textLib

un ensemble de routines pour manipuler texte et tables en Applescript

Guide d'utilisation

G. Coray

mars 2022

1. Introduction

1.1 Motivation

TextLib est un ensemble de routines de conversion et d'édition de texte, utiles dans une application Applescript. Quelques applications programmées avec textLib sont l'évaluation de stratégies dans les jeux comme le SUDOKU, une calculatrice de bureau, un test d'inversion de matrices et une sorte de tableur.

Dans toute application, la présentation à l'utilisateur des données ou de resultats requiert soit une interface graphique soit la conversion de données en texte clair. textLib propose des fenêtres et des fichiers de texte comme moyen de communiquer des resultats à l'utilisateur et de les conserver sur l'ordinateur. Il s'appuie pour cela sur l'application *textEdit*, pilotable en Applescript et disponible en configuration standard de Mac OS X.

Les structures de données prises en compte sont les nombres, chaînes de caractères, ensembles, fichiers et divers types de liste. En particulier les entrées/sorties de texte, l'édition et le formatage de données typiques des opérations matricielles ou des tableurs, de même que l'analyse et l'évaluation d'expressions arithmétiques pourront s'appuyer sur un jeu de primitives de traitement de listes détaillées ci-après.

Une motivation importante pour rassembler ces primitives de base sous la forme d'une bibliothèque de routines est d'éviter les sources d'ambiguité propres à Applescript dues à sa syntaxe trop libre et à une sémantique trop sensible aux propriétés globales telles que les *"Applescript's text item delimiters*" ou les options *"case sensitive"*.

Le style adopté ici est proche du paradigme procédural que l'on retrouve dans Algol, Pascal, Python ou C. Chaque routine a un nombre fixe de paramètres positionnels, plus faciles à documenter et à comprendre.
Les applications telles qu'un grapheur ou un générateur de fractales requièrent, en plus de textLib, le module complémentaire graphLib.

En fait textLib (et son complément graphLib, turtleLib ou EuclidLib) pourraient fournir un jeu d'outils appropriés au prototypage ou au test d'algorithmes séquentiels, récursifs, tels que ceux enseignés dans un cours d'introduction à la programmation.

1.2 Contenu

textLib peut être groupé en quatre sections ou modules, chacune apportant les outils de programmation adaptés à une structure de données spécifique.

Les caractères, chaînes et sous-chaînes
-- chaînes de caractères, occurrences et substitution
-- concaténation de chaînes et listes simples de symboles
-- ensembles de caractères representés par des chaînes

Les listes, tables et leur conversion en (ou à partir de) texte
-- listes, opérations de base, sous-liste, sélection, offsets
-- tableaux rectangulaires, transposés, concaténés, colonnes
-- la conversion de texte en tableau bi-dimensionnel et vice versa
-- le formatage d'un texte et sa representation interne matricielle

Les entrée/sortie de textes et l'interaction avec l'utilisateur
-- accès aux objets texte, fichier et fenêtre, affichage et style
-- trace d'exécution conservée dans un fichier séquentiel
-- utilitaires d'interaction: dialogues, buttonSelection, menus déroulants

L'arithmétique
-- l'analyse et l'évaluation d'expressions selon la syntaxe arithmétique usuelle
-- la représentation textuelle de données numériques, listes et matrices
-- l'utilisation d'une table de symboles comme mémoire

Chacune de ces sections aurait pu faire l'objet d'une *lib.scpt* à elle seule. Leur usage fréquent dans une même instruction (et la granularité grossière de la qualification par le tell d'Applescript) nous ont motivé à les réunir en un seul script.

Voici tout d'abord quelques précisions sur les types de paramètres et résultats des routines définies dans textLib.

1.3 Les types de données

Nous distinguons quatre[1] types de données: texte, liste, nombre et reference. Explicitons l'usage que l'on peut en faire.

Texte

Applescript a trois classes, *character, text* et *string*, pour les objets textuels, mais ne fait pas de distinction entre elles. *text* et *string* sont interchangeables et *character* est un *text* de longueur unité[2].
Il nous appartient de limiter volontairement la présence des caractères de contrôle line-feed, return et tab dans les chaînes, alors qu'ils sont admis dans un texte en général. En effet, nous emploierons systématiquement la fonte à chasse fixe (Monaco or Menlo) lors de l'affichage via *textEdit.app*, afin de contrôler la mise en page par des espaces (caractère space ou " ") en excluant délibérément l'usage de tabulateurs à cause de leur sémantique variable. Les seuls caractères de contrôle nécessaires sont dès lors linefeed et space, standards en Applescript, le return étant obsolète.

Les chaînes sans espace sont souvent utilisées comme identificateurs (noms) ou comme chemins d'accès (filepaths) identifiant un objet dans le système de fichiers.
D'autre part une utilisation particulière des chaînes de caractères sera proposée pour représenter des ensembles, sous-ensembles d'un univers de petite taille (une centaine d'éléments) dont on veut pouvoir faire l'union, l'intersection, etc.

Listes

Applescript offre des primitives de traitement de listes pratiques et faciles d'emploi à l'instar par exemple de LISP ou Scheme. Les listes simples de chaînes ou nombres (vecteurs) sont fréquentes dans les applications; quelques primitives pour traverser, combiner ou extraire des sous-listes font partie de la panoplie.

Par contre, à la différence de LISP, Applescript ne fournit pas de primitive comme "map" permettant d'appliquer une même opération à chaque élément d'une liste.

---

[1] Nous ferons aussi usage du type Boolean pour des instructions conditionnelles ou itératives.

[2] Applescript fournit l'opérateur "&" pour la concaténation de chaînes de caractères et les notations de chaîne "" pour la chaîne nulle, les "citations" entre guillemets pour les constantes de type text (avec la convention habituelle "\"" resp. "\\" pour les guillemets (quote) resp. la barre oblique arrière.

Ceci a conduit à compléter la plupart des routines de traitement de chaînes par un test sur le type de leur argument, assorti de l'énoncé itératif de parcours de liste si nécessaire.
On a ainsi fait usage du test dynamique du type de paramètre, une facilité qui remplace la surcharge p.ex. dans Ada et compense l'absence de typage statique des paramètres.

