awk, mawk, nawk

Interpréteur de langage AWK

awk est un interpréteur pour le langage de programmation awk. Ce langage est utile pour manipuler des fichiers de données, traitement et recherche de texte, et pour prototyper et expérimenter des algorithmes.

Un programme **awk** est une séquence de paires **pattern action** et de définitions de fonction. Les programmes cour sont entrés sur la ligne de commande entre **' '** pour éviter une interprétation par le shell. Les programmes plus longs peuvent être lus depuis un fichier. Les données sont lues depuis une liste de fichiers sur la ligne de commande ou depuis l'entrée standard quand la liste est vide. L'entrée est scindée en enregistrements comme déterminé par la variable de séparation d'enregistrement, **RS**. Initialement, **RS = "\n"**. Chaque enregistrement est comparé avec chaque motif et s'il y'a correspondance, l'**action** est exécutée.

OPTIONS

-F value Définis le séparateur de champ, **FS**

-f file Lit depuis le fichier au lieu de la ligne de commande. Peut-être spécifié plusieurs fois.

-v var=value Assigne une valeur à une variable

– Indique la fin des options de manière non-ambiguë

mawk fournis :

-W version Version de mawk puis quitte

-W dump Écrit en listing style assembleur la représentation interne du programme sur stdout puis quitte.

-W interactive Définis les écritures sans mise en tampon sur stdout et les lignes mise en tampon lues depuis stdin. Les enregistrements depuis stdin sont lus sans regarder **RS**.

-W exec file Le programme est lu depuis le fichier.

-W sprintf=num Ajuste la taille du tampon sprintf de mawk au nombre d'octets spécifiés.

-W posix_space force mawk à ne pas considérer **'\n'** comme étant un espace.

Le langage AWK

Un programme awk est une séquence de paires **motif action** et de définitions de fonctions.

Un motif peut être
BEGIN
END
expression
expression, expression

motif ou **action** peut être omis (mais pas les 2). Si **action** est omis, il est implicitement **print**. Si le **motif** est omis, il est simplement matché. **BEGIN** et **END** nécessitent une **action**.

Les déclarations sont terminées par une nouvelle ligne, **';'** ou les deux. Les groupes de déclarations tels que les actions ou les boucles sont délimité par **,** comme en C. La dernière déclaration dans un block ne nécessite pas de terminaison. Les lignes blanches ne signifient rien ; une déclaration vide est terminée par un **';'**. Les déclarations longues peuvent continuer sur une ou plusieurs lignes avec \. Une déclaration peut être scindée sans un \ après un **',', [, &&, ||, do, else**, la parenthèse droite d'un **if, while** ou **for**, ou d'une définition de fonction. Un commentaire commence avec **#** et s'étend jusqu'à la fin de la ligne.

Les déclarations suivantes contrôlent le flux d'un programme dans les blocks :

if (expr) statement

if (expr) statement else statement

while (expr) statement

do statement while (expr)

for (opt_expr ; opt_expr ; opt_expr) statement

for (var in array) statement

continue

break

Types de données, conversions et comparaisons

Il y'a 2 types de données de bases, les chaînes et les nombres. Les constantes numériques peuvent être entier, décimal, ou en notation scientifique. Tous les nombres sont représentés en interne et tous sont calculés en virgule flottante. Donc **0.2e2 == 20** est vrai, et **true** vaut **1.0**

Les chaînes sont placées entre guillemets. Elles peuvent être continuées sur une nouvelle ligne en échappant **newline**. Les caractères échappés suivant sont reconnus :

**** \

\" "

\a alert, ascii 7

\b backspace, ascii 8

\t tab, ascii 9

\n newline, ascii 10

\v vertical tab, ascii 11

\f formfeed, ascii 13

\r carriage return, ascii 13

\ddd 1, 2, ou 3 chiffre octal pour le code ascii

\xhh 1 ou 2 chiffres hexa pour le code ascii

Il y'a réellement 3 types de données ; le troisième est de type "nombre et chaîne" qui a des données numériques et des chaînes en même temps. Les variables utilisateur, lorsqu'elles sont créées, sont initialisées à null, un nombre ou une chaîne qui vaut **0** ou **""**.

