Le manuel AWK

Édition 1.0
Décembre 1995

Diane Barlow Close
Arnold D. Robbins
Paul H. Rubin
Richard Stallman
Piet van Oostrum

Voici la première édition (1.0) du manuel AWK, pour
la nouvelle implémentation d'AWK (parfois appelée nawk).

Avertissement : Ce document est dérivé du manuel original de Gawk. Adaptations pour NAWK réalisées par Piet van Oostrum, décembre 1995, juillet 1998.

Préface

Comme beaucoup d'utilisateurs d'ordinateurs, vous souhaitez probablement modifier régulièrement des fichiers texte en fonction de certains motifs ou extraire des données de certaines lignes tout en ignorant le reste. Écrire un programme pour cela dans un langage comme le C ou le Pascal est fastidieux et peut nécessiter de nombreuses lignes de code. Avec awk, la tâche est simplifiée.

L'utilitaire awk interprète un langage de programmation spécialisé qui permet de
Gérez facilement les tâches simples de reformatage de données avec seulement quelques lignes de code.

Ce manuel vous explique le fonctionnement d'awk et comment l'utiliser efficacement. Vous devriez déjà Familiarisez-vous avec les commandes système de base telles que ls. Avec awk, vous pouvez :

- gérer de petites bases de données personnelles
- générer des rapports
- valider les données
- produire des index et effectuer d'autres tâches de préparation de documents • et même
  expérimenter des algorithmes qui pourront être adaptés ultérieurement à d'autres langages informatiques

Ce manuel a la lourde tâche de servir à la fois de tutoriel et de référence. Si vous êtes novice, n'hésitez pas à ignorer les détails qui vous semblent trop complexes. De même, vous pouvez ignorer les nombreuses références croisées ; elles sont destinées aux utilisateurs experts et à la version en ligne du manuel.

Histoire de l'embarras

Le nom awk provient des initiales de ses concepteurs : Alfred V. Aho, Peter J. Weinberger et Brian W. Kernighan. La version originale d'awk a été écrite en 1977. En 1985, une nouvelle version a enrichi le langage de programmation en introduisant les fonctions définies par l'utilisateur, la gestion de plusieurs flux d'entrée et le calcul d'expressions régulières. Cette nouvelle version a été généralisée avec System V Release 3.1. La version de System V Release 4 a ajouté de nouvelles fonctionnalités et a également amélioré le comportement de certains aspects complexes du langage. La spécification d'awk dans la norme POSIX (Command Language and Utilities) a permis de clarifier davantage le langage.

Nous tenons à remercier chaleureusement toutes les personnes qui ont contribué à l'élaboration de ce manuel. Jay Fenlason a apporté de nombreuses idées et des exemples de programmes. Richard Mlynarik et Robert J. Chassell ont formulé des commentaires précieux sur les premières versions de ce manuel. L'article « A Supplemental Document for awk » de John W. Pierce, du département de chimie de l'UC San Diego, a mis en lumière plusieurs points importants concernant la mise en œuvre d'awk et ce manuel, points qui nous auraient échappé autrement. David Trueman, Pat Rankin et Michal Jaegermann ont également contribué à la rédaction de certaines sections de ce manuel.

Les personnes suivantes ont formulé de nombreuses remarques utiles concernant cette édition du manuel : Rick Adams, Michael Brennan, Rich Burridge, Diane Close, Christopher (« Topher ») Eliot, Michael Lijewski, Pat Rankin, Miriam Robbins et Michal Jaegermann. Robert J. Chassell a fourni de précieux conseils sur l'utilisation de Texinfo.

LICENCE PUBLIQUE GÉNÉRALE GNU

Version 2, juin 1991

Préambule

La plupart des licences logicielles sont conçues pour restreindre votre liberté de partage et de modification. À l'inverse, la Licence Publique Générale GNU (GPL) vise à garantir votre liberté de partager et de modifier les logiciels libres, afin que ces logiciels restent gratuits pour tous leurs utilisateurs. Cette licence s'applique à la plupart des logiciels de la Free Software Foundation et à tout autre programme dont les auteurs s'engagent à l'utiliser. (Certains autres logiciels de la Free Software Foundation sont couverts par la Licence Publique Générale GNU pour les Bibliothèques (GPL).) Vous pouvez également l'appliquer à vos propres programmes.

Quand on parle de logiciel libre, on parle de liberté, pas de prix. Nos licences publiques générales sont conçues pour vous garantir la liberté de distribuer des copies de logiciels libres (et de facturer ce service si vous le souhaitez), de recevoir le code source ou de l'obtenir si vous le désirez, de modifier le logiciel ou d'en utiliser des parties dans de nouveaux programmes libres ; et pour vous assurer que vous êtes en droit de faire tout cela.

Afin de protéger vos droits, nous devons imposer des restrictions interdisant à quiconque de vous les refuser ou de vous demander d'y renoncer. Ces restrictions impliquent certaines responsabilités de votre part si vous distribuez des copies du logiciel ou si vous le modifiez.

Par exemple, si vous distribuez des copies d'un tel programme, gratuitement ou contre rémunération, vous devez accorder aux destinataires tous les droits dont vous disposez. Vous devez vous assurer qu'ils reçoivent également le code source ou puissent l'obtenir. Enfin, vous devez leur présenter ces conditions afin qu'ils connaissent leurs droits.

Nous protégeons vos droits en deux étapes : (1) le logiciel est protégé par le droit d'auteur, et (2) nous vous proposons cette licence. qui vous autorise légalement à copier, distribuer et/ou modifier le logiciel.

Par ailleurs, pour la protection de chaque auteur et la nôtre, nous tenons à préciser que ce logiciel libre n'est assorti d'aucune garantie. Si le logiciel est modifié et redistribué, nous souhaitons que ses destinataires sachent qu'il ne s'agit pas de la version originale, afin que les problèmes introduits par d'autres ne portent pas atteinte à la réputation des auteurs initiaux.

Enfin, tout logiciel libre est constamment menacé par les brevets logiciels. Nous souhaitons éviter que les redistributeurs d'un logiciel libre n'obtiennent individuellement des licences de brevet, rendant ainsi le logiciel propriétaire. Pour ce faire, nous avons clairement indiqué que tout brevet doit faire l'objet d'une licence permettant une utilisation libre par tous, ou ne pas être concédé du tout.

Les conditions précises de copie, de distribution et de modification sont détaillées ci-après.

