               kuntzig
000003  gustave eiffel               03.82.88.72.41  basse ham
****** ***************************** Bottom of Data *******************
```

La fonction Queued() sera vue avec le STACK, mais ici il permet de savoir le nombre de lignes du fichier. Dans la commande EXECIO le paramètre STEM assigne un nom de variable accessible par le 'nom de la variable'**.**'Position dans le stack'.

La position dans le stack est spécifiée par la variable attribuée à la boucle :

```
 "ALLOC FI(EXOOUT) DA('HH594D.EXOOUT.EX01') shr reuse"
"execio * diskr exoout (stem ex. finis"
do I = 1 to ex.0
say ex.i 'is arg ' I
end
```

Après avoir manipulé le stack vous vous apercevrez que l'utilisation de STEM permet de récupérer plus facilement les arguments du fichier.

POSITIONNEMENT DANS LE FICHIER :

1 en utilisant la commande execio avec un positiionnement (pos dans l'exemple de début).

```
"execio 1 diskr inf 2 (stem x. finis"
          say x.0 x.1
```

2 En passant par la technique du SKIP :

```
 "execio 1 diskr inf (skip "                    on skip une (1) ligne
  "execio 1 diskr inf (stem x. finis"           on lit donc la deuxieme (2eme) ligne
                say x.0 x.1
                "free fi(infile)"
```

Dans ces exemples on voit l'utilisation de variables dans des tableaux à deux dimensions (voir page 11), en REXX elles ont le format stem.ext ou stem est le nom de la variable et ext est le positionnement dans le tableau.. Notez que **X.** peut etre remplacer par **X** . Dans ce cas il faudra remplacer **X.i** par **XI** . Mais l'utilisation du format *stem.ext* est plus intuitif.

Il est aussi possible d'acceder à la directory d'un Partitioned Dataset grace a DISKR, voir chapitre particularité MVS.

b.Ecriture d'un fichier :

Comme pour la lecture, la commande d'allocation influence l'accès au fichier, ainsi, une alloc en disp SHR ou OLD écrasera le fichier alors que MOD le mettra à jour par la fin, une alloc en NEW suivi de CATALOG créera et cataloguera le fichier.

```
/* REXX */
 "delete '"userid()".test.exo'"
 "ALLOC FI(EXOOUT) DA('"USERID()".test.exo') new catalog Space(5,2)",
 "TRACK  RECFM(F B) BLKSIZE(27920) LRECL(80)"
parse source hh jjj KKk ll mm nn
queue 'hello ' userid()
queue 'env  ' hh jjj kkk
queue 'env  ' ll mm nn
"execio " queued() " diskW exoout (finis"
```

Ce petit REXX va écrire des informations concernant votre procédure de LOGON et l'EXEC appelant dans un fichier (EXOOUT).
Ici, j'ai utilisé la fonction **queud()** plutot que la paramètre ***** de la commande EXECIO car ce paramètre attend une ligne vide comme ordre de fin d'input (ce qui ne l'empêche pas d'etre écrite dans le fichier)..
Pour vous entraîner, écrivez un REXX pour écrire la date, l'heure, le userid, la version du REXX que vous utilisez ainsi que l'année d'installation dans un fichier NON-existant.

c. Update d'un fichier :

Il est possible d'updater une ligne dans un fichier, pour ce faire il faut le lire et traiter les infos ligne par ligne, grâce a la commande EXECIO DSKRU.
Faire un diskru sur la ligne à lire et garder, Modifier la ligne lue, Faire un diskw d'une ligne.

Format du diskru :
"execio *k* diskru exoout *C* (stem m."
K est le nombre de ligne a updater
C est le numero de ligne a updater

Dans cet exemple on update la dernière ligne

```
/* REXX */
 "ALLOC FI(EXOOUT) DA('"USERID()".test.exo') shr  "
"execio * diskr exoout (finis"
C=queued()
 "execio 1 diskru exoout " C "( stem m."
Say m.1
m.1=strip(m,l,t,'')
M.1=m.1 !!' updated at :' time()
ligne.
push M.1
"execio 1 diskW exoout (finis"
qu'elle a ete lue
say rc ' rc on write '
```

on lit totalement et ferme le fichier
la function queued() nous retourne le nombre de lignes.
on recupere la derniere ligne et on la holde pour update.
colle 'updated ' et l'heure au cul de la

Met la variable M1 dans le stack
elle est ecrite a la meme position

DSKRU est peu employé. Mais diskr et diskw sont communément utilises. Pour vous entraîner, reprenez l'exemple d'écriture et updatez la 3eme ligne par exemple.

Le stack

REXX a un 'data STACK' ou 'pile de données' accessible par les instructions : «PUSH» « QUEUE» et «PULL». PULL (PARSE PULL comme vous le savez) va chercher dans le stack, Les données entres au clavier vont directement dans le stack donc s'il y a quelque chose dans le stack Il vaut mieux utiliser l'instruction « PARSE EXTERNAL » pour limiter l'entrée de données au clavier.

```
/*  REXX */                          Accède le stack  Rexx
queue "Hello!"
parse pull a                         /* a contains "Hello!" */
parse pull b                          /* b is input from the user */
push "67890"
push "12345"
parse pull c                          /* c contains "12345" */
```

La différence entre les instructions PUSH et QUEUE se voit lorsque que ce qui a été pushé et queué est récupéré

«QUEUE» les donnes sont recuperees dans l'ordre ou elles ont été queuees (FIFO)
«PUSH » c'est l'inverse. (LIFO).

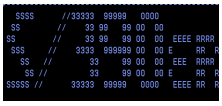
Si la queue (pile) contient un mélange de données (« QUEUED » et PUSHED ») les données «pushed» seront prises les premières.

Le stack est utilise pour passer des données dans un programme REXX, entre programme REXX, a des subroutines, il peut même être utilise pour passer des données entre un programme REXX et un programme COBOL ou assembleur.

Avant d'attaquer le stack, il est préférable de s'assurer qu'il est vide :

```
 /* REXX */
if queued()>0 then do c=1 to queued()
                pull .
                end
```

Avant de continuer avec les routines, deux instructions à utiliser avec ces routines : «NEWSTACK » et « DELSTACK » : elles permettent de créer et détruire un stack au-dessus de celui en cours isolant ainsi les données de l'appelant des données d'exécution de l'appelé.



Interpret

Cette instruction exécute le contenu d'une variable.
Par exemple 2 variables sont passées à un pgm l'une d'elle est une opération arithmétique ('-','+') l'autre est un chiffre, je passe au rexx mes deux valeurs lors de l'appel et le pgm va effectuer l'opération avec un chiffre hardcoder (5) et le chiffre passé :

```
/* rexx */
arg ope num
say ope 'is operation'
oper='j=5'!!ope!!num
interpret oper
say j
```

Dans une instruction IF THEN ELSE on peut assigner une variable si une condition est rencontrée sinon on execute une instruction..
La manipulation de données est l'un des points forts de REXX. Il est donc important de bien le comprendre.

fonctions et sous routines

Cette partie n'est pas importante à ce stade… Pourtant par expérience elle m'apparaît vitale pour la compréhension de l'exécution d'un programme REXX et peut aider a comblé certaines lacunes de l'interpréteur sous TSO.
Il existe trois sortes de fonctions : Built-in (livrées avec le langage), interne et externe.

Built-in :

Le Rexx Summary contient une liste de fonctions disponibles en REXX. Toutes ces fonctions peuvent recevoir des paramètres :

```
/* REXX */
say date("W")',' date()
```

affichera (le vendredi 22 mai 1992) "Friday, 22 May 1992".

Pour appeler une fonction il suffit d'écrire son nom suivi de l'argument ou paramètre entre parenthèses sans blanc entre le nom et la parenthèse ouvrante car le blanc serait considéré comme opérateur de concaténation..

Ici, je vais présenter les plus utiles :
Les fonctions de temps (date, heure, etc…)

Il existe une fonction date dont la réelle utilisation semble mal perçue, la fonction date :
Les classiques :

DATE() retourne DD Month CCYY – pareil que DATE('Normal').
DATE('Base') retourne le nombre de jours écoulés depuis le premier jour de l'an 1.
DATE('Century') retourne le nombre de jours écoulés depuis le début du siècle.
DATE('Days') retourne le nombre de jours écoulés depuis le début de cette année.
DATE('European') retourne le date au format européen.

Pour le reste trouvez vous le manuel REXX d'IBM.
Maintenant voyons un exercice plus marrant :

month = Date('Month', 177 ,'Days')
traduire par retourne moi le *mois* du *177*ème *Jour* de l'année

say DATE('B','07/02/02','E')
Renvoie-moi la ***base*** date de la date passée au format ***européen***

En clair, le format d'appel de la fonction date est : DATE('X','argument', 'y')

X	est ce que l'on veut recevoir (date en Base, US, Euro ?) argument est la valeur a traité
Y	le format de la valeur passée (date en Base, US, Euro ?) .

Ecriture de fonctions

Tout d'abord, la différence entre SUBroutines et functions :
Une 'SUB routine' – sous-routine – n'a pas besoin de valeur derrière son return de fin, mais il peut y en avoir une auquel cas l'appelant recevra le résultat dans la variable result
Elle est appelle par CALL.
Elle partage les variables du programme principal.
Elle ne peut etre qu'interne
ARG en en-tete est optionel

Une function –fonction-
a besoin de valeur derrière le return.
ARG en entete est obligatoire
Peut etre externe ou interne
Elle ne recupere qu'une ligne à la fois (pas de tableau)

REXX permet à la plupart des fonctions d'être appelées comme une sous-routine (CALL)
Que ce soit lors d'un appel a une sous-routine ou une fonction, les arguments passés doivent suivre le même format qu'a la réception.

Fonctions internes
Voici un exemple de fonction interne :

```
/* REXX   */
say "The results are:" square(3) square(5) square(9)
exit
square: /* function to square its argument */
     parse arg in
     return in*in
```

Define a function

on s'arrete ! ! !
le label
on recoit l'argument
on renvoie tout en calculant

l'output aura cette 'tronche' : "The results are: 9 25 81"
Quand REXX trouve l'appel à la fonction (nom(arg)) SQUARE(3) il recherche un label de ce nom – un label commence en colonne 1 et se termine par deux points ':' – il l'exécute jusqu'a la première expression RETURN qui dans notre cas calcul le carre d'un argument en input.
l'EXIT de fin de programme évite a celui ci de continuer dans la fonction après le say.

Ce type de fonction est intéressant si un morceau de code n'est utilise que dans un seul programme et ce plusieurs fois.