Une classe particulière de listes est nécessaire pour les matrices, les jeux sur damier et autres casse-têtes, les tables ou les données de tableur. En particulier le formatage de texte en colonnes utilise une structure de données donnant access aux éléments individuels du texte de la page. Nous appellerons ce type de données une "**mat**" (**m**atrix of **a**lphanumeric **t**okens) et l'implémentons comme une liste de listes.
Une mat est un tableau bi-dimensionnel de chaînes ou nombres.
L'ensemble de routines destinées à opérer sur des mats portent des noms avec suffixe "Mat". Certaines servent à la manipulation des lignes et colonnes, d'autres assurent la conversion ou la représentation textuelle d'une telle mat à l'aide d'une tabulation i.e. par une justification des éléments de textuels en colonnes dans la page (Formatage).

Arithmetique

Abordons enfin une autre classe de listes, celle des arbres syntaxiques, rarement mentionnée car cachée dans les routines d'analyse et d'évaluation d'expressions arithmétiques. Voici la motivation.
L'utilisateur d'une application doit pouvoir (dynamiquement, e.g. dans un dialogue) fournir un paramètre numérique, disons de 0.5, dans sa forme équivalente ½. De plus, lorsque deux constantes telles que width et height sont contraintes par un ratio donné de 16/9 , l'utilisateur doit pouvoir exprimer ce fait par exemple et donnant:

width = 500, height = width * 9 / 16.

Nous sommes donc confrontés à la nécessité d'interpréter des expressions arithmétiques voire des affectations telles que "height = width*9/16" dans le texte.
En général la représentation d'une expression arithmétique, outre sa notation textuelle parenthésée, est un celle d'un *arbre syntaxique*. Cet arbre est une liste, résultat de l'analyse syntaxique d'une expression algébrique, et donne ensuite lieu à l'évaluation par une routine récursive semblable à l'*eval* de LISP.
Inversément la présentation littérale de données numériques (nombres, matrices ou listes) dans un texte, en notation décimale, sera facilitée par les routines de conversion ad hoc. Leurs noms se terminent en général par "asText".

References

Les objets tels que les fichiers ou les fenêtres sont visibles par l'utilisateur d'un script. En même temps ils doivent pouvoir être repérés par une référence à l'intérieur du script pour un accès répété. Il y a deux sortes de référence pour un objet textuel: *document* respectivement *alias,* selon l'application (textEdit resp. Finder) qui gère l'objet. Notons que, dans le cas des fichiers, Finder accepte également des chemins d'accès (c.-à-d. des chaînes) pour désigner les fichiers.
Note: les références internes à des listes, sous-listes ou leurs éléments entrent en jeu dans un script sous diverses formes, par exemple d'une *variable* lors de **set** *variable* **to** *...* ou **repeat with** *variable* **in** *...* ou d'un *sélecteur* tel que **item** *ind* **of** *... .*
Applescript tente de les rendre complètement implicites et automatiquement "déréférencés".La confusion entre référence et valeur (ou contenu) qui en résulte peut conduire à des résultats inattendus dans les tests d'égalité (a=b), d'appartenance (**is in**) ou autres localisations (**offset).** textLib fournit des primitives eq(a,b), is_in(a,b) et offsets(a,b) plus sûres (au prix de la performance) qui conviennent aux tests entre objets quelconques représentés par des structures de liste. De même pick(aList, indices) fournit une image (une copie) des éléments de aList indicés, évitant les alias des sélections **item** indice **of** aList.

2. Texte, chaînes de caractères

Ce module definit les routines de manipulation de texte, en particulier deux variantes de substitution ainsi qu'une généralisation de la concaténation.

2.1 Opérations sur les chaînes

substring

substring(atext, a, b) extrait une sous-chaîne de atext commençant à la position a et terminant à b. substring gère les dépassements, en corrigeant les paramèters a ou b si nécessaire, afin d'éviter les messages d'erreur.

Généralisation

a ou b peuvent être négatifs, ils comptent alors depuis la fin de atext.

P.ex. substring ("abracadabra", 3, -2) = "racadabr"

repetition

repetition(littlestring, ntimes, separator) insère la chaîne separator, (ntimes -1) fois, entre les copies concaténées de littlestring. Ex. repetition("--", 5, "+") = "--+--+--+--+--"

occurrences

occurrences(atext, c) est le nombre d'occurrences du caractère c dans atext.

Ex. occurrences("abracadabra", "a") = 5 mais occurrences("Abracadabra", "a") = 4

Généralisations

c peut être un ensemble (une liste) de plusieurs caractères qui doivent être pris en compte.

Si atext est une liste, alors occurrences fait le total en parcourant ses éléments.

strip

strip(atext, Char) élimine toute occurrence du caractère Char en fin ou en début de atext (comme le fait p.ex. REXX d'IBM).

Souvent Char est l'espace (" " ou *space*). Par exemple strip(" 3.14 "," ") vaut "3.14"

Cependant, à la différence de REXX, notre strip traverse les listes, "strippant" chaque élément de la liste tour à tour. Par ex. strip({" do", "re ", " mi "}," ") = {"do", "re", "mi"}

Cas particulier

Si atext n'est composé que de Chars alors strip ne les élimine pas tous: strip("----------", "-") = "-"

pad

pad(atext, Char, total, Mode) ajoute des copies de Char aux deux bouts de atext, jusqu'à concurrence d'une longueur total imposée.

La répartition des Chars ajoutés aux extrémités se fait selon la valeur du paramètre Mode.

- Mode = "C" texte centré, padding bilatéral avec Char (~équilibré)
- Mode = "R" or "D" justification du texte à droite, padding à gauche
- Mode = "L" or "G" justification du texte à gauche, padding à droite
- Mode = "A" ou "" justification automatique, à gauche si atext commence par une lettre, à droite si atext finit par des chiffres, centré si aucun des deux.

Par exemple pad("Alpha",".",11,"D")="......Alpha", pad("Alpha",".",11,"C")="...Alpha..."

Generalisation

pad traverse les listes, chaque élément du paramètre atext est alors "paddé"; on peut ainsi justifier une colonne de texte (comparer avec Formattext ci-dessous).

replace

replace(atext, substring, substitut) effectue séquentiellement le remplacement des occurrences de la sous-chaîne substring par autant de substitut dans atext.