Le type d'une expression est déterminé par son contexte et une conversion de type se produit. Par exemple, pour évaluer :

```
y = x + 2 ; z = x "hello"
```

y est de type numérique. Si **x** est une chaîne, elle est convertie en numérique avant le calcul. **z** sera de type chaîne, et la valeur de **x** sera convertie en chaîne si nécessaire et concaténé avec **"hello"**. Une chaîne est convertie en numérique en utilisant le plus long préfixe numérique comme avec **atof(3)**. Une expression numérique est convertie en chaîne en remplaçant **expr** avec **sprintf(CONVFMT, expr)**, sauf si **expr** peut être représentée sur une machine comme un entier exact, alors il est convertit en **sprintf("%d", expr)**. **sprintf** est embarqué dans awk et duplique la fonction **fprints(3)**, et **CONVFMT** est une variable intégrée utilisée pour les conversions interne des nombres en chaîne et initialisé à **"%.6g"**. Les conversions peuvent être forcées, **expr ""** est une chaîne et **expr+0** est numérique.

Pour évaluer, **expr1 rel-op expr2**, s'ils sont de type numérique ou un nombre et une chaine, la conversion est numérique ; s'ils sont de type chaîne la comparaison est de type chaine ; si un opérande est de type chaîne, l'opérande non-chaîne est converti et la comparaison est de type chaîne. Le résultat numérique est 1 ou 0.

En booléen, tel que **if (expr) statement**, une expression chaîne est vrai si et seulement si elle n'est pas vide ; une valeur numérique si et seulement si elle ne vaut pas 0.

Expressions régulières

Dans le langage awk, les enregistrements, champs et chaînes sont souvent testés avec des expressions régulières. Elles sont délimitées par des **"/"**

expr /r/

Cette expression évalue à 1 si **expr** matche **r**, qui signifie qu'une sous-chaîne de **expr** est dans le jeu de chaîne définie par **r**.

/r/ action et $0 /r/ action

Sont identiques, et pour chaque enregistrement en entrée qui match **r**, **action** est exécuté. en fait, **/r/** est une expression régulière awk qui est équivalent à **($0 /r/)**.

awk utilise des expressions régulières étendues comme avec **egrep**. Les méta caractères, par exemple, ceux qui ont une signification spéciale dans les expressions régulières sont : **^$. [ ] | ( ) * + ?**

Les expressions régulières sont construites comme suit :

c match le non méta caractères

\c match un caractère définis par la même séquence échappée utilisée dans les constantes chaînes ou le caractère littéral c if \c n'est pas une séquence échappée.

. Mach n'importe quel caractère (incluant newline)

^ Match le début d'une chaîne

$ Match la fin d'une chaîne

[c1c2c3...] Match un caractère dans la classe c1c2c3... un intervalle de caractères est noté c1-c2.

[^c1c2c3...] Match tout caractère non listé dans la classe

Les expressions régulières sont construites depuis d'autres expressions régulières comme suit :

r1r2 Match r1 suivi immédiatement par r2 (concaténation)

r1 | r2 Match r1 ou r2 (alternation)

regen Match r répété 0 ou plusieurs fois

r+ Match r répété une ou plusieurs fois

r ? Match r 0 ou une fois

(r) Match r, fournissant un groupage

La précédence des opérateurs est : ***, +, ?**

Exemple

/^[_a-zA-Z][_a-zA-Z0-9]*$/ et /^[-+] ?([0-9]+\. ?|\.[0-9])[0-9]*([eE][-+] ?[0-9]+) ?$/
Sont matchés par les identifiants awk et les constantes numériques awk, respectivement. Noter que "." doit être échappé pour être reconnu comme point décimal, et que les méta caractères ne sont pas spéciaux dans les classes de caractères
BEGIN identifier = "[_a-zA-Z][_a-zA-Z0-9]*"
$0 "^" identifier
Affiche toutes les lignes qui commencent avec un identifier awk.
mawk reconnait l'expression régulière vide //, qui match la chaîne vide et est matché par une chaîne avant, après et entre chaque caractère. Par exemple
echo abc | mawk ' gsub(//, "X") ; print '
XaXbXcX

Enregistrement et champs

Les enregistrements sont lus en une fois, et stockés dans la variable **$0**. L'enregistrement est splité en champs qui sont stockés dans **$1**, **$2**, **...**, **NF**. La variable **NF** contient le nombre de champs et **NR** et **FNR** sont incrémentés de 1. Les champs derrière **$NF** sont mis à "".