CONDITIONS GÉNÉRALES DE COPIE ET DE DISTRIBUTION ET MODIFICATION

1. La présente licence s'applique à tout programme ou autre œuvre contenant une mention du titulaire des droits d'auteur indiquant qu'il peut être distribué selon les termes de la présente licence publique générale.
   Le terme « Programme », ci-dessous, désigne tout programme ou œuvre de ce type, et une « œuvre dérivée du Programme » désigne soit le Programme lui-même, soit toute œuvre dérivée protégée par le droit d'auteur : c'est-à-dire une œuvre contenant le Programme ou une partie de celui-ci, reproduite à l'identique ou modifiée et/ou traduite dans une autre langue. (La traduction est incluse sans limitation dans le terme « modification ».) Chaque titulaire de licence est désigné par le terme « vous ».

   Les activités autres que la copie, la distribution et la modification ne sont pas couvertes par la présente Licence ; elles sont hors de son champ d'application. L'exécution du Programme n'est pas soumise à restriction, et le résultat de son exécution n'est couvert que si son contenu constitue une œuvre dérivée du Programme (indépendamment du fait qu'elle ait été créée par son exécution). La nature de cette œuvre dépend de ce que fait le Programme.

2. Vous pouvez copier et distribuer des copies conformes du code source du Programme tel que vous le recevez, sur tout support, à condition de publier de manière visible et appropriée sur chaque copie un avis de droit d'auteur et une clause de non-responsabilité appropriés ; de conserver intactes toutes les mentions qui font référence à la présente Licence et à l'absence de toute garantie ; et de remettre à tout autre destinataire du Programme une copie de la présente Licence avec le Programme.

   Vous pouvez facturer des frais pour le transfert physique d'une copie, et vous pouvez, à votre discrétion, offrir une garantie en échange de frais.

3. Vous pouvez modifier votre ou vos copies du Programme ou toute partie de celui-ci, créant ainsi une œuvre dérivée du Programme, et copier et distribuer ces modifications ou cette œuvre conformément aux termes de l'article 1 ci-dessus, à condition de respecter également toutes les conditions suivantes : a. Vous devez

   faire en sorte que les fichiers modifiés comportent des mentions claires indiquant que vous avez modifié le Programme. fichiers et la date de toute modification.

   b. Vous devez faire en sorte que toute œuvre que vous distribuez ou publiez, qui contient en tout ou en partie le Programme ou une partie de celui-ci, ou qui en est dérivée, soit concédée sous licence dans son intégralité et sans frais à tous les tiers aux termes de la présente Licence.

   c. Si le programme modifié lit normalement les commandes de manière interactive lors de son exécution, vous devez faire en sorte que, lorsqu'il est lancé pour une utilisation interactive de la manière la plus courante, il affiche un message contenant une mention de droit d'auteur appropriée et une mention indiquant qu'aucune garantie n'est fournie (ou bien, indiquant que vous fournissez une garantie) et que les utilisateurs peuvent redistribuer le programme sous ces conditions, ainsi que la procédure pour consulter une copie de la présente licence. (Exception : si le programme lui-même est interactif mais n'affiche normalement pas un tel message, votre travail basé sur le programme n'est pas tenu d'afficher un message.)

   Ces exigences s'appliquent à l'œuvre modifiée dans son ensemble. Si certaines sections identifiables de cette œuvre ne sont pas dérivées du Programme et peuvent être raisonnablement considérées comme des œuvres indépendantes et distinctes, la présente Licence et ses conditions ne s'appliquent pas à ces sections lorsque vous les distribuez séparément. En revanche, lorsque vous distribuez ces mêmes sections au sein d'une œuvre basée sur le Programme, la distribution de l'ensemble doit être soumise aux conditions de la présente Licence, dont les autorisations accordées aux autres titulaires de licence s'étendent à l'intégralité de l'œuvre, et donc à chacune de ses parties, quel que soit son auteur.

   Ainsi, la présente section n'a pas pour but de revendiquer des droits ou de contester vos droits sur une œuvre entièrement écrite par vous ; elle vise plutôt à exercer le droit de contrôler la distribution des œuvres dérivées ou collectives basées sur le Programme.

   En outre, le simple regroupement d'une autre œuvre non basée sur le Programme avec le Programme (ou avec une œuvre basée sur le Programme) sur un volume d'un support de stockage ou de distribution ne place pas l'autre œuvre dans le champ d'application de la présente Licence.

4. Vous pouvez copier et distribuer le Programme (ou une œuvre dérivée de celui-ci, conformément à la Section 2) sous forme de code objet ou de fichier exécutable, conformément aux dispositions des Sections 1 et 2 ci-dessus, à condition de réaliser également l'une des actions suivantes :

   a. Joignez-y le code source complet correspondant, lisible par machine, qui doit être distribué conformément aux dispositions des articles 1 et 2 ci-dessus sur un support couramment utilisé pour l'échange de logiciels ; ou b. Joignez-y une offre écrite, valable pendant au moins trois ans,

   proposant à tout tiers, moyennant un coût n'excédant pas celui de la distribution physique du code source, une copie complète et lisible par machine du code source correspondant, à distribuer conformément aux dispositions des articles 1 et 2 ci-dessus sur un support couramment utilisé pour l'échange de logiciels ; ou c. Joignez-y les informations que vous avez reçues concernant l'offre de distribution du code source correspondant. (Cette option est autorisée uniquement pour

   une distribution non commerciale et seulement si vous avez reçu le programme sous forme de code objet ou de fichier exécutable avec une telle offre, conformément au paragraphe b ci-dessus.)

   Le code source d'une œuvre désigne sa forme privilégiée pour y apporter des modifications. Pour une œuvre exécutable, le code source complet comprend l'ensemble du code source de tous ses modules, ainsi que les fichiers de définition d'interface associés et les scripts permettant la compilation et l'installation de l'exécutable. Par exception, le code source distribué n'a pas à inclure les éléments normalement fournis (sous forme de code source ou binaire) avec les principaux composants (compilateur, noyau, etc.) du système d'exploitation sur lequel l'exécutable s'exécute, sauf si ce composant accompagne lui-même l'exécutable.

   Si la distribution de code exécutable ou de code objet se fait en offrant un accès à la copie à partir d'un emplacement désigné, alors l'offre d'un accès équivalent à la copie du code source à partir du même emplacement compte comme une distribution du code source, même si les tiers ne sont pas obligés de copier le code source avec le code objet.