Par exemple replace("Today, John is tired", "John", "Fred") = "Today, Fred is tired"

Generalisations
Si atext est une *liste* d'éléments textuels, chaque élément est affecté (map implicite).
D'autre part substring peut être une liste de chaînes, toutes remplacées par la même substitut. Enfin, si cette liste commence par "a=A" ou "ignoring_case", alors replace ne fera pas de distinction entre minuscules et majuscules.

substitution

substitution(atext, candidates, substituts)
L'appel à substitution effectue une substitution de caractères en parallèle. Chaque caractère de atext figurant parmi les candidates est remplacé par le caractère homologue (de même offset) parmi les substituts. Par exemple on passe "abCDef12" en majuscules:
substitution("abCDef12", "abcdefghijklmnopqrstuvwxyz",
"ABCDEFGHIJKLMNOPQRSTUVWXYZ") = " ABCEF12"

Cas particuliers
S'il y a plus de candidates que de substituts, les caracères excédentaires parmi les candidates sont supprimés (en particulier lorsque substituts = "")
S'il y a plus de substituts que de candidates, alors le dernier charactère des candidates est remplacé par le solde de substituts
Si candidates="" alors la chaîne substituts est insérée entre deux caractères consécutifs de atext.
Si un caractère apparaît deux fois dans candidates la première occurrence seule fait foi.

Generalisation
Grâce aux règles de coercition d'Applescript substituts peut être une liste de chaînes; chaque caractère des candidates sera remplacé par l'élément correspondant de la liste de substituts. Sous cette forme substitution est un homomorphisme de chaînes conservant la concaténation.
Note:
Lorsque candidates est formé d'un seul caractère, disons "c" alors:
substitution(atext, "c", substitut) = replace(atext,"c", substitut)
replace est toutefois beaucoup plus rapide!

glue

glue(alist, delimiter) concatène tous les éléments de alist en insérant une copie de delimiter entre deux éléments consécutifs. En particulier, lorsque delimiter est la chaîne nulle "", glue est la concaténation.
Exemples:
glue ({"Alpha", "Beta", "Gamma"}, space) = "Alpha Beta Gamma",
glue ({"Alpha", "Beta", "Gamma"}, ", ") = "Alpha, Beta, Gamma"
glue ({"Alpha", "Beta", "Gamma"}, linefeed) =
"Alpha
Beta
Gamma"

Généralisation
Le paramètre effectif alist peut contenir des nombres, qui seront automatiquement convertis en texte (notation décimale scientifique) p.ex. glue({1,2,3},"+") = "1+2+3"

split

split(atext, delimiter_s) à l'inverse de glue, split sépare atext en une liste de sous-chaînes, les césures ayant lieu à chaque occurrence de delimiter_s. Par ex.
Si csv = "Value,1.5,100,-10" , alors split(csv, ",") = {"Value", "1.5", "100", "-10"}

Cas particulier
Pour delimiter_s = "", la chaîne nulle, on a split(atext, "") = *characters of* atext, autrement dit la liste des caractères individuels de atext.

Généralisation
le paramètre delimiter_s peut être une liste de chaînes, chacune agissant comme délimiteur possible. P.ex. split(aPage, {linefeed,return}) = *paragraphs of* aPage.[3]

2.2 Opérations sur les ensembles

Un usage particulier des chaînes de caractères consiste à représenter des ensembles, sous-ensembles d'un univers de petite taille. Lorsque les éléments de tels ensembles sont des caractères, chaque ensemble peut être représenté par une chaîne et les primitives ci-dessous d'intersection, union, etc. s'appliquent[4].

intersection

intersection(A, B) est une chaîne formée de tous les caractères apparaissant dans A **et** dans B (présentés dans le même ordre que dans A).
P.ex. intersection("AaBc", "abcd") = "ac"

difference

difference(A, B) est formée de tous les caractères apparaîssant dans A mais **pas** dans B (présentés dans le même ordre que dans A). C'est le complément relatif de B dans A.
P.ex. difference ("AaBc", "abcd") = "AB"

union

union(sets) est une chaîne formée de tous les caractères apparaîssant dans sets, où sets est une famille d'ensembles donnée par une liste ou une concaténation.
Contrairememt à intersection, union ne prend qu'un paramètre; pour deux ensembles A, B, on écrira union({A,B}) ou union(A&B) au lieu de union(A,B).
Le résultat est une chaîne sans répétitions, triée dans l'ordre alphabetique (Unicode).
P.ex. union("AçaB" & "abc1") = "1ABabcç"

Note: union est une opération associative, commutative et idempotente.
P. ex. union({A,B,C}) = union({A ,union({B,C})) = union({C,B,A}) et union({A,A})=union(A)

includes

includes(big, small) teste l'inclusion d'ensembles représentés par des chaînes. Le résultat, *true* ou *false* selon que small est effectivement contenu dans big, peut être utilisé comme condition dans les instructions **if** ou **repeat**.
P. ex. includes("A1B2C3", "ABC") est vrai mais includes("a1b2c3", "BC") est faux.

Généralisation
big et small peuvent être donnés comme liste de caractères au lieu de chaînes. P. ex.
includes({"A","B","C"},{"C","B"}) = *true*.
Toutefois includes ne concerne que les ensembles de caractères et ne s'applique pas aux listes généralisées du prochain chapitre.

---

[3] Si *delimiter_s* commence par "a=A" ou "ignoring_case" *split* ne différenciera pas minuscules et majuscules

[4] Si une application a besoin d'ensembles d'objets, autres que des caractères, on pourra tout de même utiliser des caractères pour les indexer et ainsi bénéficier des opérations ensemblistes fournies.