Assigner **$0** recalcule les champs et **NF**. Assigner **NF** ou un champ reconstruit **$0** en concaténant le **$i** séparé par **OFS**. Assigner un champ avec un index plus grand que **NF** augmente **NF** et reconstruit **$0**.

Les données en entrée stockés dans les champs est de type chaîne, à moins que tout le champ est sous la forme numérique, le champ est alors de type chaîne et nombre. Par exemple :

echo 24 24E | mawk ' print($1 > 100, $1>"100", $2 > 100, $2>"100") '
0 1 1 1

Expressions et opérateurs

La syntaxe est similaire au C. Les expressions primaires sont des constantes numériques, variables, champs, tableaux et des appels de fonction. L'identifiant pour une variable, tableau ou fonction peut être une séquence de lettres, chiffres et '_', qui ne commencent pas avec un chiffre. Les variables ne sont pas déclarées ; elles existent quand elles sont référencées la première fois et sont initialisées à NULL.

Les nouvelles expressions sont composées avec les opérateurs suivant par ordre de priorité croissant :

Assignement = += -= *= /= %= ^=
Conditionnel ? :
Ou logique ||
Et logique $$
Array membership in
Matching !
Relationnel < > <= >= == !=
Concaténation (pas d'opérateur explicite)
Ajout + -
Multiplication * / %
Unaire + -
Non logique !
Exponentiel ^
Incrément et décrément ++ —
Champ $

Tableaux

awk fournis des tableaux à une dimension. Les éléments de tableau sont exprimé comme **array[expr]**. **expr** est convertit en interne en type chaîne, donc **A[1]** et **A["1"]** sont identique et l'index est "1". Les tableaux indexés par chaîne sont appelés des tableaux associatifs. Un tableau initialisé est vide ; les éléments existent au premier accès. Une expression dans un tableau vaut 1 si **array[expr]** existe, sinon vaut 0.

Il y'a une forme de **for** qui boucle chaque index d'un tableau :

```
for ( var in array ) statement
```

Définis **var** à chaque **index** de **array** et exécute **statement**. L'ordre dans lequel **var** traverse les indices n'est pas définis.

La déclaration **delete array[expr]**, cause **array[expr]** de ne pas exister. **mawk** supporte une extension, **delete array**, qui supprime tous les éléments d'un tableau.

Les tableaux multi-dimensionnels sont synthétisés avec concaténation en utilisant la variable intégrée **SUBSEP**. **array[expr1,expr2]** est équivalent à **array[expr1 SUBSEP expr2]**.

Tester des éléments multi-dimensionnels utilise un index en parenthèse, tel que :

```
if ( (i, j) in A ) print A[i, j]
```

Variables intégrées

Les variables suivantes sont intégrées et initialisées avant l'exécution du programme.

ARGC Nombre d'arguments sur la ligne de commande

ARGV Tableau des arguments sur la ligne de commande, **0..ARGC-1**

CONVFMT Format pour la conversion interne des nombres, chaînes, initialisé à **"%.6g"**

ENVIRON Tableau indexé par variable d'environnement. Une chaine d'environnement, **var=value** est stockée comme **ENVIRON[var] = value**

FILENAME nom du fichier d'entrée

FNR Nombre d'enregistrements dans **FILENAME**

FS Split les enregistrements dans des champs en tant qu'expression régulière

NF Nombre de champs dans l'enregistrement courant

NR Nombre d'enregistrement courant total dans le flux d'entrée

OFMT Format pour afficher les nombres ; initialisé à **"%.6g"**

OFS Séparateur de champs en sortie, initialisé à **" "**

ORS Termine chaque enregistrement en sortie, initialisé à **"\n"**

RLENGTH Longueur définis par le dernier appel de la fonction intégrée match()