```
  /* Define a function with three arguments */
    say "The results are:" conditional(5,"Yes","No") conditional(10,"X","Y")
    exit
    conditional:
/* if the first argument is less than 10 then return the second, else return the third.  */
    parse arg c,x,y
    if c<10 then return x
    else return y
```

Il est important de noter le point suivant : l'appelant doit passer les arguments comme l'appelé les attend.
Ainsi si les arguments sont reçus avec une virgule comme séparateur, l'appelant devra respecter ce format.
De plus, si la sous-routine ou fonction doit être appelée directement, le seul moyen de séparer les arguments passés est l'espace.
Afin de protéger les variables de l'appelant, on peut utiliser l'instruction PROCEDURE qui préparera un nouveau jeu de variables pour l'exécution de l'appelé SEULEMENT, les variables de l'appelant ne sont plus accessibles par l'appelé.

Pour passer des variables (rendre visibles) de l'appelant vers l'appelé on utilise EXPOSE surtout si on a utilise PROCEDURE dans le codage de l'appelé. :

```
 /* REXX */
    parse pull x .
    say x"!="factorial(x)
    exit
```

Calculate factorial x, that is, 1*2*3* ... *x

```
    factorial:
/* REXX */
    procedure
    parse arg p
    if p<3 then return p
    else return factorial(p-1) * p
```

Calcule la factorielle de l'argument

Un nouveau paramètre est apparut dans l'exemple précédent, PROCEDURE dont le but est de protéger les variables du programme appelant des instructions de l'appelé.
Ce type de paramètre est surtout utile lors du codage de fonctions internes

Fonctions Externes :

Ce type de fonction se code **comme une fonction interne**, la différence étant que l'appelant ne contient pas le code appelé, mais celui ci est stocké dans un fichier à part (généralement un PDS alloué au même DDNAME que l'appelant – SYSPROC, voire SYSEXEC).
Il permet de palier aux 'oublis' d'IBM. Ex. : une fonction linein pour lire un certain numéro de ligne dans un fichier sans passer par l'instruction EXECIO.
IBM conseille de permettre aux fonctions externes de pouvoir être appelées soit en tant que fonction soit en tant que Sous-routine comme c'est le cas pour leurs fonctions built-in.
Les fonctions seront utilisées surtout pour coder des utilitaires à utilisation récurante (formatage de date, fonction linein, lineout, etc…) appelable par d'autre EXEC, alors que les sous-routines seront codées à usage interne (dans le programme).
De plus, il est conseillé d'avoir un standard de codage de fonctions au niveau de la réception des paramètres, à savoir soit les arguments sont séparés par une virgule (le choix d'IBM.) soit ils sont
séparés par un blanc.

Enfin, pour en terminer avec les fonctions, différentes invocations de la même fonction :

Fonction FUNC01 :

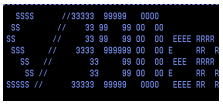
```
 /* REXX */
 TRACE OFF
 ARG VAL1,VAL2
JJ=VAL1 * VAL2
 RETURN JJ
```

Appelée comme fonction :

```
/* REXX */
say func01(5,2)
```

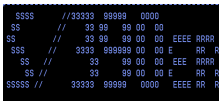
Appelée comme une sous-routine (externe) :

```
 /* REXX */
call func01 5,2
 say result
```



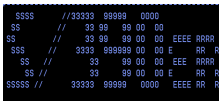
Liste de fonctions : (1/3)

num = **ABS**(number) valeur absolue d'un nombre
str = **ADDRESS**() interroge l'environnement courant
num = **ARG**() retourne le nombre d'arguments
val = **ARG**(n) retourne le nth argument
bit = **ARG**(n,'Exists') l'argument numero N existe ?
bit = **ARG**(n,'Omitted') l'argument N a ete omis ?
str = **BITAND**(str1[,[str2][,pad]]) Logically and strings
str = **BITOR**(str1[,[str2][,pad]]) Logically or strings
str = **BITXOR**(str1[,[str2][,pad]]) Logically xor strings
str = **B2X**(str) Binaire vers Hex (0F=00001111)
str = **CENTER**(str,length[,pad]) centrer str
str = **CENTRE**(str,length[,pad]) CENTER pour les anglais (hunour cowlishawesque)
str = **CHARIN**([name][,[start][,length]]) lecture de name, start –a partir de, length – sur-
num = **CHAROUT**([name][,[start][,length]]) ecrire dans name , aprtir de, sur
bit = **CHARS**([name]) il y a t'il quelquechose dans name ?
cix = **COMPARE**(str1,str2[,pad]) comparaison de str1 et str2 0 ou premiere différence
str = **CONDITION**('Condition') nom de la condition capturée
str = **CONDITION**(['Instruction']) retourne le type (CALL ou Signal)
str = **CONDITION**('Description') Description or null
str = **CONDITION**('Status') ON, OFF, or DELAY
str = **COPIES**(str,n) N copies de str
rc = **CSL**('rtnname retcode parms') Callable services library
num = **C2D**(str[,n]) Caaractere vers decimal
str = **C2X**(str) Caractère vers Hexadecimal
str = **DATATYPE**(str) NUM ou CHAR
bit = **DATATYPE**(str,type) STR correspond a type ?
str = **DATE**() date du jour(dd Mmm yyyy)
str = **DATE**(dopt) Date info (voir l'exemple date)
... = **DB...**(str,...) 13 DBCS support functions
str = **DELSTR**(str,n) Delete str de n jussqu'a la fin
str = **DELSTR**(str,n,length) Delete str a partir de n sur length
str = **DELWORD**(str,n) Delete le mot de n jussqu'a la fin
str = **DELWORD**(str,n,length) Delete wix a partir de n sur length
str = **DIAG**(hex[?][,data][,data]...) ? displays diagnostics
str = **DIAGRC**(hex[?][,data][,data]...) rtourne CP codes (VM)
str = **D2C**(wholenumber[,n]) Decimal vers Caractère
str = **D2X**(wholenumber[,n]) Decimal vers Hexadecimal
str = **ERRORTEXT**(n) texte msg d'erreur associé avec le RC n (0-99)
num = **EXTERNALS**() See PARSE EXTERNAL
wix = **FIND**(str,arg) 0=not found; preferez WORDPOS
str = **FORM**() verifie le format numérique
str = **FORMAT**(num[,[before][,[after]]]) Around decimal place
str = **FUZZ**() Query NUMERIC FUZZ
cix = **INDEX**(haystack,needle[,start]) Default start=1; prefer POS
str = **INSERT**(new,str[,[n][,[length][,pad]]]) Insert after cix n
str = **JUSTIFY**(str,length[,pad]) Right-left justify
cix = **LASTPOS**(needle,haystack[,start]) POS de droite a gauche.
str = **LEFT**(str,length[,pad]) aligne a gauche
num = **LENGTH**(str) longueur de str.
str = **LINEIN**([name][,[line][,count]]) list la ligne de name
bit = **LINEOUT**([name][,[string][,line]]) ecrit dans name
num = **LINES**([name]) Input nombre de lignes restante en input
num = **LINESIZE**() longueur de ligne du terminal (VM)
num = **MAX**(num[,num...]) Maximum (jusque 10 chiffres)
num = **MIN**(num[,num...]) Minimum (jusque 10 chiffres)



Liste de fonctions : (2/3)

str = **OVERLAY**(new,str[,[n][,[length][,pad]]]) remplace str par new a partir de n sur lentgh
cix = **POS**(substr,str[,n]) position de substr dans str a partir de n 0=not found
num = **QUEUED**() nombre de lignes dans la pile
num = **RANDOM**() nombre entire au hasard 0-999
num = **RANDOM**([min][,[max][,seed]]) nombres entier se suivant au hasard
str = **REVERSE**(str) retourne str
str = **RIGHT**(str,length[,pad]) aligne sur la droite
num = **SIGN**(num) signe de num -1, 0, or 1
num = **SOURCELINE**() nombre de lignes dans fichier exec appelant
str = **SOURCELINE**(n) nieme ligne du fichier appelant
str = **SPACE**(str[,[n][,pad]]) arrance les blancs
hex = **STORAGE**() taille de la memoire virtuelle en hex
hex = **STORAGE**(address,length) lecture memoire virtuelle
hex = **STORAGE**(address,length,data) ecriture dans la memoire virtuelle
str = **STREAM**(name[,'State']) etat dustream
str = **STREAM**(name,'Description') State of stream, more detail
str = **STREAM**(name,'Command',cmd) appliquer la commande au stream
str = **STRIP**(str[,[option][,char]]) L, T, ou default=Both retirer caractere de str (ou blanc –defaut).
str = **SUBSTR**(str,firstcix[,[length][,pad]]) Substring
str = **SUBWORD**(str,firstwix[,length]) Def length=rest of string
str = **SYMBOL**(name) State: BAD, VAR, or LIT
str = **TIME**() heure (hh:mm:ss)
str = **TIME**(topt) info heure
str = **TRACE**() montre les options de trace
str = **TRACE**(option) change l'option de trace
str = **TRANSLATE**(str[,[new][,[old][,pad]]]) Map old to new
num = **TRUNC**(num[,n]) tronque num a n decimal
str = **USERID**()logon userid
val = **VALUE**(name) interroge sur la valeur de name
val = **VALUE**(name,val) change la valeur de name
val = **VALUE**(name[,val],selector [groupname]) GLOBAL/LASTING/SESSION var
cix = **VERIFY**(str,okchars[,['Nomatch'],start]) First bad cix; 0=all ok
cix = **VERIFY**(str,okchars,'Match'[,start]) First good cix; 0=none
str = **WORD**(str,wix) extrait le nieme mot
cix = **WORDINDEX**(str,n) position du nieme mot dans str
num = **WORDLENGTH**(str,n) retourne la longueur du nieme mot de str
wix = **WORDPOS**(word,str[,start]) retourne la position du mot word dans str (depuis start).



Liste de fonctions : (3/3)

Il existe 6 fonctions NON-SAA disponibles sous TSO (elles ne sont pas forcement disponibles dans d'autres environnements IBM). :
EXTERNALS, FIND, INDEX, JUSTIFY, LINESIZE, and USERID. If you plan to write REXX programs that run on other SAA environments, note that these functions are not available to all the environments.

EXTERNALS - returns the number of elements in the terminal input buffer (always 0 for TSO/E)
FIND - returns the word number of a substring in its string (for SAA use **WORDPOS**)
INDEX - returns the position of one string within another string (for SAA use **POS**)
JUSTIFY - returns a formatted string from blank-delimited words to justify both margins
LINESIZE - returns the current terminal line width minus 1
USERID - returns the TSO/E user ID

TSO/E External Functions

GETMSG – recupère les messages d'une session console TSO/E CONSOLE dans des variables
LISTDSI – retourne des information sur des fichiers
MSG – retourne la valeur de display des message TSO (peut la modifier) ON/OFF.
MVSVAR – retourne des informations systèmes mvs dans des variables.
OUTTRAP – sert a capturer l'output de commandes TSO dans des variables
PROMPT - returns / sets the current value of TSO/E command prompting within an exec
SETLANG - returns / sets a code that indicates the langageof displayed REXX messages
STORAGE – affiche la memoire virtuelle comprise entre 2 valeurs (modif possible).
SYSCPUS – retourne des informations sur les CPUs dans une variablecomposée.
SYSDSN retourne des informations sur l'existance d'un dataset.
SYSVAR – retourne des informations sur une variable spécifié.

Vous pouvez utiliser les functions MVSVAR, SETLANG, STORAGE and SYSCPUS dans n'mporte quel espace addresse, TSO/E et non-TSO/E., pour les autres, il est nécessaire d'executer dans un espace adresse TSO/E.

DEBUGGING and tracing errors

Execution de commandes

REXX peut être utilisé sur de nombreuses plate-formes (Windows, VM, UNIX, MVS, VSE).
Voici ce qui se passe quand un REXX est exécuté :
Quand REXX (l'interpréteur) rencontre une ligne qui ne ressemble pas à une instruction (REXX) ou à un assignement de valeur a une variable (c='2') il evalue la ligne et l'envoie à l'environnement (sous TSO cela peut être TSO, ISPF, MVS, ou tout autre environnement appelé par la commande ADDRESS).
Par exemple :