3 Listes

3.1 Listes de chaînes ou nombres

discard

discard(littleList, bigList) est une liste contenant tous les éléments de bigList n'appartenant **pas** à littleList. Les éléments communs sont éliminés. Analogue du "complement relatif" de littleList dans big List pour les ensembles. Les éléments peuvent aussi être des listes.

intersect

intersect(alist, blist) est la liste des éléments communs à alist **et** blist, dans le même ordre que dans alist. Pour un équivalent de l'union voir concat et noduplicates.

concat

concat(lol) concatène les éléments de lol, une liste de listes (lol=list of lists :-).
Le résultat est une liste où chaque élément de lol apparaît comme sous-liste[5].
P.ex. concat({{1,2},{3,4},{5}}) = {1,2,3,4,5}, en particulier concat({L1,L2,L3}) = L1 & L2 & L3.

noduplicates

noduplicates(alist) élimine les éléments répétitifs de alist, tout à l'inverse de replicate

replicate

replicate(littleList, ncopies) est la concatenation de ncopies identiques de littleList.
P.ex. replicate({0,1}, 3) = {0,1, 0,1, 0,1}, replicate({0}, 4) = {0,0,0,0}

interval

interval(i, k) est la liste de tous les entiers compris entre i et k. Si k<i l'ordre est décroissant.
Note: i ou k peuvent être négatifs. P.ex. interval(-1,+2) = {-1, 0, 1, 2}
P. ex. pick(aList, interval(a,b)) est la sous-liste de aList d'indices allant de a à b.

shuffle, sort

shuffle(alist) est une permutation (pseudo-) aléatoire des éléments de alist.
P.ex. shuffle({A,B,C}) revient au même que pick({A,B,C}, shuffle({1,2,3}))
sort(aList) trie les nombres ou les chaînes de aList dans l'ordre standard. Cf. aussi sortMat.

pick

pick(alist, indices) est la liste tous les éléments de alist dont l'indice (la position ou l'offset) figure parmi les indices. La permutation, les répétitions et les sous-listes sont permises.
Les indices négatifs comptent depuis la fin. Ainsi: pick({A,B,C,D}, {2,1,-2}) = {B,A,C}[6]
Si ind est un entier on a: pick(liste, ind) = *contents of* **item** ind **of** liste

offsets, firstOffset

firstOffset(littleList, bigList) est la première position où littleList apparaît comme sous-liste[7] dans bigList. Zéro si absente. P.ex. firstOffset({2,3},{1,2,3,2,3}) = 2.
offsets(littleList, bigList) est la liste des positions où littleList est une sous-liste de bigList.
P.ex. offsets({2,3},{1,2,3,2,3}) ={2,4}
offsets(A,B)={} ssi firstOffset(A,B)=0 c.à-d. ssi A n'apparaît pas comme sous-liste dans B.

---

[5] Une *sous-liste* de L est une liste formée *d'éléments* ***consécutifs*** pris dans L.

[6] Notons le cas particulier prévu dans Applescript: pick({A,B,C,D},{2,3,4}) = **rest of** {A,B,C,D}={B,C,D}

[7] La distinction entre minuscule/majuscule peut être spécifée au préalable par *considering case*

3.2 Manipulation de tableaux

Nous appellerons "**mat**" (**m**atrix of **a**lphanumeric **t**okens) un tableau bi-dimensionnel de chaînes ou nombres. En Applescript une mat est une liste de listes simples.
Si les lignes d'une mat peuvent ainsi être manipulées à l'aide des opérations sur les listes ci-dessus, il n'en va pas de même pour les colonnes. Par analogie, pickMat, concatenationMat et concatMat font apparaître une mat comme une liste de vecteurs-colonne.

rectangularMat

rectangularMat(aMat) force une liste de listes aMat en une mat aux lignes de même longueur. Les éléments manquants de aMat sont comblés par des "_" (ou des 0 si les autres éléments de aMat sont des nombres)

transpositionMat

transpositionMat(aMat) est la transposée de aMat si aMat est une *mat* rectangulaire

concatenationMat, concatMat

concatenationMat(aMat, bMat) est une mat dont chaque ligne est la concaténaion par "&" des lignes de même indice dans aMat resp. bMat.
Note: On suppose (*count* aMat) = (*count* bMat), sinon aMat a la priorité.

Généralisaion
concatMat(listOfMats) concatène les mats, éléments de la listOfMats, par plusieurs concatenationMat. Analogue au concat pour les listes.

pickMat, restOfMat

pickMat(aMat, column_indices) sélectionne les *colonnes* de aMat dont l'indice est dans column_indices. Les permutations et répétitions sont permises. Comparer à pick.

Cas particulier
restOfMat(aMat) = toutes les colonnes de aMat sauf la première. Comparer à **rest of.**
On a par ex. concatenationMat(pickMat(M, {1}), restOfMat(M)) = M

Note:
Si k est un entier pickMat(aMat, k) est la liste des éléments dans la colonne k de aMat.
Par ex. pickMat(M, k) = pick*(*transpositionMat(M),k) est une liste simple
alors que pickMat(aMat, {k}) est une matrice à une seule colonne.

Divers (encore deux opérations concernant les lignes d'une mat)

sortMat

sortMat(aMat, columnIndex) trie les lignes de aMat selon la colonne columnIndex.
sortMat n'est pas rapide mais regroupe les lettres majuscules, minuscules et accentuées.

LookUpMat

LookUpMat(aMat, key, keyColumn, resultColumn) recherche la clef key dans la colonne keyColumn et renvoie le contenu de la colonne resultColumn, présent sur la même ligne.
Si key ne figure pas dans la colonne keyColumn le résultat est *missing value*.

Conversion texte-mat et formatage

textAsMat

textAsMat(atext, principal, secondary) est la conversion de atext en mat, séparant les lignes aux occurrences de principal et les éléments de chaque ligne aux secondary.
P.ex. textAsMat ("1 2 3//4 5 6", "//", space) = { {1, 2, 3}, {4, 5, 6}}

matAsText

matAsText(aMat, principal, secondary) inverse de textAsMat, insère secondary entre les éléments d'une même ligne de aMat, puis principal entre les lignes ainsi obtenues.
P.ex. matAsText({{1, 2, 3},{"a", "b", "c"}}, linefeed, space) = "1 2 3
a b c"

Généralisations

asMat

asmat(atext) convertit atext en une liste de listes en prenant soin d'éliminer toute répétition de space ou linefeed au préalable. Elle est ainsi plus souple que textAsMat.

matText

matText(any) est une généralisation de matAsText qui transforme une mat donnée en texte. Dans le cas où any est effectivement une mat, matText aligne ses éléments par colonnes comme Formattext(any, 1, "") ci-dessous. Dans ce cas elle est l'inverse de asmat. Dans tous les autres cas la liste any est représentée par ses éléments, séparés par des virgules et entourés de { }. Elle est alors identique à la conversion en texte par asText(any) valable pour n'importe quel type de données (section 5).