RS Séparateur d'enregistrement en entrée, initialisé à **"\n"**

RSTART Index définis par le dernier appel à match()

SUBSET Utilisé pour construire des tableaux multiples, initialisé à **"\034"**

Fonctions de chaînes intégrées

gsub(r,s,t) gsub(r,s) Substitutions globale, chaque match avec l'expression régulière r dans la variable t est remplacée par la chaîne s. Le nombre de remplacement est retourné. Si t est omis, $0 est utilisé. un & dans s est remplacé par la sous-chaîne matché de t.

index(s,t) Si t est une sous-chaine de s, la position où t comment est retournée, sinon 0.

length(s) Retourne la longueur de s

match(s,r) Retourne l'index du plus long match de r dans s. sans match, retourne 0. RSTART contient la valeur de retour, RLENGTH la longueur du match ou 61 si aucun match. si une chaîne vide match, RLENGTH vaut 0 et 1 est retourné sur le match est au début, et length(s)+1 si le match est à la fin.

split(s,A,r) split(s,A) s est splitté en champs par r et les champs sont chargés dans le tableau A. Le nombre de champs est retourné. si r est omis, FS est utilisé.

sprintf(format, expr-list) Retourne un chaîne construite depuis expr-list en accord avec format.

sub(r,s,t) sub(r,s) Simple substitution, identique à gsub except au moins une substitution.

substring(s,i) substr(s,i) Retourne le sous-chaîne de s, commençant à l'index i de longueur n. Si n est omis, le suffix de s, commençant à i est retourné.

tolower(s) retourne s en minuscule

toupper(s) retourne s en majuscule

Fonctions arithmétiques intégrées

atan2(y,x) arctan de y/y entre -pi et pi

cos(x) Cosinus, x en radian

exp(x) Fonction exponentielle

int(x) Retourne x tronqué à 0

log(x) Logarithme naturel

rand(x) Retourne un nombre aléatoire entre 0 et 1

sin(x) fonction sinus, x en radian

sqrt(x) Retourne la racine carré de x

srand(expr) srand() Générateur de nombre aléatoire, utilisant l'horloge si expr est omis, et retourne la valeur. mawk génère un nombre aléatoire depuis l'horloge au démarrage donc il n'y a pas de réel besoin de srand(). srand(expr) est utile pour répéter des séquences pseudo-aléatoires.

Entrée et Sortie

Il y'a 2 déclarations de sortie, **print** et **printf**.

print expr1, expr2, ..., exprn Écrit **$0 ORS** sur la sortie standard, les expressions numériques sont converties en chaîne avec **OFMT**.

printf format, expr-list Duplique la fonction C printf. Toutes les spécifications sont reconnues avec les conversions **%c, %d, %e, %E, %f, %g, %G, %i, %o, %s, %u, %x, %X** et **%%**, et les qualifieur de conversion **h** et **l**.

La liste des arguments de **print** et **printf** peuvent optionnellement être entre parenthèses. Les nombres sont affichés en utilisant **OFMT** ou **"%u"** pour les entiers exacts. **%c** avec un argument numérique affiche le caractère 8-bits correspondant. Avec un argument chaîne, affiche le premier caractère de la chaîne. La sortie de **print** et **printf** peuvent être redirigés avec **>** ou **»** vers un fichier ou **|** vers une commande à la fin de la déclaration **print**. Les redirections accolées sont toujours de type flux ouvert. Par convention **mawk** associe le nom du fichier **/dev/stderr** avec **stderr**. **mawk** associe également **"-"** et **/dev/sdtout** avec **stdin** et **stdout** qui permet à ces flux d'être passés en fonctions.