```
/* REXX */
say 'addressing ispexec '
 address ispexec
« Display panel(TESTP) »
```

Execution

addressing ispexec
Affiche le panel TESTP

```
/* REXX */
say 'addressing ispexec '
  « Display panel(TESTP) »
```

addressing ispexec
IKJ56500I COMMAND DISPLAY NOT FOUND
3 *-* "DISPLAY PANEL(TESTP)"
+++ RC(-3) +++

Dans le 2eme cas la commande n'est pas trouvée dans l'interpréteur REXX ni dans les commandes TSO.
Les environnements disponibles en REXX sous MVS sont : MVS, TSO, ISPF (ISPEXEC), l'éditeur ISPF (ISREDIT), la linklib (LINK).
L'instruction ADDRESS() permet de connaître l'environnement courant..

Signal

Les COBOListes peuvent comparer cette instruction au GOTO, en effet, le programme 'saute' directement au label spécifié. Un label est aussi utilisé dans le codage de FONCTIONS INTERNES, et c'est d'ailleurs le même type – LABEL: en colonne 1-. Le hic, c'est que Signal nettoie derrière lui, en clair si on 'signal ' dans une structure de contrôle, il est impossible d'y revenir. Voilà pourquoi SIGNAL est surtout utilisé pour trapper les erreurs afin de sortir proprement du programme.
Il est possible de limiter le 'trap' à certaines conditions :
"syntax" (Erreur de syntaxe), "error" (toute commande d'environnement ne reournant pas un code retour Zero), "halt" (interruption d'execution – PA1-) et "novalue" (un symbole n'a pas de valeur).

La capture de ces erreurs se déclenche par **signal on <condition>**

« s 'éteint » par **signal off <condition>**

Quand l'une de ces conditions survient, le programme pointe immédiatement sur le label dont le nom suit la clause signal. le Trapping n'est plus possible, sauf si vous coder un nouveau signal.
il est possible d'avoir un nom de label différent du nom de la condition grâce au paramètre NAME de SIGNAL. :

signal on syntax name my_label

A chaque SIGNAL, la variable SIGL est initialisée au numéro de ligne de l'instruction en erreur. Si le SIGNAL
Est le fait d'une 'Capture d'erreur' (error trap), la variable RC est aussi initialisée (avec le code erreur).

```
 /* REXX */
say "Press Control-C to halt"
     signal on halt
     do i=1
       say i
       do 10000
       end
     end
 EXIT
     halt:
     say "Ouch!"
     say "Died at line" sigl
EXIT
```

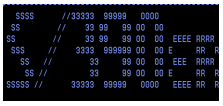
This program goes on forever until someone stops it.

Tracing

Cette fonctionalité permet de suivre l'execution d'un programme REXX en 'direct live'.

trace r
Chaque instruction est affichée avant d'être executée : Elle affiche les résultats de tout calcul

trace ?a
Affiche l'instruction suivante et pause. Il faut appuyer sur ENTER pour l'exécuter.
Idéal pour vérifier les variables et les commandes.



REXX sur MVS

REXX sur MVS remplace la CLIST. Certaines personnes hurleront au mensonge, et pourtant c'est la vérité.

Tout comme la CLIST, REXX accède à ISPF par l'interface ISPEXEC ou ISREDIT pour l'éditeur.
Tout comme en CLIST, il est possible d'exécuter des taches comme on le ferait en JCL.
Mais le fait que REXX soit le langage SAA d'IBM a forcer celui ci à produire d'autres interfaces, ainsi une Interface DB2 existe-t-elle (DSNREXX) et depuis 1995 REXX peut être exécuter dans un environnement CICS.
En fait, une interface SDSF serait la bienvenue…. (je plaisante à peine).
Nous verrons certaines fonctions REXX particulières à MVS plus loin .
En resume, MVS a plusieurs environnments accessibles par REXX :

ISPF	ISPEXEC	
Editeur	ISREDIT	
LINK	LINK	
TSO	TSO	Défaut
MVS	MVS	
DB2	DSNREXX	

Jusqu'ici tous les exemples étaient testés en tapant EX devant, maintenant nous allons voir comment le rendre accessible comme une commande TSO normale.
Toutes les clists et les REXX de votre site se trouvant dans une concaténation de librairies SYSPROC voire SYSEXEC, il va falloir ajouter votre librairie à cette concaténation ! ! !

Pour ce faire voici deux programmes rexx que vous allez recopier dans votre PDS :

```
 /* REXX */
"alloc fi(sysuexec) da('votre-pds') shr reuse"
 "altlib activate user(exec)"
my_bib='votre-pds'
v=concat('sysproc',N)
 "ALLOC FI(exo1) DA('"USERID()".new.exo1') new catalog Space(5,2)",
 "TRACK  RECFM(F B a) BLKSIZE(27920) LRECL(80) REUSE"
 Queue " free fi(sysproc)"
 Queue " alloc fi(sysproc) shr reuse da( + "
do I=1 to v
queue  "  '"concat('sysproc',i)!! "' +"
end
queue "  '"!!my_bib!!"' +"
queue "    ) "
"execio " queued() "diskw exo1 (finis"
"free fi(exo1)"
```

page suivante, la fonction concat appelée par le precedent programme :

```
/* REXX */
arg dd,p
"newstack"
address tso
   CALL OUTTRAP "LINE.","*"
"listalc sta"
do I=2 to line.0
parse var line.i name sta
  if name=dd then do j=i-1 to line.0
                    parse var line.j name sta
                    select
                      when name=dd then nop
                      when  pos('.',name)=0 & sta ^='' then do
                                                            n=queued()-1
                                                            leave i
                                                           end
                      when  pos('.',name)>0 then queue name
                      otherwise  nop
                    end
                  end
   else iterate
end
   CALL OUTTRAP "off"
   select
   when queued()=0 then do
                         call bye_bye
                         return -1
                         end
   when queued()=1 then n=queued()
   when N='N' then n=queued()
   otherwise n=n
   end
 if arg(2)='N' then do
                    dsname=n
                    call bye_bye
                    return dsname
                    end
  else if P>n then do
                   call bYE_BYE
                   return -2
                   end
     do i=1 to n
        pull name
        if i=p then dsname=name
     end
    call bye_bye
  return dsname
bye_bye:
"delstack"
   return
```

Pour vous éclairer un peu :

Concat émet une commande TSO LISTALC STA, celle ci liste toutes les BIB. allouées à votre USERID, On se ‘balade’ dans l’OUTPUT pour trouver un DDNAME sysproc et on compte le nombre de librairies.

Le deuxième concat compte-lui aussi, mais arrive au chiffre passé dans l’appel de fonction retourne la bibliothèque à l’appelant.

Ces deux REXX, créer un fichier d’allocation de bibliothèques au DDNAME SYSPROC

Procédure à suivre :

1		Copier CONCAT (la fonction doit s'appeler comme ça car elle est Hardcodée dans le REXX. Copier le premier petit REXX dans une bibliothèque
2	Votre-pds	le pds ou vous avez stocké la fonction concat. Note les deux variables peuvent contenir un nom de bib différent, j'ai simplement considéré que le pds contenant vos premiers REXX serait celui que vous garderez comme bib REXX.
3	MY_lib	Le pds ou vous souhaiter stocker TOUS VOS programmes REXX voir note pour VOTRE-PDS.
4		Exécuter le PETIT REXX en tapant EX devant son nom (en EDIT, comme au début)
5		le fichier 'votre-userid.new.exo1' est prêt, sous ISPF 3.4 (DSLIST) taper EX devant…..

Vos REXX sont accessible par la ligne de commande (préfixe TSO).

Chaque fois que vous vous loggerez il faudra re-faire la manipulation.
(si vous vous re-loggez en cours de journée seule l'étape 5 est necessaire.)

Vous êtes maintenant paré pour faire mumuse avec REXX….

Je vais maintenant vous montrer comment 'attaquer' différents environnements avec REXX.
En effet, l'un des gros avantages de REXX sur MVS est l'accessibilité aux environnements.
Nous allons voir ISPF… :

- display de panel
- appel de programmes (SELECT PGM) pour utilisation SDSF
- macros isredit

Le but étant de vous donner quelques conseils d'utilisations ISPF en REXX.

ISPF et REXX (the deadly combination…)

ISPF tel que vous le voyez aujourd'hui (mars 2004) est en fait composé de deux composants : Dans les années mi 70 (19, c'était au siècle dernier) IBM sort SPF, un 'package' équipé de fonctions de Dialogue (offrant entre autre un support plein écran) et d'un éditeur ultra puissant. Par la suite les développeurs ont développé des applications en utilisant les fonctions de dialogue de SPF.
En 1981, le support de dialogue d'SPF devient dialogue manager, la partie éditeur devient PDF.
Le tout devenant ISPF….
En 1982 ISPF contient seulement Dialogue manager PDF étant une application a part tournant sous ISPF.
En 1987 ISPF est intégré à la politique SAA d'IBM.
Donc nous avons bien deux produits séparés, ISPF (ISPEXEC) et PDF (ISREDIT).

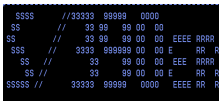
Afin d'accéder à ces deux produits il est nécessaire de le faire savoir à l'interpréteur. Ce que fait la commande ADDRESS.

Syntaxes :	ADDRESS ISPEXEC	(connect to ISPF)
	ADDRESS ISREDIT	(connect to PDF)
	ADDRESS()	(what am I connected to ?)

Juste pour vous amuser dans un membre de votre pds, créer un membre CONNECT et écrivez :