5. Vous n'êtes pas autorisé à copier, modifier, concéder en sous-licence ni distribuer le Programme, sauf dans les conditions expressément prévues par la présente Licence. Toute tentative de copie, de modification, de concéder en sous-licence ou de distribution du Programme en dehors de ces conditions est nulle et entraînera la résiliation automatique de vos droits au titre de la présente Licence. Toutefois, les licences des tiers ayant reçu de votre part des copies ou des droits en vertu de la présente Licence resteront valides tant qu'ils continueront à respecter pleinement les termes de celle-ci.

6. Vous n'êtes pas tenu d'accepter cette Licence puisque vous ne l'avez pas signée. Toutefois, aucun autre élément ne vous autorise à modifier ou à distribuer le Programme ou ses œuvres dérivées. Ces actions sont interdites par la loi si vous n'acceptez pas cette Licence. Par conséquent, en modifiant ou en distribuant le Programme (ou toute œuvre dérivée du Programme), vous acceptez la présente Licence et l'intégralité de ses conditions relatives à la copie, à la distribution ou à la modification du Programme ou des œuvres dérivées.

7. Chaque fois que vous redistribuez le Programme (ou toute œuvre dérivée du Programme), le destinataire reçoit automatiquement du concédant de licence initial une licence l'autorisant à copier, distribuer ou modifier le Programme, sous réserve des présentes conditions générales. Vous ne pouvez imposer aucune restriction supplémentaire à l'exercice par les destinataires des droits qui leur sont accordés par les présentes. Il ne vous incombe pas de veiller au respect de la présente Licence par les tiers.

8. Si, suite à une décision de justice, une allégation de contrefaçon de brevet ou pour toute autre raison (y compris, mais sans s'y limiter, les questions de brevets), des conditions vous sont imposées (par décision de justice, accord ou autre) et contredisent les conditions de la présente Licence, vous n'êtes pas dispensé du respect de ces dernières. Si vous ne pouvez pas distribuer le Programme de manière à satisfaire simultanément à vos obligations au titre de la présente Licence et à toute autre obligation pertinente, vous ne pouvez en conséquence pas le distribuer du tout. Par exemple, si une licence de brevet n'autorise pas la redistribution gratuite du Programme par tous ceux qui en reçoivent des copies directement ou indirectement par votre intermédiaire, la seule façon de satisfaire à la fois à cette licence et à la présente Licence serait de vous abstenir totalement de distribuer le Programme.

Si une partie quelconque de cet article est jugée invalide ou inapplicable dans une circonstance particulière, le reste de l'article est destiné à s'appliquer et l'article dans son ensemble est destiné à s'appliquer dans d'autres circonstances.

Le but de cette section n'est pas de vous inciter à enfreindre des brevets ou autres droits de propriété intellectuelle, ni de contester la validité de tels droits ; elle vise uniquement à protéger l'intégrité du système de distribution de logiciels libres, mis en œuvre par les principes des licences publiques. De nombreuses personnes ont généreusement contribué à la vaste gamme de logiciels distribués par ce système, en se fiant à son application cohérente ; il appartient à l'auteur/donateur de décider s'il souhaite distribuer son logiciel par un autre moyen, et un titulaire de licence ne peut lui imposer ce choix.

Cette section a pour but de clarifier parfaitement ce qui est considéré comme une conséquence du reste de la présente licence.

9. Si la distribution et/ou l'utilisation du Programme sont restreintes dans certains pays par des brevets ou des interfaces protégées par le droit d'auteur, le titulaire initial des droits d'auteur qui soumet le Programme à la présente Licence peut ajouter une limitation de distribution géographique explicite excluant ces pays, de sorte que la distribution ne soit autorisée que dans ou entre les pays non exclus. Dans ce cas, la présente Licence intègre cette limitation comme si elle y figurait.

10. La Free Software Foundation peut publier périodiquement des versions révisées et/ou nouvelles de la Licence Publique Générale. Ces nouvelles versions seront similaires dans leur esprit à la version actuelle, mais pourront différer dans le détail afin de répondre à de nouveaux problèmes ou préoccupations.

    Chaque version se voit attribuer un numéro de version distinctif. Si le Programme spécifie un numéro de version de la présente Licence qui lui est applicable, ainsi que la mention « toute version ultérieure », vous pouvez choisir d'appliquer les termes et conditions de cette version ou de toute version ultérieure publiée par la Free Software Foundation. Si le Programme ne spécifie pas de numéro de version de la présente Licence, vous pouvez choisir n'importe quelle version publiée par la Free Software Foundation.

11. Si vous souhaitez intégrer des parties du Programme à d'autres logiciels libres dont les conditions de distribution diffèrent, veuillez contacter l'auteur pour obtenir son autorisation. Pour les logiciels protégés par le droit d'auteur de la Free Software Foundation, veuillez contacter la Free Software Foundation ; des exceptions peuvent être accordées. Notre décision sera guidée par deux objectifs : préserver le statut libre de tous les logiciels dérivés de nos logiciels libres et promouvoir le partage et la réutilisation des logiciels en général.

AUCUNE GARANTIE

12. Le programme étant concédé sous licence gratuitement, aucune garantie n'est fournie, dans la mesure permise par la loi applicable. Sauf mention contraire écrite, les titulaires des droits d'auteur et/ou autres parties fournissent le programme « en l'état », sans aucune garantie, expresse ou implicite, y compris, mais sans s'y limiter, les garanties implicites de qualité marchande et d'adéquation à un usage particulier. Vous assumez l'intégralité des risques liés à la qualité et aux performances du programme. Si le programme s'avère défectueux, vous prenez en charge tous les frais de maintenance, de réparation ou de correction nécessaires.

13. EN AUCUN CAS, SAUF SI LA LOI APPLICABLE L'EXIGE OU SI UN ACCORD ÉCRIT L'EST, LE TITULAIRE DU DROIT D'AUTEUR OU TOUTE AUTRE PARTIE AUTORISÉE À MODIFIER ET/OU À REDISTRIBUER LE PROGRAMME COMME AUTORISÉ CI-DESSUS, NE SERA RESPONSABLE ENVERS VOUS DES DOMMAGES, Y COMPRIS LES DOMMAGES GÉNÉRAUX, SPÉCIAUX, ACCESSOIRES OU INDIRECTS DÉCOULANT DE L'UTILISATION OU DE L'INCAPACITÉ D'UTILISER LE PROGRAMME (Y COMPRIS, MAIS SANS S'Y LIMITER, LA PERTE DE DONNÉES OU L'INCAPACITÉ DE DONNÉES, LES PERTES SUBIES PAR VOUS OU DES TIERS OU L'INCAPACITÉ DU PROGRAMME À FONCTIONNER AVEC D'AUTRES PROGRAMMES).