FormatText

FormatText(atext, minWidth, mode) ajuste atext en colonnes de largeur minimale minWidth selon le mode suivant.
Soit mode = adjust & pad & empty est formé de trois caractères, alors
adjust: "R" = rightadjust, "L" = leftadjust, "C" = centered, "A" = automatic.
pad = padding character. P. ex. un point de conduite "." , le plus souvent un space.
empty remplace un champ vide pour éviter l'ambiguité. Empty: "-" , "_" , 0 ou "ø" .
Soit mode = "" et il prend sa *valeur par défaut*: mode = "A _" = "A" *&* space &"_" .

Cas particulier
Dans FormatText le paramètre atext peut être une mat[8].
P. ex. la conversion d'une matrice numérique M en texte formaté peut s'énoncer:
Formattext(matAsText(M,linefeed,space), 1, "A _") = Formattext(M, 1, "") = matText(M)

3.3 Structures de listes

Chaque élément d'une liste peut être de n'importe quel type, notamment une liste.
Les opérations de la section 3.1 (à l'exception de interval et sort) ont été généralisées pour ces cas, propre aux objets structurés.
Deux nouvelles fonctions, eq et is_in ont été introduites pour pallier les limitations des opérations **=** et **is in** d'Applescript (elles utilisent systématiquement *contents* **of** et **copy**).

eq(A,B) est *l'égalité* des objets désignés par A resp. B (alors que A=B est *l'identité* des réf.)
is_in(littleList,biglist) est vrai ssi littleList est *égale* (selon eq) à une sous-liste de biglist.
pick(alist, indices) suit le même principe et se généralise à des indices dont la forme peut être une structure de liste construite sur des entiers (indices dans alist).

[8] Techniquement l'opération de formatage d'un texte requiert au préalable sa conversion en une mat.

4. Entrées/sorties et interactions

4.1 Entrée/sortie de texte sur fichier ou document

Un objet textuel peut être repéré a*)* par une référence au *document* créé par l'application *textEdit* ou *b)* par un *chemin d'accès* ou un *alias* créé par l'application *Finder.*
Les trois routines *gettext, settext* et *closetext* traitent les deux cas de manière transparente pour le programmeur.

setText

setText(adocument, atext) remplace le contenu de adocument par atext.
adocument doit repérer un objet (ouvert) de *textEdit* ou un fichier (ouvert, fermé ou pas encore créé) du *Finder*.
Pour un fichier .txt on peut fournir l'alias ou le chemin d'accès (filepath). Si le fichier n'existe pas mais le dossier est correct, le fichier sera créé dans ce dossier.
Si le dossier est absent le bureau sera pris par défaut.
P.ex. setText("test","mon petit texte") crée le fichier "test.txt" sur le bureau avec pour contenu "mon petit texte".

addText

addtext(adocument, atext) ajoute atext au contenu de adocument, séparé par un *linefeed*.
Equivalent à setText(adocument, getText(adocument) & *linefeed* & atext) mais 100x plus rapide pour de gros fichiers.
Généralisation
Le séparateur *linefeed* peut être changé en affectant textRecordDelimiter au préalable.

getText

getText(adocument) retourne le texte contenu dans adocument.
adocument doit repérer un objet de *textEdit* ou un fichier existant du *Finder*. Il peut être ouvert ou fermé.
P.ex. après setText("test", "mon petit texte") on retrouve getText("test") = "mon petit texte"

openText

openText(adocumentfile) ouvre une fenêtre pour le fichier adocumentfile et retourne une référence. Le paramètre adocumentfile doit être un *alias* du *Finder* ou un chemin d'accès valable, avec l'extension ".txt", ".rtf", ".docx" ou ".applescript". Si l'extension est absente ".txt" sera ajouté automatiquement.
Cas particuliers:
adocumentfile = "" signifie que l'utilisateur sera invité à choisir le fichier à ouvrir.
Si adocumentfile est la reference à une fenêtre elle sera réactivée comme *front window*

styleText

styleText (documentref, itsFont, itsSize) corrige le style (fonte et corps) du documentref ouvert; on l'emploie pour agrémenter la lecture d'un document qu'on vient d'ouvrir.
On prendra "Menlo" ou "Monaco" pour une fonte à chasse fixe.

closeText

closeText (adocument) referme la fenêtre ouverte par openText ou display. P.ex.

```
set ref to openText("")
set texteLu to getText(ref)
closeText(ref)
```

permettrait à l'utilisateur de choisir un fichier mais refermerait ce dernier aussitôt lu.

4.2 Trace d'éxécution dans un fichier séquenciel

trace, startTrace, stopTrace

startTrace(title) crée un fichier nommé "title-Trace.txt" sur le bureau, afin d'y recueillir une trace d'exécution. Il s'agit d'un fichier séquentiel auquel trace accède en mode écriture.

trace(myindex, object) ajoute une trace de l'object, convertie en texte. L'application peut fournir un indice ou une légende dans myindex. Cependant, avec myindex = "" cet élément de trace est numéroté automatiquement et estampillé d'une date.

stopTrace() inhibe la trace temporairement. Pour réactiver la trace refaire startTrace(title).

4.3 Utilitaires d'intéraction avec l'utilisateur

display

display(atext) affiche atext dans une fenêtre du bureau (en créant un document de l'application "textEdit"). display(atext) retourne une référence à cette fenêtre, ce qui permet de mettre à jour son contenu à l'aide de setText, resp. de le récupérer avec getText.
A l'utilisateur de faire une sauvegarde s'il le souhaite puis de fermer la fenêtre affichée.
Attention, plusieurs appels à display oblitèrent les fenêtres précédentes (elles réapparaîssent si on ferme la dernière, mais il vaut mieux utiliser setText dans ce cas).

Généralisation
display est un passe-partout: atext peut être n'importe quelle valeur, texte, nombre ou liste et sera présentée dans le même format que asText ci-dessous.