```
/* REXX */
un=address()
address ispexec                    resultat :
deux = address()                   TSO ISPEXEC ISREDIT TSO
address isredit                    ***
trois = address()
address TSO
quatre = address()
say un deux trois quatre
```

Juste pour votre info, Il est possible d'adresser un environnement inconnu essayez d'écrire un ordre d'address vers un environnement bidon, et la ligne d'en dessous affichez l'environnement (ADDRESS()) et vous verrez.
Cela permet de laisser la liberté aux programmeurs COBOL et ASSEMBLEUR d'écrire des environnements E.G. : SDSF (il n'existe pas encore). Cela a permit à des sociétés d'écrire des environnements de support VSAM (REXXTOOLS par exemple)



ISPF et REXX

FAQ (bidons) ISPF :
Comment fait ISPF pour savoir dans quelles bibliothèques se trouvent les panneaux ISPF ?
Ce brave ISPF alloue toute les librairies de panneaux ISPF à un même DDNAME : ISPPLIB.

Comment connaître ces librairies ?
Reprenez les deux rexx, et dans l'appel de concat remplacer sysproc par ISPPLIB exécutez puis
éditez le fichier résultat (seul hic vous aurez votre pds en dernier fichier alloué) ceci étant une Vérification, N'EXECUTEZ PAS le fichier résultat.

ISPF s'alloue-t-il plusieurs librairies ?
Oui, des librairies de programmes, de squelettes (JCL), de messages, de tables, etc….
Dans l'ordre les DDNAMES sont :
ISPLLIB, ISPSLIB, ISPMLIB, ISPTLIB.
Note sur les ddnames de tables, il y'en a effectivement 2 : ISPTABL (sortie) et ISPTLIB (entrée).

Maintenant attaquons les panneaux ISPF.

Panels ISPF
Il faut que vos panneaux soient créer dans un pds alloués à ISPPLIB…, OU BIEN, on peut s'allouer une bibliothèque de panneaux pour la durée de l'exec. Grâce à la commande LIBDEF :
«libdef » DDNAME «DATASET ID(‘» your-dataset' «)»
la suite du REXX « libdef » DDNAME (le même DDNAME qu'a your-dataset).
Bien sûr il faudra précéder votre libdef d'une instruction ‘ address ispexec'

```
/* REXX */
 address ispexec
 "libdef ispplib DAtaset id(‘vot-lib-de-rexx') "
 "control errors return"
 "display panel(TEStP)"
say name ‘is name u entered'
 "libdef ispplib"
```

on ajoute not' lib à not' concatenation d'panneaux
On pourra passer les RC ISPF a REXX.
on affiche not' ed beaux panel
on y a rentrer queuq' chose… et on affiche
on r'met la concatnation comme c'est qu'elleétait..

et voila le panel TESTP :

```
)ATTR DEFAULT(%+_)
)BODY WINDOW(45,10)
%- test prod.rexx version
 %SELECTION ==>_ZCMD                +
 + TEST         _name               +
 +-----------------------------------------+
)INIT
.HELP=LISTHELP
)PROC
)END
```

l'affichage

```
              - test prod.rexx version
 SELECTION ==>
TEST
 ----------------------------------------
```

Si lorsque le panneau s'affiche vous entrez quelque chose, des que vous appuyez sur ENTER Vous le verrez s'afficher (variable NAME) sinon NAME s'affichera.
RAPPEL : LES VARIABLES REXX SONT NON TYPEES.

)ATTR initialise les défauts (ici % text protected + text non protected, _ input)
)BODY est le corps (! !) et window indique qu'il s'affiche dans une fenêtre de 45 colonnes
de large sur 10 lignes de profondeurs.
Les 4 lignes du dessus sont de l'affichage l'input ZCMD est obligatoire (au moins
un champ input ZCMD Dans une panel)
)init c'est l'initialisation des zones input AU MOMENT de l'affichage (ici rien)
)PROC traitement des données APRES l'affichage (ici rien).

Un petit exercice EXO 1 :
Reprenez le REXX et le panel, rajouter le code nécessaire au deux pour entrer le nom, prénom téléphone du bureau
Et de la maison de personnes.
Pour vous aider voila l'affichage :

```
- test prod.rexx version
 le nom :                    prenom :
 TEL of :              tel Home      :
-----------------------------------------
```

et il faudra allouer un fichier en lrecl 80 réponse en annexe.

TABLES ISPF
Voyons brièvement les tables ISPF (seulement lecture à partir d'un fichier) :
Les tables peuvent être considérées comme des tableaux à deux dimensions : colonnes et lignes.
L'intersection d'une colonne et d'une ligne correspond à une variable dont le contenu peut être traité.
Les tables peuvent être temporaires ou permanentes (ici nous n'étudions que les temporaires.), la différence étant que l'on réécrit une table permanent pour la sauvegardée (paramètre WRITE sur l'ordre TBCREATE, et ordre TBCLOSE de la table à la fin du traitement).
Les panels pour ce service (TBDISPL) sont différents de ce que l'on a vu précédemment :
Il y a une)model section qui définit l'affichage des variables :

```
)MODEL
£entree                                                    +
```

Après le tbcreate, il est nécessaire d'initialiser les variables de la ligne traitée, puis d'ajouter cette ligne à la table créée.

Il y a donc 2 mouvements pour ajouter une ligne a une table :
Initialisation des Variables de la section model du panel et TBADD de votre table :

Var1=whatever
Var2=whatever
« TBADD TABLE »
Les 2 variables doivent se trouver dans la section MODEL de votre panel !

une ligne peut contenir plusieurs variables. (correspondant à nos colonnes).
Ici un nouvel exemple :

```
/* REXX */
address ispexec
"libdef ispplib DAtaset id('hh594d.exo.rexx') "
"control errors return"
"TBcreate phoneL names(entree) nowrite"
say rc 'on tbcreate '
address tso
"alloc fi(phone) da('HH594D.PHONE.EXO') shr reuse"
"execio * diskr phone (finis"
"free fi(phone)"
do i=1 to queued()
     pull entree
     address ispexec
    "tbadd phonel"
    say rc 'tbadd'
     "tbbottom  phonel"
     say rc 'tbtop'
end
address ispexec
 "tbtop  phonel"
 "tbdispl phonel panel(testp1)"
  if RC=8 then call bye_bye
   else nop
 address ispexec
 "libdef ispplib"
   exit
 bye_bye:
 "tbend phonel"
 return
```

le panel TESTP1 :

```
)ATTR DEFAULT(%+_)
 £ TYPE(OUTPUT) intens(low) color(blue)
 \ TYPE(OUTPUT)
 ä TYPE(INPUT)
 )BODY EXPAND(//)
 %-------------------------------------------------------------------------
 %SELECTION ==>_ZCMD                                    £zdate

 %  nom        prenom          tel office     tel home
 +-------------------------------------------------------------------------
 )MODEL
 £entree
 )INIT
 &ZTDMARK='-------------------------------------------------------------+
   ----- S/390er exo -'
 )PROC
 )END
```

la variable Entrée est définie en OUTPUT. Elle devra être initialisée dans l'EXEC par la commande ENTRÉE = *value* puis les commandes ISPF TBADD et TBBOTOM devront être passées afin de rajouter la ligne dans la table puis de la déplacer un cran vers le bas.. &ZTDMARK affiche une dernière ligne.

Voila l'affichage :

```
------------------------------------------------------------- Row 1 to 3 of 3
SELECTION ==>                                          03/03/18

  nom        prenom          tel office     tel home
-------------------------------------------------------------------------------

  LOMME     RENE        45.10.23.63     64.25.36.96
  KENT     JON        44.99.2.25.56     26.20.25.98
  JACKSON     RICHARD        47.89.5.69.21     21.68.95.47
------------------------------------------------------------- S/390er exo -
```

L'affichage est merdique, mais il suffit de rajouter un peu de code pour formater les variables Ainsi, on peut modifier l'ordre TBCREATE comme suit :

```
"TBcreate phoneL names(name first telof telhome) nowrite"
```

plus loin il faudra lui définir 4 nouvelles variables, pour ce faire nous utilisons l'ordre REXX PARSE que l'on peut traduire par 'éclater' : **parse var entree name first telof telhome** juste en dessous le pull ENTRÉE (logique, pusique c'est justement la variable ENTRÉE que l'on 'éclate'.

ISPF et REXX

enfin il faut modifier le nom de variable dans notre panel (section')model') comme suit :

```
)MODEL
£name          £first          £telof      £telhome        +
```

et on recommence…
et voila l'affichage :

```
------------------------------------------------------------- Row 1 to 3 of 3
SELECTION ==>                                        03/03/19

  nom            prenom          tel office        tel home
-------------------------------------------------------------------------------
LOMME            RENE            45.10.23.63       64.25.36.96
KENT             JON             44.99.2.25.56             26.20.25.98
JACKSON          RICHARD         47.89.5.69.21             21.68.95.47
------------------------------------------------------------- S/390er exo –
```

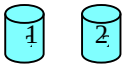
Nous venons de voir les tables ISPF et les panneaux ISPF par REXX…

Nous allons continuer avec ISPF et les variables.

Variables ISPF

A chaque appel au service SELECT un pool 'fonction' est crée. Un REXX ne créé pas de pool fonction, A moins de faire un appel au service SELECT.
Il existe plusieurs pools ISPF disposé en hiérarchie ou même couche :
Le plus haut est le PROFILE POOL (garder entre chaque session ispf, c'est le fichier ISPPROF)
Puis c'est le Shared POOL, qui dure le temps de la session et accessible par toutes les fonctions.
Puis c'est le function pool – créé par le service SELECT, valide le temps de la commande.

ici, un ordre ISPF SELECT a été passe, le pool 3 est le function pool accessible par VGET sans paramètre supplémentaire et tous les displays récupèreront leurs variables dans ce pool.



Ici un exec REXX a été lance comme une commande TSO, il n'a accès qu'au 2 pools ispf habituels (SHARED et PROFILE)

Note quand un ordre SELECT est exécuté, il y a recopie des variables système du shared vers le function, puis celle ci sont actualisées dans le function pool.
Ainsi lorsqu'un REXX exécute un ordre VGET (ZDATE) c'est le shared pool qui est accédé,
A moins que vous n'ajoutiez PROFILE à la fin dans ce cas c'est le profile pool qui est cherché.
Les Services disponibles par REXX pour accéder aux variables sont VGET et VPUT :
Address ispexec « VGET (zdate) » retournera la date ISPFdu SHARED pool.
Maintenant les instructions suivantes vont créer une variable JJKKLL dans le shared pool