La fonction d'entrée **getline** a les variations suivantes :

getline Lit dans **$0**, met à jours les champs, **NF**, **NR** et **FNR**.

getline < file Lit dans **$0** depuis **file**, met à jours les champs et **NF**.

getline var Lit l'enregistrement suivant dans **var**, met à jours **NR** et **FNR**.

getline var < file Lit le prochain enregistrement de file dans **var**

command | getline Pipe un enregistrement depuis **command** dans **$0** et met à jours les champs et **NF**

command | getline var Pipe un enregistrement depuis **command** dans **var**

getline retourne 0 sur end-of-file, -1 sur erreur, sinon 1.

La fonction **close(expr)** ferme le fichier ou le pipe associé avec **expr**. close retourne **0** si **expr** est un fichier ouvert ou une commande en pipe, **-1** sinon. **close** est utilisé pour relire un fichier ou une commande.

La fonction **fflush(expr)** vide le fichier de sortie ou le pipe associé avec **expr**. **fflush** retourne **0** si **expr** est un flux de sortie ouvert, sinon **-1**. **fflush** sans argument vide **stdout**. **fflush** avec un argument vide ("") vide toutes les sorties ouvertes.

system(expr) utilise **/bin/sh** pour exécuter **expr** et retourner le code de sortie de la commande **expr**. Les changements sont fait dans le tableau **ENVIRON** ne sont pas passé à la commande exécutée avec system ou les pipes.

Fonctions utilisateurs

La syntaxe pour une fonction utilisateur est la suivante
function name(args) statements
Le corps de la fonction peut contenir une déclaration de retour
return opt_expr
Une déclaration de retour n'est pas requise. Les appels de fonction peuvent être imbriqués ou récursifs. Les fonctions sont passées en expressions par valeur et les tableaux par référence. Les arguments supplémentaires servent de variables locales et sont initialisés à null. Par exemple,
csplit(s,A)
place chaque caractère de s dans le tableau A et retourne la longueur de s

```
function csplit(s, A, n, i)

 n = length(s)
 for( i = 1 ; i <= n ; i++ ) A[i] = substr(s, i, 1)
 return n
```

Placer des espaces supplémentaires entre les arguments passés et les variables locales est conventionnel. Les fonctions peuvent être référencés avant qu'elles soient définies, mais le nom de la fonction et le '(' des arguments doivent se toucher pour éviter les confusions avec les concaténations.

Splittes les chaînes, enregistrements et fichiers

awk utilise le même algorithme pour splitter les chaînes dans les tableaux avec **split()**, et les enregistrements dans les champs dans **FS**. **mawk** utilise essentiellement le même algorithme pour splitter les fichiers en enregistrement dans **RS**. **split(expr,A,sep)** fonctionne comme suit :

(1) si **sep** est omis, il est remplacé par **FS**. **sep** peut être une expressions régulière ou une expressions. Si c'est une expression de type non-chaîne, il est convertit en chaîne.

(2) si **sep = " "** (un simple espace), alors il est recherché du début à la fin de **expr**, et **sep** devient "". **mawk** définie en tant qu'expression régulière **/[\t\n]+/**. Sinon **sep** est traité comme expression régulière, excepté que les méta-caractères sont ignorés pour les chaînes de longueur 1, par exemple, **split(x, A, "μ")** et **split(x, A, /*/)** sont équivalents.

(3) Si **expr** n'est pas une chaîne, il est convertit en chaîne. si **expr** vaut **""**, **split()** retourne 0 et 1 est vide. Sinon, les matches de **sep** dans **expr**, sépare **expr** dans des champs qui sont stockés dans **A** et **split()** retourne le nombre de champs dans **A**.

Splitter les enregistrements en champs fonctionne de la même manière excepté que les pièces sont chargées dans **$1, $2, ..., $NF**. Si **$0** est vide, **NF** vaut 0 et tous les **$i** sont à "".

mawk split les fichiers en enregistrements par le même algorithme, mais à la différence que **RS** est réellement un terminateur au lieu d'un séparateur.

exemple, si **FS = " :+"** et **$0 = "a : :b :"** , alors **NF = 3** et **$1 = "a", $2 = "b"** et **$3 = ""**, mis si **"a : :b :"** est le contenu d'un fichier d'entrée et **RS = " :+"**, alors il y'a 2 enregistrements "a" et "b".