MÊME SI CE TITULAIRE OU CETTE AUTRE PARTIE A ÉTÉ AVERTIE DE LA POSSIBILITÉ DE TELS DOMMAGES.

FIN DES CONDITIONS GÉNÉRALES

Comment appliquer ces conditions à vos nouveaux programmes

Si vous développez un nouveau programme et que vous souhaitez qu'il soit le plus utile possible au public, la meilleure façon d'y parvenir est d'en faire un logiciel libre que chacun peut redistribuer et modifier selon ces conditions.

Pour ce faire, ajoutez les mentions suivantes au programme. Il est plus sûr de les ajouter au début de chaque fichier source afin de bien faire comprendre l'exclusion de garantie ; chaque fichier doit comporter au moins la mention « copyright » et un lien vers l'emplacement de la mention complète.

> Une seule ligne pour indiquer le nom du programme et une brève description de sa fonction.
> Copyright (C) 19yy nom de l'auteur
>
> Ce programme est un logiciel libre ; vous pouvez le redistribuer et/ou le modifier selon les termes de la licence publique générale GNU telle que publiée par la Free Software Foundation ; soit la version 2 de la licence, soit (à votre choix) toute version ultérieure.
>
> Ce programme est distribué dans l'espoir qu'il sera utile, mais SANS AUCUNE GARANTIE ; sans même la garantie implicite de QUALITÉ MARCHANDE ou d'ADÉQUATION À UN USAGE PARTICULIER. Voir le
>
> Licence publique générale GNU pour plus de détails.
>
> Vous auriez dû recevoir une copie de la licence publique générale GNU avec ce programme ; sinon, écrivez à Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, États-Unis.

Veuillez également ajouter vos coordonnées par courrier électronique et postal.

Si le programme est interactif, faites en sorte qu'il affiche un court message comme celui-ci lorsqu'il démarre en mode interactif :

> Gnomovision version 69, Copyright (C) 19yy nom de l'auteur Gnomovision est livré SANS AUCUNE GARANTIE ; pour plus de détails, tapez 'show w'.
>
> Il s'agit d'un logiciel libre, et vous êtes libre de le redistribuer sous certaines conditions ; tapez « show c » pour plus de détails.

Les commandes hypothétiques « show w » et « show c » devraient afficher les parties pertinentes de la licence publique générale. Bien entendu, les commandes que vous utilisez peuvent porter un autre nom ; il peut s'agir de clics de souris ou d'éléments de menu, selon les besoins de votre programme.

Vous devriez également obtenir la signature de votre employeur (si vous travaillez comme programmeur) ou de votre établissement scolaire, le cas échéant. Une mention légale relative aux droits d'auteur pour le programme, le cas échéant. Voici un exemple ; modifiez les noms :

> La société Yoyodyne, Inc., décline par la présente tout droit d'auteur sur le programme « Gnomovision » (qui effectue des passes sur les compilateurs) écrit par James Hacker.
>
> Signature de Ty Coon, 1er avril 1989
> Ty Coon, président de Vice

Cette licence publique générale n'autorise pas l'intégration de votre programme dans des programmes propriétaires. Si votre programme est une bibliothèque de sous-programmes, il peut être plus utile d'autoriser la liaison.
applications propriétaires utilisant la bibliothèque. Si c'est ce que vous souhaitez faire, utilisez la bibliothèque GNU.
Licence publique générale au lieu de la présente licence.

1 Utilisation de ce manuel

Le terme awk désigne un programme particulier, et le langage que vous utilisez pour indiquer à ce programme son fonctionnement.
Que faire ? Lorsqu'il faut être prudent, on appelle le programme « l'utilitaire awk » et le langage…
« Le langage awk. » Le terme gawk fait référence à une version d'awk développée dans le cadre du projet GNU.
Ce manuel a pour but d'expliquer à la fois le langage awk et comment exécuter l'utilitaire awk.

Le terme « programme awk » désigne un programme que vous avez écrit dans le langage de programmation awk.

Consultez le chapitre 2 [Premiers pas avec awk], page 13, pour connaître les notions essentielles à maîtriser.
Commencez à utiliser awk.

Quelques phrases courtes et utiles sont incluses pour vous familiariser avec le langage awk (voir chapitre 5) .
[Répliques utiles], page 45).

Un exemple de programme awk vous a été fourni (voir l'annexe B [Exemple de programme], page 119).

Si vous trouvez des termes qui vous sont inconnus, essayez de les rechercher dans le glossaire (voir l'annexe C [Glossaire], page 121).

L'intégralité du langage awk est résumée pour une consultation rapide dans l'annexe A [Résumé awk],
page 107. Consultez cette page si vous avez simplement besoin de vous rafraîchir la mémoire concernant une fonctionnalité particulière.

Le plus souvent, des programmes awk complets sont utilisés comme exemples, mais dans certains cas plus avancés…
Dans certaines sections, seule la partie du programme awk illustrant le concept décrit est affichée.

1.1 Fichiers de données pour les exemples

De nombreux exemples de ce manuel utilisent deux fichiers de données d'exemple. Le premier,
Appelée « liste BBS », elle représente une liste de systèmes de forums informatiques ainsi que des informations
concernant ces systèmes. Le deuxième fichier de données, appelé « inventory-shipped », contient des informations sur
Les expéditions sont enregistrées mensuellement. Chaque ligne de ces fichiers correspond à un enregistrement.