```
JJKKLL='test by me'
Address ispexec « VPUT (JJKKLL) »
JJKKLL = ''
Address ispexec « VGET (JJKKLL) »
Say  JJKKLL
```

ecrase la valeur de jjkkll

retourne test by me

Donc quand on lance un REXX par la commande ISPF :
ADDRESS ISPEXEC « SELECT CMD(%cmd-rexx) »
On creer un nouveau function pool en plus du shared et du profile, et une commande VGET (VAR) ira chercher dans ce pool, si on ajoute SHARED a la fin on ira dans le shared pool (un étage plus bas). Si on ajoute PROFILE, on ira encore un etage plus bas.

Exercice de VGET :

```
/* REXX */
address ispexec
"vget (zdate ztime zsysid)"
 say zdate ztime zsysid
```

Sauvegardez cela dans votre bib d'exercices et noter le systeme id – ZSYSID. Les 2 autres noms de variables parlent d'eux même.

ISPF les services de fichiers : LM services

Ces services permettent la manipulation (update, lecture, création, etc…) de fichiers, de membres (si le fichier est un PDS), et de 'dataset list' (e.g. 3.4) sous ISPF.
Ces services travaillent TOUS sur un DATAID ou un LISTID(dataset list), à savoir une variable contenant un numero. Seul deux services ne l'utilisent pas : les services d'initialisation de DATAID ou de LISTID, ceux la l'attribuent.
Il existe plusieurs types de services LM :
LM sur fichier
LMD sur concaténation
LMM sur membres (PDS)

LMINIT : initialisation de dataid pour fichier (Sequentiel ou PDS), il peut se faire sur un dataset ou sur un DDNAME
« lminit dataid(did) dataset('votre-dsn') enq(shr) » ENQ : SHR/SHRW/EXCLU/MOD
'lminit dataid(lst) DDNAME('DDNAME') enq(SHR)'

ATTENTION dans notre exemple le DATAID n'est pas entre apostrophe, alors que dans les services attaquent ce meme dataid il l'est. C'est parce que dans les autres services c'est UNE VARIABLE (initialisée par LMINIT).

```
ddname='ISPCTL0'                                    Le défaut
 ADDRESS ispexec                                    ne pas oublier, on demande ISPF !
'control errors return'
'lminit dataid(lst) DDNAME('DDNAME') enq(SHR)'      on initialise notre variable…
'EDIT dataid('lst')'                                et on l'edite – pas de lmopen dans ce cas.
'lmfree dataid('lst')'                              ne pas oublier, relacher le dataid ! ! !
exit 1
```

LMOPEN : Ouvre le composant associé au DATAID
"lmopen dataid("did") option(output) OPTION : OUTPUT/INPUT
LA variable de DATAID est ici entre apostrophe (on récupère la valeur de cette variable).
OUTPUT on va écrire dans le fichier
INPUT on affichera des infos de ce fichiers.
LMFREE : Libère le dataid.
"lmfree dataid("DID")"
exemple d'utilisation de services LM :

```
arg dsn memb
address ispexec
"control errors return"
"lminit dataid(DID) dataset('"dsn"') enq(SHR)"
"lmopen dataid("DID")"
"lmmfind dataid("DID") member("memb") lrecl(lrecl) stats(yes)"
if  RC = 0 then say zllib zluser memb
 else say memb 'not found !!'
"lmfree dataid("DID")"
exit 1
```

ISPF le service SELECT

Ce service permet d'exécuter des programmes, d'appeler des panels, d'exécuter des commandes (TSO), avec possibilité de créer un nouveau 'function pool' de variables.
Il est étudié très brièvement, en effet, il existe des docs très complètes sur ISPF.
Pour appeler une commande :

ADDRESS ISPEXEC « select cmd(%your-cmd) »	Executer votre commande
ADDRESS ISPEXEC « select cmd(%your-cmd &ZPARM) »	Executer votre commande en passant un paramètre.
ADDRESS ISPEXEC « select cmd(%your-cmd) newappl(*prof*)»	Executer comme precedemment mais en utilisant un profile specifique (le defaut est ISP)
ADDRESS ISPEXEC « select cmd(%your-cmd) newpool »	EXEcuter une commande en créeant un nouveau Shared pool. Le shared pool de départ sera restaurer a la fin de l'application

ATTENTION ! !
La commande address peut être utilisée de deux manières, one-time shot et le normal.
Ce que j'appelle le One-time shot se code comme suit :
ADDRESS *ENV* « instructions »
Il ne s'applique qu'a l'instruction qui suit sur la ligne. On peut coder plusieurs one time shot à la suite, mais aux niveau des ressources, si on veut coder plusieurs instructions d'environnement (ici ISPF), il vaut mieux utiliser la technique 'normal' :
ADDRESS *env*
« instruction>
« instruction »
address *old_env*

ISPF le service LIBDEF

Le service libdef sert à définir des librairies à ISPF. Il existe 2 types de définitions : Statique et Applicative.

-statique

La définition est faite lors de l'appel (ISPSTART) d'ISPF. En général cette définition inclue les librairies standards d'IBM et les librairies du site.

-applicative

Elle se fait au moment de l'appel a une fonction. Dans ce cas les librairies ainsi définit se

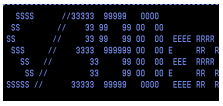
trouvent avant les librairies statiques (définies lors du logon).

Si on veut avoir une concatenation avec les librairies Statiques au dessus, il faut utiliser ces DDNAMEs Lors de l'ordre LIBDEF :

- ISPMUSR User message library
- ISPPUSR User panel library
- ISPSUSR User skelttons library
- ISPTUSR User Table (input) library
- ISPTABU User table (output) library
- ISPFILE User output file tailoring library
- ISPLUSR User load library

Après utilisation, il faut executer un LIBDEF DDNAME a vide :
Si on a executer un LIBDEF sur un ISPPLIB, il faut en faire un a vide à la fin de l'exec. :
ADDRESS ISPEXEC « libdef ISPPLIB ».

Bon maintenant on voit l'editeur, à savoir ISREDIT.



ISPF ISREDIT

Alors que pour executer les 2 derniers REXXes vous n'aviez effectivement pas besoin d'allouer votre bibliothèque a sysproc ou sysexec (vous pouviez taper dans votre bib en face de votre EXEC), ISREDIT exige que cela soit fait ! !
ISREDIT permet d'acceder à l'editeur ISPF, ATTENTION nous parlons de l'editeur –service EDIT- et non du service BROWSE.
Il est ainsi possible de recuperer l'emplacement ou se trouve le curseur (ligne et colonne), de recuperer dans une variable le nom du DATASET et membre ou l'on se trouve.
Les macros ISREDIT (et oui ca s'appele comme ca ! ! !) commencent toutes par l'ordre MACRO
Les macros recoivent des arguments par l'ORDRE MACRO et N'oubliez pas de passer la commande ADDRESS ISREDIT au début !

voici le REXX :

```
/* REXX */
/* ISTEST */
address isredit
"macro (un,deux)"
"(dsn) = dataset"
"(cha) = data_changed"
address tso
say dsn  cha       un deux
```

```
taper ISTEST H j
HH594D.EXO.REXX NO  H j

        taper ISTEST
        HH594D.EXO.REXX NO
```

maintenant amusez vous bien ! ! ! pour continuer il existe des sites internet et meme les docs IBM
peuvent vous aider ! !

REXX et DB2

Pour commencer, sur la ligne de commande ISPF, taper TSO DSN vous êtes dans l'utilitaire de commandes DB2.
Cet utilitaire permet d'executer des commands DB2 telles que –dis util –term util.
Il est possible de passer et de recevoir des paramètres a cet utilitaire par REXX en 'queuant' les ordres de l'utilitaire et en utilisant la command outtrap pour trapper le resultat. Un petit exemple :