RS = " " n'est pas spécial

si **FS = ""**, **mawk** casse l'enregistrement en caractères individuels, et, similairement, **split(s,A,"")** place les caractères individuels de s dans A.

Enregistrements multi-lignes

Vu que **mawk** interprète **RS** comme expression régulière, les enregistrements multi-ligne sont facile. Définir **RS = "\n\n+"**, créé une ou plusieurs lignes blanches pour séparer les enregistrements. Si **FS = " "** (le défaut), alors un simple newline, via les règles plus haut, devient un espace et les simples newline sont des séparateurs de champs.

Par exemple, si un fichier est **"a b\nc\n\n", RS = "\n\n+"** et **FS = " "**, alors il y'a un enregistrement **"a b\nc"** avec 3 champs "a", "b" et "c", changer **FS = "\n"**, donne 2 champs "a b" et "c". Changer **FS = ""**, donne un champ identique à l'enregistrement.

Si vous voulez que les lignes avec des espaces ou tabulation soient considérés blanc, définir **RS = "\n([\t]*\n)+"**. Pour la compatibilité avec d'autres awk, définir **RS = ""** a le même effet que si les lignes blanches sont enlevées du début à la fin des fichiers et les enregistrements sont déterminé comme si **RS = "\n\n+"**. Posix nécessite que **"\n"** sépare toujours les enregistrements quand **RS = ""** sans regarder la valeur de FS. **mawk** ne supporte pas cette convention, parce que définir **"\n"** n'est pas nécessaire.

La plupart du temps quand vous changez **RS** pour les enregistrements multi-ligne, vous voulez également changer **ORS** à **"\n\n"** pour que l'espacement des enregistrements soit préservé en sortie.

Exécution de programme

Cette section décrit l'ordre d'exécution des programmes. D'abord **ARGC** est définit au nombre total d'arguments passé sur la ligne de commande. **ARGV[0]** a le nom de l'interpréteur awk et **ARGV[1] ... ARGV[ARGC-1]** contiennent les arguments de la ligne de commande.

Par exemple avec
mawk -f prog v=1 A t=hello B
ARGC = 5
avec
ARGV[0] = "mawk", ARGV[1] = "v=1", ARGV[2] = "A", ARGV[3] = "t=hello" et ARGV[4] = "B".
Ensuite, chaque block **BEGIN** est exécuté dans l'ordre. Si le programme consiste entièrement de blocks **BEGINS**, alors l'exécution se termine, sinon un flux d'entrée est ouvert et l'exécution continue. Si **ARGC** vaut 1, le flux d'entrée et mis à stdin, sinon les arguments de la ligne de commande sont examinés à la recherche d'un fichier.

Les arguments se divisent en 3 ensembles : fichiers, arguments et chaînes vide "". Un assignement a le format **var=string**. Quand **ARGV[1]** est examiné comme un argument fichier possible, s'il est vide, il est ignoré ; si c'est un assignement, l'assignement à **var** est fait et **i** saute au suivant ; sinon **ARGV[i]** est ouvert en entrée. S'il échoue à l'ouverture, l'exécution se termine avec le code d'erreur 2. Sans argument fichier, l'entrée devient stdin. **getline** dans une action **BEGIN** ouvre l'entrée. "-" comme argument fichier dénote stdin.

Une fois un flux d'entrée ouvert, chaque enregistrement entrant est testé avec chaque **pattern**, et si çà matche, l'action associée est exécutée. Une expression mach si elle vaut true. Un **BEGIN** match avant qu'une entrée soit lue, et un **END** match après que toutes les entrées aient été lues. Une plage de motif, **expr1,expr2,** match chaque enregistrement entre le match de **expr1** et le match de **expr2**, inclus.

Quand la fin d'un fichier sur produit sur un flux d'entrée, les arguments restants sur la ligne de commande sont examinés à la recherche d'un fichier, et s'il y'en a un, il est ouvert, sinon le pattern END est considéré matché et toutes les actions END sont exécutées.