Dans le fichier « BBS-list », chaque enregistrement contient le nom d'un forum informatique, son numéro de téléphone
Le numéro, le débit en bauds de la carte et un code indiquant le nombre d'heures de fonctionnement. Un « A » dans le
La dernière colonne indique que le conseil d'administration fonctionne 24 heures sur 24. Un « B » dans la dernière colonne signifie que le conseil d'administration
Fonctionne uniquement en soirée et le week-end. Un « C » signifie que le conseil d'administration fonctionne uniquement le week-end.

piqûres de	555-5553	1200/300	B
barfly alpo-net	555-3412	2400/1200/300	UN
d'yctérope	555-7685	1200/300	UN
	555-1675	2400/1200/300	UN
Camelot	555-0542	300	C
cœur	555-2912	1200/300	C
fooey	555-1234	2400/1200/300	B
foot	555-6699	1200/300	B
macfoo	555-6480	1200/300	UN
se rendre	555-3430	2400/1200/300	UN

excusez-moi 555-2127 1200/300 C

Le deuxième fichier de données, intitulé « inventory-shipped », contient des informations sur les expéditions effectuées au cours de l'année. Chaque enregistrement comprend le mois de l'année, le nombre de caisses vertes expédiées, le nombre de boîtes rouges expédiées, le nombre de sacs orange expédiés et le nombre de colis bleus expédiés. Ce fichier comporte 16 entrées, couvrant les 12 mois d'une année et les 4 mois de l'année suivante.

13 janvier 25 15 115
15 févr. 32 24 226
15 mars 24 34 228
31 avril 52 63 420
16 mai 34 29 208
31 juin 42 75 492
24 juil. 34 67 436
15 août 34 47 316
13 sept. 55 37 277
29 oct. 54 68 525
20 novembre 87 82 577
17 déc. 35 61 401

21 janvier 36 64 620
26 février 58 80 652
24 mars 75 70 495
21 avril 70 74 514

2 Premiers pas avec awk

La fonction de base d'awk est de rechercher dans les fichiers des lignes (ou autres unités de texte) contenant certaines conditions. modèles. Lorsqu'une ligne correspond à l'un des modèles, awk effectue les actions spécifiées sur cette ligne. awk continue de traiter les lignes d'entrée de cette manière jusqu'à ce que la fin du fichier d'entrée soit atteinte.

Lorsque vous exécutez awk, vous spécifiez un programme awk qui indique à awk ce qu'il doit faire. Il se compose d'une série de règles. (Il peut également contenir des définitions de fonctions, mais c'est un niveau avancé.) Cette fonctionnalité n'est pas prise en compte, nous l'ignorerons donc pour le moment. (Voir le chapitre 12 [Fonctions définies par l'utilisateur], page 95.) Chaque règle spécifie un modèle à rechercher et une action à effectuer lorsque ce modèle est trouvé.

Du point de vue syntaxique, une règle consiste en un modèle suivi d'une action. L'action est encadrée par des accolades. des accolades pour le séparer du modèle. Les règles sont généralement séparées par des sauts de ligne. Par conséquent, un awk Le programme ressemble à ceci :

```
modèle { action }
modèle { action }
. . .
```

2.1 Un exemple très simple

La commande suivante exécute un programme awk simple qui recherche dans le fichier d'entrée 'BBS-list' les éléments suivants : la chaîne de caractères : « foo ». (Une chaîne de caractères est généralement appelée une chaîne de caractères. Le terme chaîne de caractères est utilisé pour désigner une chaîne de caractères.) est peut-être basé sur un usage similaire en anglais, comme « un collier de perles » ou « une file de voitures dans un former.")

```
awk '/foo/ { print $0 }' Liste BBS
```

Lorsque des lignes contenant « foo » sont trouvées, elles sont affichées, car « print $0 » signifie afficher le courant ligne. (Le simple fait d'écrire « imprimer » signifie la même chose, nous aurions donc pu écrire cela à la place.)

Vous remarquerez que des barres obliques, '/', entourent la chaîne 'foo' dans le programme awk. Les barres obliques indique que « foo » est un motif à rechercher. Ce type de motif est appelé une expression régulière, et est traité plus en détail ultérieurement (voir section 6.2 [Expressions régulières en tant que modèles], page 47). Les guillemets simples entourent le programme awk afin que le shell ne l'interprète pas comme une commande spéciale. personnages.

Voici ce que ce programme affiche :

```
pied de 555-1234 2400/1200/300 B
foire 555-6699 1200/300 B
Macfoo 555-6480 1200/300 UN
excusez-moi 555-2127 1200/300 C
```

Dans une règle awk, soit le motif, soit l'action peut être omis, mais pas les deux. Si le motif Si l'action est omise, elle est exécutée pour chaque ligne d'entrée. Si l'action est omise, le comportement par défaut est appliqué. L'action consiste à imprimer toutes les lignes qui correspondent au modèle.

Ainsi, nous pourrions omettre l'action (l'instruction d'impression et les accolades) dans ce qui précède.
Par exemple, le résultat serait le même : toutes les lignes correspondant au modèle « foo » seraient imprimées.
En comparaison, omettre l'instruction d'impression tout en conservant les accolades crée une action vide.
Cela ne fait rien ; alors aucune ligne ne sera imprimée.

2.2 Un exemple avec deux règles

L'utilitaire awk lit les fichiers d'entrée ligne par ligne. Pour chaque ligne, awk teste les motifs de
chacune des règles. Si plusieurs modèles correspondent, plusieurs actions sont exécutées, dans l'ordre où elles apparaissent.
Les modèles doivent apparaître dans le programme awk. Si aucun modèle ne correspond, aucune action n'est exécutée.

Après avoir traité toutes les règles (ou peut-être aucune) correspondant à la ligne, awk lit la ligne suivante.
(Voir toutefois la section 9.7 [Déclaration suivante], page 78). Ceci se poursuit jusqu'à la fin du fichier.
est atteint.

Par exemple, le programme awk :

```
/12/ { imprimer $0 }
/21/ { imprimer $0 }
```

contient deux règles. La première règle a la chaîne '12' comme modèle et 'print $0' comme action.
La deuxième règle a la chaîne « 21 » comme modèle et « print $0 » comme action. Chaque règle
L'action est encadrée par sa propre paire d'entretoises.

Ce programme awk affiche chaque ligne contenant la chaîne de caractères '12' ou '21'. Si une ligne
Contient les deux chaînes de caractères, elle est imprimée deux fois, une fois par chaque règle.

Si nous exécutons ce programme sur nos deux fichiers de données d'exemple, « BBS-list » et « inventory-shipped », comme
présenté ici :

```
awk '/12/ { print $0 }
     /21/ { imprimer $0 }' Liste BBS inventaire expédié
```

nous obtenons le résultat suivant :

```
oryctérope 555-5553 1200/300 B
piqûres de 555-3412 2400/1200/300 UN
mouches de 555-7685 1200/300 UN
bar en filet alpo 555-1675 2400/1200/300 UN
cœur 555-2912 1200/300 C
pied de 555-1234 2400/1200/300 B
foire 555-6699 1200/300 B
Macfoo 555-6480 1200/300 UN
se rendre 555-3430 2400/1200/300 UN
excusez-moi 555-2127 1200/300 C
sabafoo 555-2127 1200/300 C
21 janvier 36 64 620
21 avril 70 74 514
```

Notez que la ligne de « BBS-list » commençant par « sabafoo » a été imprimée deux fois, une fois pour chaque règle.