```
/* REXX */
util='*'
queuE "-dis util("util") "
queuE "end "
call outtrap "line.","*"
   address tso
"DSN "
call outtrap "off"
say line.0
do I=1 to line.0
say line.i
end
```

vous pouvez utiliser cet exemple comme base pour differents outils DB2.

Nous allons maintenant voir la technique (la première utilisée) pour passer des ordres SQL depuis un fichier en EDIT (en clair, Il y'aura de la macro a coder) : ATTENTION, je ne vais pas vous donner le CODE, mais comment vous devez proceder pour ecrire le code (le code se trouve à WWW.S390er.COM dans REXX et EXEC).

La première technique pour passer des ordres SQL à DB2 fut de passer par le programme DSNTEP2 (standard IBM) et de l'executer comme du JCL, à savoir, on alloue les librairies dont il a besoin et on execute le programme. Dans ce cas il vous faudra :
connaitre le plan associé au programme DSNTEP2
queuer les ordres SQL et les écrires dans votre fichier SYSIN
Puis il suffit de 'queuer' les commande DB2 :
RUN PROGRAM(DSNTEP2) PLAN(le-plan-de-votre-site)
END

Et de coder la commande suivante : « DSN S(« votre-db2-sys ») »

Déjà on s'apercoit que l'on peut écrire du REXX comme on écrirait du JCL.

La deuxième solution est l'environnement DSNREXX, ce qui rejouit nos amis cobolistes, car on s'amuse a préparer, declarer, et fetcher des curseurs et autres output descriptor…
DSNREX est un environnement disponible avec la Version 5 de DB2 for OS/390.

Tout d'abord, il faut connaitre le nom de son système DB2.
Ensuite on peut attaquer le codage :

1 tout d'abord s'assurer que l'environnement d'un point de vue REXX est là (commande subcom)
2 sinon on l'ajoute (commande ADD) aux environnement existant – c'est du REXX pur.
3 maintenant on va se connecter a DB2 (commande connect SSID)
4 on assigne a une variable l'instruction SQL a executer .
5 il faut 'declarer' un curseur … voir plus bas

la suite logique diffère s'il sagit d'un SELECT ou d'un INSERT

SELECT

6 il faut preparer l'instruction avec un 'output descriptor'
7 il faut ouvrir le curseur de l'étape 5. (v5 OK MAIS db2 v7 retourne 1 en RC)
8 il faut recuperer le curseur en utilisant le descripteur de l'etape 6
9 il faut traiter l'output du descripteur voir plus bas
10 il faut fermer le curseur .

INSERT

6 il faut preparer l'instruction avec un 'output descriptor'
7 il faut executer l'instruction

11 il faut deleter l'environnement REXX ajouter a l'étape 2.
12 Et voila

La declaration de curseur (plus chiante que la déclaration des revenus) :
En SQL/REXX il existe un jeu prédéfinis de noms pour les curseurs (cursor) et pour les instructions préparées (Prepared statement)

Les curseurs :
C1 a C100
Noms prédéfinis pour DECLARE CURSOR, OPEN, CLOSE et FETCH

C1 à C50 pour les cursors définis sans l'option WITH HOLD.
C51 à c100 pour les cursors définis avec WITH HOLD.

Tous les curseurs sont définis avec l'option WITH RETURN donc n'importe quel nom de curseur peut Être utilisé pour récuperer les resultats d'une 'STORED PROCEDURE' écrite en REXX.

Les 'prepared statements' :

S1 to S100
Noms prédéfinis pour les instructions DECLARE STATEMENT, PREPARE, DESCRIBE et EXECUTE

Attention :
les ordres SQL passés a la fonction prepare NE DOIVENT PAS AVOIR DE POINT VIRGULE (;) A LA FIN !!
Les variables assignées en REXX sont accessible par DB2 . Lors de lappel DB2 elles doivent etre préfixées de deux points (:)

Il y a une différence de codage entre un SELECT et un INSERT :
Pour un SELECT
Il faut ouvrir le curseur et le fermer une fois terminé.
il y a necessité d'avour un output descriptor dans lequel seront envoyés les ligne de resultat de la commande.
L'output descriptor (SQLDA) est une variable obligatoire lors d'un SELECT. Le nom est indifférrent, mais elle est 'déclaré' dans le PREPARE STATEMENT.
La variable *variablename*.sqld contient le nombre DE LIGNES du contenus.
Les données de l'output se trouvent dans *variablename*.x.sqldata, X étant le numéro de ligne.
Il y a une ligne d'output par colonnes passées dans le SELECT, il faudra donc concatener l'output du SQLDATA.

Et n'oublions pas la fameuse SQLCA (SQL communication AREA) créée à chaque appel de procédure.
Pour vous donner une idée de développement voila un REXX et ISPF pour afficher les entrées de la syscopy dont le HLQ de fichier commence par ce que vous specifiez en premier argument et dont la date correspond au 2ème argument passé

Rappel : les 3 variables de la section model seront initialisées, puis un TBADD sera fait de la table.
Le chapitre suivant traite de l'appel de programmes ou utilitaires en REXX (comme en JCL).

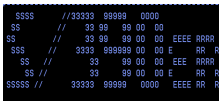
REXX comme du JCL

Il est possible d'exécuter du REXX comme du JCL, permettant d'exécuter les utilitaires IBM.
Le codage est le même qu'en JCL, il est juste nécessaire de savoir ou se trouve le module
Attention, il est aussi nécessaire de faire la distinction d'utilisation en mode batch ou mode 'online' car on pourra faire des call tso ou des select pgm ISPF (c'est le cas de SORT et SDSF).
Les allocations sont ainsi faites explicitement en REXX par la commande TSO ALLOC, et la carte EXEC du JCL correspond à un CALL TSO voire un SELECT PGM d'ISPF.
Les differents fichiers ne posent pas de problèmes excepté peut être le fichier sysin,car il est souvent coder 'instream'. Mais là aussi une petite ruse permet de passer outre, il s'agit de 'queuer les ordres sysin en sens inverse et d'executer la commande REXX EXECIO DSKW sur la sysin. Et après on appelle le programme.

Voici un exemple de sort :

```
/* REXX */
"ALLOC FI(SYSOUT) DUMMY"
"ALLOC FI(SYSPRINT) DA(*)"
  "ALLOC FI(SORTIN) DA('"USERID()".TEST.SDSF') SHR REUSE"
"ALLOC FI(SORTOUT) DA('"USERID()".LIST.S0PJDN')  SPACE(15,6) CYL
    RECFM(F B A) NEW CATALOG LRECL(240) BLKSIZE(24000)"
  "ALLOC FI(SORTWK01) DUMMY"
 CMD.1=" OMIT COND=(1,9,CH,EQ,C'         ')"
 CMD.2=" SORT FIELDS=COPY"
 DO J=1 TO 2
 PUSH CMD.J
 END
"ALLOC FI(SYSIN) UNIT(SYSDA)  NEW CATALOG DA('"USERID()".TEMP.SYSIN')
  TRACK SPACE(5 2) RECFM(F B) LRECL(80) BLKSIZE(8000)"
  "EXECIO" QUEUED() "DISKW SYSIN (FINIS"
  "DELSTACK"
  ADDRESS ISPEXEC "SELECT PGM(SORT) "
  SAY RC 'ON SORT '
   ADDRESS TSO
  "FREE FI(SYSIN SYSPRINT SYSOUT SORTIN SORTOUT SORTWK01)"
 /*    "DELETE '"USERID()".LIST.S0PJDN'" */
  "DELETE '"USERID()".TEMP.SYSIN'"
```

Il est donc important de bien maitriser la commande ALLOC de TSO, ainsi que le format de commande du CALL et du SELECT d'ISPF.
De plus, certaines commandes AMS (utilitaire IDCAMS) sont appelables directement depuis TSO, c'est le cas de REPRO avec tous les paramètres optionnels de l'utilitaire.
Noter qu'il est possible d'allouer la sysin en disposition NEW DELETE elle sera ainsi supprimée au prochain FREE.



REXX comme du JCL

CALL TSO ou SELECT PGM d'ISPF ?

Dans les 2 utilitaires suscités (SORT et SDSF), il y a quelques différences d'utilisation.
Ainsi un programme REXX appelant SDSF est executé dans un environment ISPF devra faire appel au services d'ISPF (SELECT).
Si le REXX appelant le module SDSF est appelé en batch il faut alors passer par le CALL TSO.
Comme en JCL, et quelquesoit le choix de l'appel du programme il est necessaire d'allouer les fichiers avant l'execution du programme ou de l'utilitaire.
Nous allons donc revoir la commande ALLOC de TSO.

Tout d'abord il est necessaire de savoir quel type d'allocation vous voulez faire, a savoir les fichiers à alloueer sont ils output, input, sysout, sysprint ?
Pour ce faire il est donc necessaire de connaître le programme, c'est pour cela que je vais utiliser
les utilitaires standards MVS d'IBM – ils se trouvent sur tous les sites et VOUS ETES SUPOSE les connaître sinon reportez vous a la documentation utilitaires d'IBM.
Pour illustrer la commande allocation je vais utiliser l'utiliser ICEGENER d'IBM (IEBGENER si vous preferez ce sont les mêmes) .
Voici un JCL executant ICEGENER :

```
.
//SORTTEVS  JOB (CA1),'TMS UTILITY',MSGLEVEL=(1,1),
//            CLASS=A,MSGCLASS=X,NOTIFY=&SYSUID
//SORT1    EXEC PGM=ICEGENER
//SYSOUT     DD SYSOUT=*
//SYSUT1   DD DSN=&SYSUID..EXO.ICE01,DISP=SHR
//SYSUT2    DD  DSN=&SYSUID..EXO.ICE02,DISP=(,CATLG),
//     UNIT=SYSDA,SPACE=(CYL,(9,2),RLSE)
//SYSPRINT   DD SYSOUT=*
//SYSIN      DD DUMMY
```

Et, a titre de comparaison, le REXX correspondant

```
/* REXX */
"alloc fi(sysut1) da('"userid()".exo.ice01') shr reuse "
  "ALLOC FI(sysut2) DA('"USERID()".exo.ice02') NEW CATALOG SPACE(9,2)",
  "CYL RECFM(F B A) BLKSIZE(13300) LRECL(133) REUSE"
  "alloc fi(sysprint) da(*)"
  "alloc fi(sysout) dummy"
  "alloc fi(sysin) dummy"
  address ispexec "select pgm(icegener)"
  say rc
    "free fi(sysout sysprint sysut1 sysut2)"
```

et recherchez les commandes allocation (ALLOC) dans une doc TSO avant d'aller a la page suivante.

REXX comme du JCL

Les commandes d'allocation des fichiers SYSUT et SYSIN sont traduites literralement (one to one), seules les allocations sysout et sysprint sont différentes :
La SYSOUT a été 'dummier' pour eviter une allocation plus compliqué pour un fichier dont on ne se sert pas de toute facon, mais elle aurait pu etre allouée a un fichier sysout :
« alloc fi(sysout) sysout(X) » .
La SYSPRINT a été codée pour envoyer les output à l'écran. ATTENTION ce type d'alloc ne marchera que pour les fichiers output.
L'appel du programme est par ISPF, une variante aurait été le call direct par MVS :
Remplacez **address ispexec "select pgm(icegener)"** par **address link « icegner »**.
Et surtout rajouter immediatement derrière le call icegener une commande ADDRESS TSO sous peine d'abend S0C4.

La commande ADDRESS LINK vous permet d'acceder au programmes se trouvant dans la LINKLIB ce qui est le cas des utilitaires IBM, mais les programmes de production ne s'y trouvent certainement pas et si vous voulez faire des calls a des programmes faisant eux même appels à des routines se trouvant dans des librairies différentes…
Certes, dans ce cas, le REXX apparaît limité mais il existe la solution ISPF.
Tout d'abord une chose importante a savoir LA BIBLIOTHEQUE DE LOAD MODULES ACCESSIBLES sous ISPF est réferencée par le DDNAME ISPLLIB !
Il vous faut donc faire connaître a ISPF votre lib de load et quelle commande vous permet cela ?
LIBDEF evidemment : "libdef ispLlib DAtaset id('"userid()".exo.LOAD') "
N'oubliez pas d'ajouter les librairies des differents load appellés en interne !!!
n'oubliez pas – non plus - a la fin de votre exec une ligne "libdef ISPLLIB » .

alors CALL TSO ou ISPF ?

si vous utilisez vos execs en batch comme dit au début utilisez le CALL TSO avec une restriction toutefois au niveau du passaage de paramètres (voir chapitre cobol).
C'est plus facile d'utilisation.
Si vous utilisez vos programmes ONLINE (genre appel dans REXX) utilisez l'appel par ISPF meme si cela semble galerant au premier abord, notamment au niveau du passage de paramètres, puisque l'on ne peut en passer qu'un seul.

Nous allons donc voir un chapitre d'appel cobol en REXX …. En fait le lien entre COBOL et REXX. Et l'utilisation d'ISPF pour appeler vos programmes.

REXX et COBOL :

Ce pettit chapitre n'a pas pour but de vous apprendre le cobol, il est donc preferable d'en avoir des notions. Mais je vais indiquer le passage de paramètre.
COBOL recoit les parametres dans la linkage section {attention il s'agit de passage de parametre et non de control statement de la sysin} il vous faut donc autant de variable par paramètre passès, l'ordre des paramètres etant spécifiè par la reception au niveau de la PROCEDURE DIVISION.
Mais, il y a une limitation au niveau du passage de paramétres, a savoir on ne peut en passer qu'un et il ne peut mesurer plus de 100 caractères de long que ce soit par appel ISPF ou TSO..
Un petit exemple de programme accepatant un paramètre et son REXX d'accompagnement

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. RXTST.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 OPE1        PIC 9(3).
 01 OPER        PIC X(1).
 01 OPE2        PIC 9(3).
 01 REST        PIC 9(6).
 LINKAGE SECTION.
 01 PARM1       PIC X(12).
 PROCEDURE DIVISION
    USING PARM1.
    DISPLAY 'PARM1 :'PARM1.
    MOVE PARM1(3:3) TO OPE1.
    MOVE PARM1(6:1) TO OPER.
    MOVE PARM1(7:3) TO OPE2.
    DISPLAY OPER ' IS OPERATION'.
    EVALUATE OPER
    WHEN '+'  ADD OPE1 OPE2 GIVING REST
    WHEN '+'  ADD OPE1 OPE2 GIVING REST
    WHEN '-'  SUBTRACT OPE1 FROM OPE2 GIVING REST
    WHEN OTHER DISPLAY 'WRONG OPER ON ' OPE1 ' ' OPE2 ' ' OPER
    END-EVALUATE.
    DISPLAY 'RESULTAT : ' REST .
    STOP RUN.
```