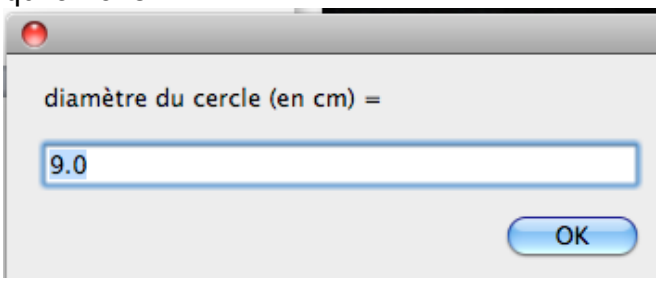
Voici encore un utilitaire passe-partout et très tolérant pour un dialogue simple.

dialog

dialog(prompt, defaultValue, buttonlist) ouvre une boîte de dialogue pour confirmer/modifier une donnée defaultValue telle qu'une liste, un nombre ou un texte.
defaultValue peut être de n'importe quel type; le résultat retourné sera du même type[9].
L'invite prompt incite l'utilisateur à confirmer la valeur affichée ou à la modifier. Le bouton choisi pour confirmer peut ensuite être consulté dans la variable globale dialogbutton.

Ex. 1 la valeur numérique diam du "diamètre du cercle" peut être définie par:
set diam **to** dialog("diamètre du cercle (en cm) =", 2*4.5, {"OK"})

qui affiche:



Avant de taper OK, l'utilisateur pourra écrire 10 à la place de 9.0 pour affecter diam.

---

[9] *Note*: Pour communiquer une valeur matricielle *aMat* on prendra soin de la convertir explicitement:
set *aMat* **to** asMat(dialog(matText(*aMat*), buttonlist)

Ex. 2 un texte "Convention d'échange ...", peut être montré à l'utilisateur pour accord:
dialog("vérifier s.v.p.", "Convention d'échange ...", {"d'accord", "demande révision" })
if dialogbutton = "d'accord" **then** return -- c'est OK, sinon procédons aux révisons ...

Ex. 3 les coordonnées d'un point, soit une liste de trois nombres, disons {0.5, 0, 1}, peuvent être initialisées (avec contrôle de l'utilisateur) par:
set {x, y, z} **to** dialog("{x, y, z}", {0.5, 0, 1}, "OK")

On notera que l'étiquette "OK" (et plus généralement une chaîne de noms séparés d'espaces) est tolérée à la place d'une liste de boutons.
L'invite prompt = "{x,y,z}" pourra également prendre plusieurs formes, y compris celle d'une liste telle que {"x","y","z"}.
En effet, {"x","y","z"} ou bien "x, y, z = ", voire "Coordonnées:" sont également lisibles et peuvent convenir, l'invite n'ayant de signification que pour l'utilisateur.

Cas particuliers de dialogues

OK - pour un dialogue concis

ok(yes_or_no_question, no_button, OK_button) est le booléen rapportant le choix de l'utilisateur. A noter que seule la position du bouton compte, pas son libellé, ce qui peut faciliter la programmation d'interactions multilingues.
P.ex. **if not** ok("Continuer ?", "bof", "oui, go") **then quit**

buttonSelection

buttonSelection(question, button_list) retourne le libellé du bouton sélectionné par l'utilisateur en réponse à la question affichée. Le nombre de boutons dans button_list est moins limité comparé à *display dialog* d'Applescript; voir cependant menuSelection ci-dessous.
P.ex. **if** buttonSelection("Continuer ?", {"non", "plus tard", "oui"}) ≠ "oui" **then quit**

menuSelection

menuSelection(prompt, choices) est l'élément choisi parmi une liste de choices.
La présentation sous la forme de menu déroulant diffère, mais le but est le même que pour buttonSelection[10].
P. ex. buttonSelection dans l'exemple ci-dessus peut être remplacé par l'énoncé:
if menuSelection("Continuer ?", {"non", "plus tard", "oui"}) ≠"oui" **then quit**

Généralisation (noter le pluriel!):
menuselections(prompt, choices) permet la sélection de **plusieurs** éléments parmi la liste des choices.
Le résultat est une liste (éventuellement vide si aucun élément n'est sélectionné).

[10] Deux détails toutefois à l'avantage de menuSelection: la liste choices peut être beaucoup plus longue et l'utilisateur a une possibilité d'*exit* **sans** faire de choix. Le résultat est alors *missing value*.

5. Arithmétique

Le but de cette section est la conversion des valeurs numériques en texte et réciproquement. Commençons par la conversion d'une valeur en texte

5.1 la représentation textuelle d'une valeur

asText

asText(anyvalue) est la représentation textuelle de anyvalue,
en notation décimale scientifique pour les nombres,
à l'aide d'accolades pour les listes,
avec des guillemets pour du texte.

Le paramètre anyvalue de cette routine passe-partout peut être de n'importe quel type[11].
P.ex. asText({{1, 2}, {3, 4, 5}}) = "{{1, 2}, {3, 4, 5}}"
Deux variantes concernent les *quote* = "\"" dans les chaînes et la notation décimale.

quoteText, unquoteText

quoteText(atext) ajoute des guillements à atext pour citer son contenu, en tenant compte des guillemets déjà présents dans atext, comme p. ex. dans "j'ai dit \"oui\"".
quoteText et asText sont donc identiques *pour les chaînes*.
Par contre, quoteText s'applique aux listes en transformant chaque chaîne qui y apparaît pour en faire une citation. Ce comportement est très différent de celui de asText, dont le résultat est toujours un texte. C'est la raison de cet aparté sur les "guillemets".
unquotetext fait l'inverse de quoteText en ôtant les guillemets des citations.

numberAsText

numberAsText(value, decimalDigits) est la notation décimale pour la valeur numérique de value, arrondie à un multiple de 1/10^decimalDigits. Elle a trois rôles:
- Eviter la notation scientifique (mantisse+exposant de 10) jusqu'à 20 décimales
- Aligner des valeurs numériques justifiées à droite dans un tableau (Cf. FormatText)
  numberAsText(pi/2, 4) = "1.5708", numberAsText(4*atan(1), 4) = "3.1416"
- Contrôler des arrondis à une puissance de 10 négative ou positive
  numberAsText(3.14159265358979, 11) *as real* = 3.14159265359
  numberAsText(141592653, -3) *as integer* = 141593000

Généralisation
Le paramètre value peut être une liste de valeurs numériques. Le résultat de numberAsText est alors la liste des notations décimales produites pour chaque élément.
En particulier une mat avec des éléments numériques peut être convertie avec numberAsText avant d'être formatée à l'aide de FormatText (avec un nombre de décimales uniforme).