2.3 Un exemple plus complexe

Voici un exemple pour vous donner une idée du fonctionnement typique des programmes awk. Cet exemple montre comment utiliser awk pour résumer, sélectionner et réorganiser la sortie d'un autre utilitaire. Il utilise des fonctionnalités qui n'ont pas encore été abordées ; ne vous inquiétez donc pas si vous ne comprenez pas tous les détails.

```
ls-l | awk '$5 == "Nov" { somme += $4 }
                        FIN { imprimer la somme }'
```

Cette commande affiche le nombre total d'octets de tous les fichiers du répertoire courant dont la dernière modification remonte au mois de novembre (de n'importe quelle année). (Dans un shell C, il faut saisir un point-virgule suivi d'une barre oblique inverse à la fin de la première ligne ; dans un shell compatible POSIX, comme le shell Bourne ou le shell Bourne-Again, vous pouvez saisir l'exemple tel qu'indiqué.)

La commande « ls -l » de cet exemple permet d'afficher la liste des fichiers d'un répertoire, avec leur taille et leur date. Voici un exemple de résultat :

```
-rw-r--r-- 1 fermer                7 novembre 1933 13:05 Makefile
-rw-r--r-- 1 fermeture -rw-r--    10809 7 nov. 13:03 awk.h
r-- 1 fermeture                      983 13 avril 12:14 awk.tab.h
-rw-r--r-- 1 fermeture -rw-r--    31869 15 juin 12:20 awk.y 22414 7
r-- 1 fermeture                   novembre 13:03 awk1.c
-rw-r--r-- 1 fermeture -rw-r--    37455 7 nov. 13:03 awk2.c 27511 9
r-- 1 fermeture                   déc. 13:07 awk3.c
-rw-r--r-- 1 fermer                7989 7 nov. 13:03 awk4.c
```

Le premier champ contient les permissions de lecture et d'écriture, le deuxième le nombre de liens vers le fichier, et le troisième identifie le propriétaire du fichier. Le quatrième champ indique la taille du fichier en octets. Les cinquième, sixième et septième champs contiennent respectivement le mois, le jour et l'heure de la dernière modification du fichier. Enfin, le huitième champ contient le nom du fichier.

L'expression `$5 == "Nov"` dans notre programme awk vérifie si le cinquième champ de la sortie de `ls -l` correspond à la chaîne « Nov ». À chaque fois qu'une ligne contient « Nov » dans son cinquième champ, l'action `{ sum += $4 }` est exécutée. Cela ajoute la valeur du quatrième champ (la taille du fichier) à la variable `sum`.
Par conséquent, une fois qu'awk a terminé la lecture de toutes les lignes d'entrée, la variable `sum` correspond à la somme des tailles des fichiers dont les lignes correspondent au modèle. (Ceci est possible car les variables d'awk sont automatiquement initialisées à zéro.)

Une fois la dernière ligne de sortie de la commande `ls` traitée, la règle END est exécutée et la valeur de `sum` est affichée. Dans cet exemple, la valeur de `sum` est 80 600.

Ces techniques awk plus avancées sont abordées dans les sections suivantes (voir le chapitre 7 [Présentation des actions], page 55). Avant de passer à la programmation awk avancée, il est essentiel de comprendre comment awk interprète les entrées et affiche les sorties. En manipulant les champs et en utilisant les instructions d'affichage, vous pouvez générer des rapports très utiles et visuellement impressionnants.

2.4 Comment exécuter des programmes awk

Il existe plusieurs façons d'exécuter un programme awk. Si le programme est court, il est plus simple de l'inclure.
il s'agit de la commande qui exécute awk, comme ceci :

```
awk 'programme' fichier-entrée1 fichier-entrée2 . . .
```

où le programme se compose d'une série de modèles et d'actions, comme décrit précédemment.

Lorsque le programme est long, il est généralement plus pratique de le placer dans un fichier et de l'exécuter avec un
commande comme ceci :

```
awk -f fichier-programme fichier-entrée1 fichier-entrée2 . . .
```

2.4.1 Programmes awk jetables à usage unique

Une fois que vous maîtriserez awk, vous taperez souvent des programmes simples au moment où vous voudrez les utiliser. Vous pourrez alors écrire le programme comme premier argument de la commande awk, comme ceci :

```
awk 'programme' fichier-entrée1 fichier-entrée2 . . .
```

où le programme se compose d'une série de modèles et d'actions, comme décrit précédemment.

Ce format de commande indique à l'interpréteur de commandes de lancer awk et d'utiliser le programme pour traiter les enregistrements des fichiers d'entrée. Les guillemets simples autour du programme empêchent l'interpréteur de commandes d'interpréter les caractères d'awk comme des caractères spéciaux. Ils permettent également à l'interpréteur de commandes de traiter l'ensemble du programme comme un seul argument pour awk et autorisent ainsi un programme sur plusieurs lignes.

Ce format est également utile pour exécuter des programmes awk courts ou de taille moyenne depuis des scripts shell, car il évite d'avoir recours à un fichier séparé pour le programme awk. Un script shell autonome est plus fiable puisqu'il ne contient aucun autre fichier susceptible d'être égaré.

2.4.2 Exécution d'awk sans fichiers d'entrée

Vous pouvez également exécuter awk sans aucun fichier d'entrée. Si vous saisissez la ligne de commande :

```
awk 'programme'
```

Ensuite, awk applique le programme à l'entrée standard, c'est-à-dire généralement ce que vous saisissez dans le terminal. Cela continue jusqu'à ce que vous indiquiez la fin du fichier en appuyant sur Ctrl+D.

Par exemple, si vous exécutez cette commande :

```
awk '/th/'
```

Tout ce que vous saisirez ensuite sera considéré comme une donnée pour ce programme awk. Si vous saisissez les données suivantes :

```
Kathy
Ben
Tom
Beth
Seth
Karen
Thomas
Contrôle-d
```

awk affiche ensuite la sortie suivante :

```
Kathy
Beth
Seth
```

Il a reconnu « Thomas » comme correspondant au modèle « th ». Notez qu'il ne l'a pas reconnu.
Le langage awk est sensible à la casse et correspond exactement aux modèles.