Dans l'exemple, l'assignement **v=1** prend place après que **BEGIN** ait été exécutée, et la donnée placée dans **v** est de type nombre et chaîne. L'entrée est ainsi lue depuis le fichier A. à la fin du fichier A, **t** est définis à **"hello"**, et B est ouvert en entrée. À la fin du fichier B, les actions **END** sont exécutées. Le flux de programme au niveau pattern **action** peut être changé avec :

next Le prochain enregistrement en entrée est lu et le motif est testé pour redémarrer avec le premier pattern action

exit opt_expr Force l'exécution immédiate des actions END ou la fin du programme s'il n'y en a pas ou si exit se produit à la fin d'une action END. opt_expr définie la valeur de sortie.

Exemples

Émuler cat

```
print
```

Émuler wc

```
chars += length($0) +1
words += NF
END print NR, words, chars
```

Compter le nombre de mots uniques

```
BEGIN FS = "[^A-Za-z]+"
 for(i = 1 ; i <= NF ; i++) word[$i] = ""
END delete word[""]
 for ( i in word ) cnt++
 print cnt
```

Ajoute le second champs de tous les enregistrements basés sur le premier champ

```
$1 /credit|gain/ sum += $2
$1 /debit|loss/ sum -= $2
```

Trie un fichier, comparer les chaînes :

```
line[NR] = $0 ""
END isort(line, NR)
 for(i = 1 ; i <= NR ; i++) print line[i]

function isort( A, n, i, j, hold)
 for( i = 2 ; i <= n ; i++)
 hold = A[j = i]
 while ( A[j-1] > hold )
  j-- ; A[j+1 = A[j]
 A[j] = hold
```

Problèmes de compatibilité

nawk apporte ces extensions :

nouvelles fonctions : toupper() et tolower()

nouvelles variables : ENVIRON[] et CONVFMT

Spécifications de conversion ANSI C pour printf() et sprintf()

Options de ligne de commande : -v var=value, plusieurs fois -f et l'implémentation des options en tant qu'argument de -W.

Posix AWK est orienté pour opérer sur les fichiers une ligne à la fois. **RS** peut être changé de **"\n"** à un autre caractère, mais c'est difficile de trouver une utilisation pour ça. Par convention, **RS = ""**, créé une ou plusieurs lignes blanches séparant les enregistrements, permettant les enregistrements multi-lignes. Quand **RS = ""**, **"\n"** est toujours un champ séparateur sans regarder **FS**.

mawk permet à RS d'être une expression régulière. Quand **"\n"** apparait dans l'enregistrement, il est traité comme espace, et FS détermine toujours les champs.

Supprimer la ligne à un certain moment peut simplifier les programmes et peut souvent améliorer les performances. Par exemple, en réécrivant les 3 exemples plus haut :

```
BEGIN RS = "[^A-Za-z]+"
word[ $0 ] = ""
```

```
END delete word[ "" ]
 for(i in word ) cnt++
 print cnt
```

Compte le nombre de mots uniques en faisant de chaque mot un enregistrement. Sur des fichiers de taille modérée, mawk s'exécute 2 fois plus vite.

Le programme suivant remplace chaque commentaire par un simple espace dans un fichier C

```
BEGIN
 RS = "/\*([^*]|\*+[^/*])*\*+/"
 ORS = " "
 getline hold

  print hold ; hold = $0
 END printf "%s" , hold
```

avec mawk, les déclarations suivantes sont équivalentes : x /a\+b/ x "a\+b" x "a\\+b"

Les chaînes sont scanées 2 fois, une fois en tant que chaîne, et une fois en tant qu'expression régulière. Au premier scan, **mawk** ignore les caractères d'échappement alors que **awk** reconnait **\c**.

Posix awk ne reconnait pas **"/dev/stdout,err"** ou la séquence **hexa \x** dans les chaînes. À la différence de l'ANSI C, **mawk** limite le nombre de chiffres qui suivent **\x** de 2 vu que l'implémentation actuelle supporte uniquement les caractères 8 bits.