5.2 l'évaluation d'une expression (numérique ou liste)

Passons maintenant à l'opération inverse, l'interprétation d'une donnée textuelle.

eval

eval(atext) analyse et évalue atext, qui peut être une expression arithmétique en syntaxe parenthésée, une notation de liste avec ses accolades (évent. emboîtées) ou une notation de chaîne (texte entre guillemets, concaténations).
P.ex. eval("{10/2, 18/3} & {-5, 0}") = {5.0, 6.0, -5, 0}, la valeur de {10/2, 18/3} & {-5, 0}.
Note: eval inverse l'effet de asText: eval(AsText(anyvalue) = anyvalue

[11] Le traitement des matrices diffère toutefois de celui des listes: privilégier matText.

Précisons que eval(expression) procède en deux phases, l'analyse et l'évaluation, qui font chacune l'objet d'une routine, syntaxTree resp. evalTree:

eval(expression) = evalTree(syntaxTree(expression))

syntaxTree

syntaxTree(expression) est l'arbre syntaxique correspondant à l'expression parenthésée; à la racine on trouve l'opérateur principal, aux feuilles les constantes et les variables[12].
La construction de l'arbre syntaxique utilise les routines suivantes (que nous ne détaillerons pas): expressionTree, termTree, factorTree et skipToken. Elle commence par présenter l'expression comme une liste de symboles appelés *tokens*.

tokens

tokens(atext) est la liste des symboles contenus dans atext. Un symbole ou token est soit un opérateur, un identificateur (alphanumérique), un nombre (en notation décimale), une notation de chaîne ou alors un délimiteur (parenthèses, accolades, virgule).
Les opérateurs sont "+", "-", "*", "/", "&" et "=" (voire ceux de C, Pascal ou Prolog)
P. ex. tokens ("(a+1)*sqrt(2)") = {"(", "a", "+", "1", ")", "*", "sqrt","(", "2", ")"}

treeAsText

treeAsText(expressionAsTree) est l'expression arithmétique parenthésée (en notation infixe) équivalente à l'abre expressionAsTree, mettant à profit la précédence d'opérateurs pour faire l'économie de parenthèses. C'est l'inverse de la routine syntaxTree définie ci-dessus.

Généralisation

Les routines syntaxTree et son inverse treeAsText acceptent également des *affectations* de la forme: *identificateur* = *expression.* L'opérateur principal est alors le "=".

evalTree

evalTree (expressionTree) parcourt l'arborescence expressionTree depuis les feuilles vers la racine et, à chaque noeud, synthétise sa valeur à partir de celle des fils. Le résultat est la valeur remontée à la racine.
Deux cas particuliers peuvent se présenter.
Si expressionTree a la forme d'une affectation : *identificateur* = *expression,* alors *l'expression* est évaluée par evalTree puis stockée dans une table sous la clé *identificateur.*
Inversément, si evalTree rencontre l'*identificateur* (dans une feuille de expressionTree) sa valeur est lue dans la table.

Cas d'erreur

Les erreurs de syntaxe étant fréquentes, syntaxTree effectue l'analyse le plus loin possible et affiche un message "symbole inattendu ...". Les sources d'erreur peuvent être:
Guillemet manquant parmi les tokens, parenthèses mal équilibrées dans l'expression, etc.

Une erreur peut être de nature sémantique, lorsque evalTree rencontre une valeur inconnue ou impossible. Par exemple, si un *identificateur* apparaît dans l'expression mais pas dans la table alors l'utilisateur est alerté.
De même, si l'*identificateur* (une variable) a pris la valeur factice *missing value* ou si une fonction n'est pas définie (p.ex. ln(x) ou 1/x avec x=0) alors il faut "déboguer".

---

[12] Rappel: une expression est une somme (ou différence) de termes, un terme est un produit (ou quotient) de facteurs. Un facteur est une constante, une variable, un appel de fonction ou une expression entre parenthèses. De plus, un facteur peut encore être élevé à une puissance (notation b^n).
Les fonctions reconnues sont celles exportées par textLib: cos, sin, atan, exp, ln, sign et abs.

5.3 La Table de symboles

On a vu que l'évaluation d'expressions arithmétiques peut passer par une table associative pour mémoriser les valeurs de certaines variables. L'accès direct à cette table est autorisé de manière plus générale.

lookUptable

lookUptable(symbol) trouve la valeur affectée précédemment à la clé symbol. Si symbol ne se trouve pas dans la table le résultat est *missing value*.

Généralisation

lookUptable permet de lire les valeurs de plusieurs symboles en une fois. Les clés peuvent être données sous la forme d'une liste ou dans une même chaîne, séparées par des espaces. Le résultat est alors une liste.

P. ex. lookUptable("A B") = lookUptable({"A", "B"}) = {lookUptable("A"), lookUptable("B")}

assignTable

assignTable(symbol, value) affecte la valeur donnée value à la clé symbol. Cette valeur peut être de n'importe quel type.

Généralisation

Ici aussi l'affectation peut porter sur plusieurs symboles à la fois. symbol et value sont alors deux listes appariées. P.ex. assignTable({"A", "B"}, {10,20}) ou assignTable("A B", {10,20})

initTable

initTable(assignmentsAsText) initialise la Table à l'aide d'affectations sous la forme textuelle *clé = valeur.* On peut avoir plusieurs affectations dans assignmentsAsText, une par ligne. C'est une convention pratique pour initialiser la table à partir d'un fichier.