Le REXX :

```
/* REXX */                                          Ce rexx appelle notre programme en lui passant
"free fi(sysout)"                                   un paramètre de 12 char max. 2 chiffres de 3 char
de long avec
"alloc fi(sysout) da(*)"                            un signe + ou – entre deux
parm1='052+020'                                     ce paramètre est une operation (+ ou -)
address tso                                         autrement affichage message (vor programme)
" call 'hh594d.exo.load(rxtst)' '"parm1"' "         appel du prog
say rc
"free fi(sysout)"
```

REXX et COBOL

Note sur le programme : le signe plus ou moins est en fait le signe du second chiffre.
Certes j'aurais pu faire mieux, mais j'ai ecrit ca vite fait su'le gaz merci de ne pas m'en tenir rigueur (c'est just un petit exemple de passage de paramétre et non d'opérations en COBOL) .

Pour passer des paramètres de cobol vers REXX le mieux est l'utilisation des services de variables d'ISPF que je vais vous présenter.

COBOL et ISPF :

Avant de tourner un programme COBOL il faut le compiler. Contrairement a ce que l'on pourrait croire, il n'ya pas besoin d'ajouter de librairies spéciales dans le job de compilation afin de tourner un programme COBOL, il est juste necessaire d'avoir un evironnement ISPF présent ainsi en 'online' (avec un USERID TSO) il faudra lancer ce programme **avec la commande select d'ispf**, cela semble logique, mais on l'oublie tellement facilement…
Si on doit executer ce programme en batch il faudra utiliser l'utilitaire TSO IKJEFT01 avec les librairies ISPF et la commande ISPF SELECT PGM initialisera un environnment sur base des librairies de votre JCL.
En cobol pour acceder aux Variables d'ISPF il faut avoir accés aux services ISPF, par le call à ISPLINK ou ISPEXEC. Il est nécessaire de connaître la syntaxe des deux, car les serivice faisant le lien entre variables COBOL et ISPF ne sont utilisable que par ISPLINK, autrement on peut utiliser ISPEXEC, dont le format me semble plus familier.
Syntaxe des 2 formats :
ISPLINK : **CALL 'ISPLINK' USING VDEF N-PARM REST C-CHAR QL**
Ici le format des variables passées dans la working-storage :

```
01 REST       PIC 9(6).
01 N-PARM     PIC X(4) VALUE 'RESX'.
01 C-CHAR     PIC X(4) VALUE 'CHAR'.
01 C-FIX      PIC X(5) VALUE 'FIXED'.
01 QL         PIC 9(6) COMP VALUE 6.
01 VDEF       PIC X(7) VALUE 'VDEFINE'.
```

La variable VDEF contient la commande. La variable N-PARM contient le NOM DE LA VARIABLE ISPF (sur laquelle il faudra faire un VGET après l'appel au programme). REST est la variable contenant le resultat. C-CHAR contient le format utilisé. QL contient la longueur des données de la variable REST.

Cette commande devra etre executer avant toute execution d'instruction MOVE TO dans la variable REST.

COBOL et ISPF suite:

ISPEXEC : **CALL 'ISPEXEC' USING BUF-LEN BUF**.

Ici le format des variables (en working-storage) passées a la commande :

01 BUF	PIC X(80).
01 BUF-LEN	PIC S9(9) BINARY.

La variable BUF va contenir les conmandes (ici 'VPUT (RESX)') et BUF-LEN la longueur de la commande ici 11 (cette valeur peut être plus grande mais ne peut être plus petite), il est donc nécessaire de faire un MOVE de la commande et de la longueur dans ces deux variables respectives..

En bref, il faut definir pour chaque variable accessible par ISPF, une variable contenant le nom ISPF ainsi que la variable contenant les données.

Le programme se trouve sur le CD (RXTST3 avec son REXX RXTST3).

Nous allons voir le dernier chapitre sur les spécialités MVS .

Particularités MVS.

Il existe certaines fonctions seulement disponible sur MVS et parfois seulement depuis la version OS/390. et d'utres qui existent en system 390 ou 370.
C'est le cas de la fonction STORAGE qui vous permet d'afficher des zones memoires sur votre ecran.
Ex :

```
/* REXX */
CVT      = C2D(STORAGE(10,4))
JESCT    = C2D(STORAGE(D2X(CVT + 296),4))
RESUCB   = C2D(STORAGE(D2X(JESCT + 4),4))
IPLVOL   = STORAGE(D2X(RESUCB + 28),6)
Say IPLVOL ' ç  volume d'ipl'
```

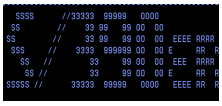
Ici, on se 'balade' dans la mémoire pour recuperer le volume d'ipl du système.
Cet exemple me permet de vous montrer l'utilisation de la fonction storage. Cette fonction éxige une bonne connaissance des zones mémoire du systéme, donc apprenez par cœur le fameux 'P.O.P' d'IBM … Toutfois sachez que la CVT est la zone de départ de toute balade en mémoire

Une fonction disponible avec TSO/E SYSDSN permet de savoir l'etat d'un dataset :
G=sysdsn('votre dataset') afficher G (say G) il sera :
OK, DATASET NOT FOUND, ou INVALID DATASETNAME

Une autre permet de recuper des info sur un dataset dans des variables de type SYS*NNNN* :
LISTDSI :

```
/* REXX */
DSN='votre-dataset'
X=listdsi(dsn) NORECALL
Say sysdsname ' on ' sysunit ' type ' sysdsorg
Say sysrecfm ' ' syslrecl ' ' sysalloc
```

L'option NORECALL permet d'eviter de recaller le dataset.



Particularité MVS

La fonction OUTTRAP permet de récuperer l'output d'une commande TSO (en fait, celle n'utilisant pas la macro assembleur TPUT pour afficher le texte) . Ainsi on peut recuperer l'output d'un listcat

```
/*-rexx  */
 arg dsn
 CALL OUTTRAP "LINE.","*"
"listc ent("dsn") all"
CALL OUTTRAP "OFF"
argum=translate(line.4,' ','-')
st=pos('CREATION',argum)+8
argum=substr(argum,st,20)
argum=strip(argum,B)
yyyy=substr(argum,1,4)
ddd=substr(argum,6,3)
d=yyyy!!ddd
say      d
```

Une autre fonction particulière à MVS, MVSVAR, permet de recuperer des informations système
Tel que le nombre cpu, la date d'IPL, etc…

Voici un REXX citant quelques unes des possibilités de cette fonction :

```
/* REXX */
        say "         MVSVAR output:"
        say " "
        say "sysdfp------------("mvsvar('sysdfp')")"
        say "sysmvs------------("mvsvar('sysmvs')")"
        say "sysname-----------("mvsvar('sysname')")"
        say "sysseclab---------("mvsvar('sysseclab')")"
        say "syssmfid----------("mvsvar('syssmfid')")"
        say "syssms------------("mvsvar('syssms')")"
        say "sysclone----------("mvsvar('sysclone')")"
        say "sysplex-----------("mvsvar('sysplex')")"
        say "symdef,sysclone---("mvsvar('symdef','sysclone')")"
        say "symdef,sysname----("mvsvar('symdef','sysname')")"
         if substr(mvsvar('sysname'),1,3) == "CSW" then do
           say " "
           say "CSW defined:"
           say "  symdef,instname-("mvsvar('symdef','instname')")"
           say "  symdef,psuffix--("mvsvar('symdef','psuffix')")"
           say " "
           end
         say " ### the next one may fail with some ugly msgs ### "
        say "sysappclu---------("mvsvar('sysappclu')")"
```

ici on recupère le jobname executant le REXX :

```
/* REXX */
 jobname = MVSVAR('SYMDEF','JOBNAME')
say jobname
```