2.4.3 Exécution de programmes longs

Parfois, vos programmes awk peuvent être très longs. Dans ce cas, il est plus pratique de les placer dans un bloc.
Enregistrez le programme dans un fichier séparé. Pour indiquer à awk d'utiliser ce fichier pour son programme, saisissez :

```
awk -f fichier-source fichier-entrée1 fichier-entrée2 . . .
```

L'option `-f` indique à l'utilitaire `awk` de récupérer le programme `awk` depuis le fichier `source-file`. N'importe quel nom de fichier peut être utilisé pour `source-file`. Par exemple, vous pouvez indiquer le programme :

```
/ème/
```

dans le fichier « th-prog ». Ensuite, exécutez cette commande :

```
awk -f th-prog
```

fait la même chose que celui-ci :

```
awk '/th/'
```

Comme expliqué précédemment (voir la section 2.4.2 [Exécution d'awk sans fichier d'entrée], page 16), il est important de noter que les guillemets simples ne sont généralement pas nécessaires autour du nom de fichier spécifié avec l'option `-f`, car la plupart des noms de fichiers ne contiennent aucun caractère spécial du shell. Par exemple, dans `th-prog`, le programme awk n'était pas entouré de guillemets simples. Ces guillemets ne sont nécessaires que pour les programmes fournis en ligne de commande avec awk.

Pour identifier clairement vos fichiers de programme awk, vous pouvez ajouter l'extension « .awk » à leur nom. Cela n'affecte pas l'exécution du programme awk, mais facilite sa gestion.

2.4.4 Programmes awk exécutables

Une fois que vous aurez appris awk, vous voudrez peut-être écrire des scripts awk autonomes, en utilisant le '#!' mécanisme de script. Vous pouvez le faire sur de nombreux systèmes Unix[1] (et un jour sur GNU).

Par exemple, vous pouvez créer un fichier texte nommé « hello », contenant ce qui suit (où « BEGIN » (c'est une fonctionnalité que nous n'avons pas encore abordée) :

```
#! /bin/awk -f

# un exemple de programme awk
COMMENCER { print "hello, world" }
```

Après avoir rendu ce fichier exécutable (avec la commande chmod), vous pouvez simplement taper :

```
Bonjour
```

dans le terminal, et le système se chargera d'exécuter awk2 comme si vous aviez tapé :

```
awk -f bonjour
```

Les scripts awk autonomes sont utiles lorsque vous souhaitez écrire un programme que les utilisateurs peuvent exécuter sans savoir que le programme est écrit en awk.

Si votre système ne prend pas en charge le mécanisme « #! », vous pouvez obtenir un effet similaire en utilisant une commande classique. Script shell. Cela ressemblerait à ceci :

```
Le deux-points garantit que ce script est exécuté par le shell Bourne. awk 'program' "$@"
```

En utilisant cette technique, il est essentiel d'encadrer le programme de guillemets simples pour le protéger de Interprétation par l'interpréteur de commandes. Sans guillemets, seul un expert en la matière pourra prédire le résultat.

Le symbole « $@ » permet à l'interpréteur de commandes de transmettre tous les arguments de la ligne de commande au programme awk, sans les interpréter. La première ligne, qui commence par deux points, permet à ce script shell de fonctionner même s'il est exécuté par un utilisateur utilisant l'interpréteur de commandes C.

2.5 Commentaires dans les programmes awk

Un commentaire est un texte inclus dans un programme à l'intention des lecteurs humains, et qui ne fait pas partie intégrante du programme. Les commentaires peuvent expliquer le fonctionnement du programme.

---

[1] Le mécanisme '#!' fonctionne sur les systèmes Unix dérivés de Berkeley Unix, System V Release 4 et certains systèmes System V Release 3.

[2] La ligne commençant par « #! » indique le chemin complet de l'interpréteur à exécuter, ainsi qu'un argument de ligne de commande initial optionnel. Le système d'exploitation exécute alors l'interpréteur avec l'argument fourni et la liste complète des arguments du programme. Le premier argument de la liste est le chemin complet du programme awk. Les arguments suivants peuvent être des options pour awk, des fichiers de données, ou les deux.

Presque tous les langages de programmation prévoient la possibilité d'utiliser des commentaires, car les programmes sont généralement difficiles à comprendre sans cette aide supplémentaire.

En langage awk, un commentaire commence par le caractère dièse « # » et se poursuit jusqu'à la fin de la ligne. Le reste de la ligne suivant un dièse est ignoré. Par exemple, on aurait pu insérer le code suivant dans « th-prog » :

```
# Ce programme recherche les enregistrements contenant le motif 'th'. C'est ainsi que # vous poursuivez
les commentaires sur les lignes suivantes. /th/
```

Vous pouvez également insérer des lignes de commentaires dans des programmes awk jetables composés au clavier, mais cela n'est généralement pas très utile ; le but d'un commentaire est d'aider vous ou une autre personne à comprendre le programme ultérieurement.

2.6 Déclarations awk versus Lignes

Le plus souvent, chaque ligne d'un programme awk correspond à une instruction ou une règle distincte, comme ceci :

```
awk '/12/ { print $0 } /21/ { print
        $0 }' Liste BBS inventaire expédié
```

Mais parfois, les instructions peuvent comporter plusieurs lignes, et les lignes peuvent contenir plusieurs instructions. Vous pouvez scinder une instruction en plusieurs lignes en insérant un saut de ligne après l'un des éléments suivants :

, { ? : || && faire autre

Tout autre saut de ligne est considéré comme la fin de l'instruction. (Le fait de couper les lignes après « ? » et « : » est une extension mineure de la syntaxe. Les caractères « ? » et « : » mentionnés ici font référence à l'expression conditionnelle à trois opérandes décrite dans la section 8.11 [Expressions conditionnelles], page 69.)