P. ex. initTable(getText("SauvegardeTable.txt").

symbolsOfTable

symbolsOfTable() est l'ensemble des symboles servant de clé définis dans la table.

tableAsText

tableAsText(symbolSelection) restitue le contenu de la table sous la forme textuelle d'affectations *clé=valeur.* Lorsque symbolSelection = {} alors on convient que tableAsText({}) = tableAsText(symbolsOfTable()). Plus généralement symbolSelection est une liste de symboles, sélectionnant ceux que l'on souhaîte afficher comme résultat.

P. ex. setText("SauvegardeTable.txt", tableAsText({})) sauvegarde le contenu de la table.

5.4 Fonctions prédéfinies

Selon les versions d'Applescript les fonctions trigonométriques ne sont pas définies[13].

textLib fournit les classiques: cos(x), sin(x), atan(x), exp(x) et ln(x), complétées par

inv(x), abs(x) et sign(x) valent 0 pour x=0, pour éviter des messages d'erreur

HS(x) est la fonction de saut unitaire en x=0, alias *Heaviside*

HSG(x, listeDeSauts) est un *Heaviside g*énéralisé aux *HiStoGrammes* avec un saut pour chaque point de la listeDeSauts. En particulier, avec listeDeSauts ={{0,1}} on retrouve la fonction de saut unitaire identique à HS(x).

trapezes(x, listOfPoints) fonction linéaire par morceaux passant par chaque point.

sigmoid(x, p) fonction de "saut" continument dérivable, centrée horizontalement en x=0, y=0.5; p=pente à l'origine (de préférence une puissance entière de 2).
Utile pour les transitions et animations.

Note: L'unité pour l'argument x dans sin(x) et cos(x) est par défaut le *degré.* Mais on peut opter pour le *radian* grâce à un appel à cos_sin_units("rad").

---

[13] On les trouve parfois sous la forme de *Scripting Additions;* attention, avec une syntaxe différente!

6. Un exemple complet

Supposons qu'une liste d'articles est donnée sous forme de texte, un article par ligne, dans le fichier "Donnee.txt" qui se trouve sur le bureau. On veut la présenter par ordre alphabétique, dans le fichier "Resultat.txt" au même endroit. Voici un script simple pour ce faire:

Variante 1

```
-- accès aux routines getText, split etc. définies dans textLib.scpt :
set lib to load script file ("Macintosh HD:library:scripts:Libs:textLib.scpt")
tell lib
	set D to getText("Donnee.txt") -- lecture de la donnée sur fichier
	set L to split(D, linefeed)	-- conversion en liste
	set T to sortlist L		-- avec sortlist de Scripting Additions[14]
	set R to glue(T, linefeed)	-- conversion du résultat en texte
	setText("Resultat.txt", R)	-- écriture du résultat sur fichier
end tell
```

Si le contenu du fichier "Donnee.txt" est, par exemple:

```
Turing A.
Wirth N.
Knuth D
Jobs S.
Zuckerberg M.
von Neumann J.
Kay A.
and some others
```

Alors le fichier "Resultat.txt" créé par ce script contiendra probablement:

```
Jobs S.
Kay A.
Knuth D.
Turing A.
Wirth N.
Zuckerberg M.
and some others
von Neumann J.
```

Le défaut visible dans ce résultat est que toutes les majuscules précèdent les minuscules, ce qui peut arriver dans certaines versions de MacOS.
On peut y remedier en utilisant *sortMat*, une routine de *textLib* qui s'applique plus généralement aux matrices, pour trier les lignes selon leur premier symbole:

Variante 2

```
set lib to load script file "Macintosh HD:library:scripts:Libs:textLib.scpt"
tell lib
	set D to getText("Donnee.txt") -- lecture de la donnée sur fichier
	set M to asMat(D)		-- conversion en mat
	set T to sortMat(M,1)		-- tri de M selon la 1ère colonne
	set R to matText(T)		-- conversion du résultat en texte
	setText("Resultat.txt", R)	-- écriture du résultat sur fichier
end tell
```

---

[14] Pour utiliser **sortlist**, **abs, cos** etc. dans les versions récentes de MacOS (≥10.14) il faut déclarer:

```
use scripting additions		-- définit sortlist, abs, cos, sin, atan
use application "SatimageOSAX"	-- téléchargé de Satimage.FR/software
```

Le fichier "Resultat.txt" créé par ce script contiendra effectivement:

```
and some others
Jobs S.
Kay A.
Knuth D.
Turing A.
von Neumann J.
Wirth N.
Zuckerberg M.
```

Considérons enfin le cas où une routine ad hoc de *tri* de liste a été programmée

Variante 3

```
set lib to load script file "Macintosh HD:library:scripts:Libs:textLib.scpt"
global lib -- rend la lib visible dans les routines telle que tri
tell lib
	set D to getText("Donnee.txt") -- lecture de la donnée sur fichier
	set L to split(D, linefeed)	-- conversion en liste
	set T to my tri(L)		-- my signifie que tri vient de mon script
	set R to glue(T, linefeed)	-- conversion du résultat en texte
	setText("Resultat.txt", R)	-- écriture du résultat sur fichier
end tell

on tri(liste)
	tell lib -- en prévision d'appels de routines de textLib
	--- quickSort de N. Wirth ou heapSort d'après D. Knuth,
	--- comparer à sort de textLib qui, en l'occurrence, aurait convenu!
	end tell
end tri
```

Note: une écriture plus compacte est possible avec la variante de **tell** lib **to,** dont la portée est d'une ligne ou encore avec la notation "orientée objet" qui utilise la qualification de chaque routine par le génitif lib's de lib, ce qui donne pour la variante 3:

Variante 3'

```
set lib to load script file "Macintosh HD:library:scripts:Libs:textLib.scpt"
tell lib to set L to split( getText("Donnee.txt"), linefeed) -- lecture d'une liste
set T to tri(L)                                    -- tri comme ci-dessus
tell lib to setText("Resultat.txt", glue(T, linefeed))    -- écriture de la liste triée
```

Variante 3"

```
set lib to load script file "Macintosh HD:library:scripts:Libs:textLib.scpt"
lib's split( lib's getText("Donnee.txt"), linefeed)        -- lecture d'une liste
set T to tri(L)                                    -- tri comme ci-dessus
lib's setText("Resultat.txt", lib's glue(T, linefeed))     -- écriture de la liste triée
```

7. Table des matières / Index