on affiche des symboles defines dans la SYS1.PARMLIBdans le membre IEASYMxx (selon le système) :

```
/* REXX *** */
SAY 'JOBNAME  = 'MVSVAR('SYMDEF','JOBNAME')
/* GMT TIME */
SAY 'YYMMDD   = 'MVSVAR('SYMDEF','YYMMDD')
SAY 'DAY      = 'MVSVAR('SYMDEF','DAY')
SAY 'HR       = 'MVSVAR('SYMDEF','HR')
SAY 'JDAY     = 'MVSVAR('SYMDEF','JDAY')
SAY 'MIN      = 'MVSVAR('SYMDEF','MIN')
SAY 'MON      = 'MVSVAR('SYMDEF','MON')
SAY 'SEC      = 'MVSVAR('SYMDEF','SEC')
SAY 'HHMMSS   = 'MVSVAR('SYMDEF','HHMMSS')
SAY 'WDAY     = 'MVSVAR('SYMDEF','WDAY')
SAY 'YR2      = 'MVSVAR('SYMDEF','YR2')
SAY 'YR4      = 'MVSVAR('SYMDEF','YR4')
/* LOCAL TIME */
SAY 'LYYMMDD  = 'MVSVAR('SYMDEF','LYYMMDD')
SAY 'LDAY     = 'MVSVAR('SYMDEF','LDAY')
SAY 'LHR      = 'MVSVAR('SYMDEF','LHR')
SAY 'LJDAY    = 'MVSVAR('SYMDEF','LJDAY')
SAY 'LMIN     = 'MVSVAR('SYMDEF','LMIN')
SAY 'LMON     = 'MVSVAR('SYMDEF','LMON')
SAY 'LSEC     = 'MVSVAR('SYMDEF','LSEC')
SAY 'LHHMMSS  = 'MVSVAR('SYMDEF','LHHMMSS')
SAY 'LWDAY    = 'MVSVAR('SYMDEF','LWDAY')
SAY 'LYR2    = 'MVSVAR('SYMDEF','LYR2')
SAY 'LYR4    = 'MVSVAR('SYMDEF','LYR4')
/* ADD YOUR OWN SYMBOLIC VARIABLES HERE */
SAY 'SYSR2    = 'MVSVAR('SYMDEF','SYSR2')
EXIT 0
```

La variable SYMDEF étant Symbol Definition

sachez que le programme EZACSMF1 permet de faire les substitutions suscitées par batch :

```
//EZACSMF1 JOB (CA1),'TMS UTILITY',MSGLEVEL=(1,1),REGION=0M,
//            CLASS=A,MSGCLASS=X,NOTIFY=&SYSUID
//SYM EXEC PGM=EZACFSM1
//* SYMBOL SUBSTITUTION
//SYSIN DD *
DA OJOB
PRINT FILE MSGFILE
FIND '&jobname'
++?
FIND 'JESYSMSG'
++XC
//SYSOUT DD UNIT=SYSDA,SPACE=(CYL,(1,1)),RECFM=FB,LRECL=80,
//  BLKSIZE=0,DSN=&&ISFIN,DISP=(,PASS)
//SDSF  EXEC PGM=SDSF,DYNAMNBR=32,REGION=1024K,TIME=5
//* CAPTURE THE JES2 MESSAGES
//MSGFILE  DD  DISP=SHR,DSN=&SYSUID..TEMP.S0PJDN
//ISFOUT   DD  DUMMY
//ISFIN    DD  DISP=(OLD,DELETE),DSN=&&ISFIN
```

on va créer un fichier avec des commandes SDSF sur notre jobname courant et on execute un step SDSF pour trapper l'output du job courant.

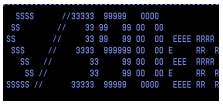
PDS directory list :

Une particularité purement MVS, les partitioned datasets. Il est possbile de lire la directory d'un PDS (mais pas un PDS-E) grace a la commande DISKR, il faut tout d'abord s'allouer le fichier en disposition SHR lrecl = 256 blksize = 256 .

Un petit exemple pour afficher les noms des membres d'un PDS:

```
/* REXX */
"alloC fi(AREAD) dataset('hh594d.prod.rexx') dsorg(PS) lrecl(256) recfm(F b) shr reuse"
"execio * diskr aread (stem dir. finis"
"FREE fi(aread)"
do i = 1 to dir.0
parse var dir.i bl 3 block
vv=length(block)-2
do n=1 to vv by 42
mname=substr(block,n,8)
  say mname datatype(mname)
end
end
```

On peut afficher d'autres infos (genre creation date, last modified date) mais il faut s'amuser à utiliser les fonctions C2D.



MVS OpenEdition

ADDRESS SYSCALL et ADDRESS SH vous donnent la possibilité d'acceder aux services Unix sur MVS.
L'avantage etant d'acceder à la riche palette des fonctions C que je ne connais pas par cœur.
ADDRESS SYSCALL est a utiliser pour les REXX executer sous TSO. Si vous executer votre REXX sous OpenEdition, il vous faudra utiliser ADDRESS SH.
Ici nous nous contenterons d'executer sous TSO.
Je vais vous expliquer comment acceder a la partie Unix de MVS ainsi que vous donner 2 rexx en exemple pour montrer l'interaction entre REXX/MVS et Unix services. Il vous faudra
chercher dans la documentation IBM les fonctions C vous interressant

Pour appeler l'environnement il est nécessaire de le 'mettre à ON' par la commande : call syscalls 'ON'.
Puis pour que REXX n'interprete pas ses commandes (voir chapitre XX) il faut passer ADDRESS 'SYSCALL' (notez le format de la commande différent de REXX classic) .

Voici les pièges les plus classiques lors de l'utilisation de cet environnement :

- Les noms externes sont en capitale. Ainsi l'appel a rxgdgv(0) plantera. RXGDGV(0) marchera.
- L'environnement SYSCALL initialise des variables qui peuvent avoir le meme nom que vos propres variables.
- La methode pour passer des varaibles au syscall diffère de REXX classic

```
/* REXX mount example */
path = /tmp/testpath/mountpoint
dsn  = 'HFS.TEST.MOUNT'
address SYSCALL "mount (path) (dsn) HFS "MTM_RDWR
say 'Rc='rc 'Retval='retval 'Errno='errno
```

- Pour les nouveaux executables il sera peut être nécessaire de faire un 'CHMOD' afin de les rendre executables
- Les variables mises a jour par la commande USS sont directement accessibles au programme REXX :

```
/* REXX */
call syscalls 'ON'
address 'SYSCALL'
'getgrent bb.'
say bb.gr_name   bb.gr_members
say retval
```

ç la variable BB sera accessible directement
ç noter le format des différeentes variables

Certaines variables se voient attributer des extensions (voir audessus) seules accessibles (la variable seule ne donne rien)

- Les codes retour des services UNIX sont ccessibles par ERRNO et RETVAL,

les pièges les plus classiques lors de l'utilisation de cet environnement (suite) :

- RC code retour passé a REXX par USS. :
  - 0 ok RAS
  - > 0 USS specific error dans ce cas, *errno* et *retval* sont mises a jour
  - -3 l'envuronnement SYSCALL est inexistant (appel SYSCALL fait ?).
  - -20 Commande passée non existante
  - - 2X paramétre X passé (X = position du parm) mauvais ou inexistant
  - < 0 Erreur négative retournée par le REXX préprocessor.

Suivent deux petits rexx pour vous démontrer l'avantage des functions C.

```
/* REXX */
 arg msg
 ms=substr(msg,1,30)
 call syscalls 'ON'
 address 'SYSCALL'
 'open /dev/console' O_WRONLY 666
 'write' RETVAL 'ms'
  exit
```

voila un REXX pour lister ce qu'il y a d'ac cessible depuis votre root directory

```
/* REXX */
call SYSCALLS 'ON'
address 'SYSCALL'
'readdir / rt.'
say rt.0   errno retval
  do i=1 to rt.0
  say rt.i
  end
```

et pour terminer, un REXX plutot fun : l'appel de la fonction C sleep en REXX :

```
arg n
select
when n='?' then call help_menu
when datatype(n) ^= NUM then call help_menu
otherwise nop
end
call syscalls 'ON'
address syscall 'sleep ' n
return n
help_menu:
say 'arg can only be numeric'
say 'arg is number of secs to sleep'
exit 1
```

à appeler par la commande (en REXX) J=doze(05) – sleep de 5 secondes

A l'Aide !!!

Ou cherchez de l'aide ?
Les documentations d'ibm sur votre site, voire l'utilitaire de chicago-soft QUICK-REF si vous avez des messages d'insultes…
Autrement si vous vous baladez sur INTERNET essayez le site d'un certain GABE (US) : WWW.THEAMERICANPROGRAMMER.COM.
Le site d'IBM evidemment, le site de thierry fallissard (le french Z/OS expert) il vous faudra chercher avec google car au moment ou j'ecris son site n'est plus accessible (ENG)..
Essayez WWW.AEMVS.COM plutot généraliste que spécifique mais il peut aider dans certaine situation d'erreur (genre VSAM ou certains RC.) (FR).
Un site pour connaître les nouveautés et astuces de différents produits d'IBM : WWW.SHARE.ORG
Et il y'a les forums d'aide :

WWW.MVSHELP.COM : forum ET documentation IBM sur les principaux produits. Site US.
WWW.IBMMAINFRAMES.COM : Forum d'aide, doc IBM et leurs propres produits. Site NDIEN.

Pour MVSOE lire la doc IBM 'Using REXX and Z/OS UNIX Services', il n'y a en effet pas trop d'exemple sur le NET (essayez www.billlalonde.tripod.com il a quelques exemples interessants).

Au revoir et merci

Ce petit bouquin est maintenent terminé. J'espère ne pas avoir été trop ennuyeux (j'en entends ronfler ?) et interessant
Pour les ceusses que cela interresse, entre deux injections d'héroine je maintiens un petit site ou vous trouverez des rexx et autres outils a mettre sur votre lieu de travail…
WWW.S390ER.COM

Et pis… Merci a un certain brun-bellut (nicolas de son prénom), pour sa relecture et surtout sa correction (vu que je viens de la banlieue -93- et que j'ai une tendance a écrire comme je parle Imaginez le bouquin sans la correction de nico, bon c'aurait été marrant je vous l'accorde, mais ce n'était pas le but du livre).

Gabriel Gabe Gargiulo pour ces petites docs et surtout m'avoir mis le pied a l'etrier du REXX et
meme du bouquin… allez le voir WWW.THEAMERICANPROGRAMMER.COM.

Mike Cowlishaw le Créateur de REXX (devinez pourquoi…).

Thierry Falissard pour sa connaissance systéme et son support lors de la mise en route de mes deux sites.

Si vous avez des questions RDV sur mon site et cliquez sur WEBMEISTER.
Merci de votre support et que le dieu cowlishaw et ses apotres soient avec vous.