Pour scinder une instruction en deux lignes à l'endroit où un saut de ligne la terminerait, il suffit de terminer la première ligne par une barre oblique inverse, '\'. Cette opération est autorisée partout dans l'instruction, même au milieu d'une chaîne de caractères ou d'une expression régulière.
Par exemple:

```
awk '/Ce programme est trop long, continuez-le sur la ligne suivante/
  { print $1 }'
```

Dans les exemples de ce manuel, nous n'avons généralement pas utilisé la continuation par barre oblique inverse. Puisqu'il n'y a pas de limite de longueur de ligne dans awk, ce n'est jamais strictement nécessaire ; cela rend simplement les programmes plus lisibles. Nous avons préféré les rendre encore plus lisibles en limitant la longueur des instructions.
La continuation par barre oblique inverse est particulièrement utile lorsque votre programme awk se trouve dans un fichier source séparé, plutôt que d'être saisi directement dans la ligne de commande. Notez également que de nombreuses implémentations d'awk sont plus strictes quant à l'utilisation de cette continuation. Pour une portabilité optimale de vos programmes awk, il est préférable de ne pas couper vos lignes au milieu d'une expression régulière ou d'une chaîne de caractères.

Avertissement : la continuation avec barre oblique inverse ne fonctionne pas comme décrit ci-dessus avec le shell C. La continuation avec barre oblique inverse fonctionne pour les programmes awk dans les fichiers, ainsi que pour les programmes à exécution unique, à condition que vous

Vous utilisez un shell compatible POSIX, comme le shell Bourne ou le shell Bourne-again. Mais le shell C utilisé sur Berkeley Unix se comporte différemment ! Il faut alors utiliser deux barres obliques inverses consécutives, suivies d'un saut de ligne.

Lorsque les instructions awk d'une même règle sont courtes, vous pouvez en placer plusieurs sur une même ligne. Pour cela, séparez-les par un point-virgule, « ; ». Ceci s'applique également aux règles elles-mêmes. Ainsi, le programme précédent aurait pu s'écrire :

```
/12/ { print $0 } ; /21/ { print $0 }
```

Remarque : l'exigence que les règles figurant sur la même ligne soient séparées par un point-virgule est un changement récent du langage awk ; il a été apporté par souci de cohérence avec le traitement des instructions au sein d'une action.

2.7 Quand utiliser awk

Vous vous demandez peut-être en quoi awk peut vous être utile. Grâce à des programmes utilitaires supplémentaires, des modèles plus avancés, des séparateurs de champs, des opérations arithmétiques et d'autres critères de sélection, vous pouvez générer des résultats beaucoup plus complexes. Le langage awk est particulièrement utile pour produire des rapports à partir de grandes quantités de données brutes, comme par exemple pour synthétiser les informations issues de programmes utilitaires tels que ls. (Voir la section 2.3 [Un exemple plus complexe], page 15.)

Les programmes écrits avec awk sont généralement beaucoup plus petits que ceux écrits dans d'autres langages. Cela rend les programmes awk faciles à composer et à utiliser. Souvent, on peut les composer rapidement dans son terminal, les utiliser une seule fois, puis les supprimer. Comme les programmes awk sont interprétés, on évite le cycle de développement logiciel généralement long qui comprend l'édition, la compilation, les tests et le débogage.

Des programmes complexes ont été écrits en awk, notamment un assembleur entièrement adaptable pour microprocesseurs 8 bits (voir l'annexe C [Glossaire], page 121, pour plus d'informations) et un assembleur de microcode pour un ordinateur Prolog dédié. Cependant, les capacités d'awk sont mises à rude épreuve par des tâches d'une telle complexité.

Si vous écrivez des scripts awk de plus de quelques centaines de lignes, il serait judicieux d'envisager un autre langage de programmation. Emacs Lisp est un excellent choix si vous avez besoin de fonctionnalités avancées de correspondance de chaînes de caractères ou de motifs. Le shell est également performant pour ces opérations ; de plus, il permet une utilisation optimale des utilitaires système. Des langages plus classiques, tels que C, C++ et Lisp, offrent de meilleures possibilités pour la programmation système et la gestion de la complexité des grands programmes. Les programmes écrits dans ces langages peuvent nécessiter plus de lignes de code source que leurs équivalents en awk, mais ils sont plus faciles à maintenir et généralement plus performants.

3 Lecture des fichiers d'entrée

Dans un programme awk classique, toutes les données d'entrée sont lues soit depuis l'entrée standard (par défaut le clavier, mais souvent via un tube nommé provenant d'une autre commande), soit depuis des fichiers dont vous spécifiez le nom sur la ligne de commande awk. Si vous spécifiez des fichiers d'entrée, awk les lit séquentiellement, en lisant toutes les données de chacun avant de passer au suivant. Le nom du fichier d'entrée courant se trouve dans la variable intégrée FILENAME (voir le chapitre 13 [Variables intégrées], page 101).

Les données d'entrée sont lues par unités appelées enregistrements et traitées par les règles enregistrement par enregistrement. Par défaut, chaque enregistrement correspond à une ligne. Chaque enregistrement est automatiquement divisé en champs, ce qui facilite le traitement des différentes parties par les règles.

Dans de rares cas, vous devrez utiliser la commande getline, qui permet de saisir explicitement des données depuis un nombre quelconque de fichiers (voir la section 3.7 [Entrée explicite avec getline], page 30).

3.1 Comment les données d'entrée sont divisées en enregistrements

Le langage awk divise ses données d'entrée en enregistrements et en champs. Les enregistrements sont séparés par un caractère appelé séparateur d'enregistrements. Par défaut, ce séparateur est le caractère de nouvelle ligne, ce qui définit un enregistrement comme une seule ligne de texte.

Il peut arriver que vous souhaitiez utiliser un autre caractère pour séparer vos enregistrements. Pour ce faire, modifiez la variable intégrée RS. La valeur de RS est une chaîne de caractères indiquant comment séparer les enregistrements ; la valeur par défaut est « \n », qui contient uniquement un caractère de nouvelle ligne. C'est pourquoi, par défaut, les enregistrements sont affichés sur une seule ligne.

RS peut prendre n'importe quelle chaîne de caractères comme valeur, mais seul le premier caractère est utilisé comme séparateur d'enregistrements. Les autres caractères sont ignorés. RS est une exception à cet égard ; awk utilise la valeur complète de toutes ses autres variables intégrées.

Vous pouvez modifier la valeur de RS dans le programme awk à l'aide de l'opérateur d'affectation '=' (voir la section 8.7 [Expressions d'affectation], page 64). Le nouveau caractère de séparation d'enregistrements doit être placé entre guillemets pour former une constante de chaîne. Il est souvent préférable de procéder à cette modification au début de l'exécution, avant tout traitement d'entrée, afin que le premier enregistrement soit lu avec le séparateur approprié. Pour ce faire, utilisez le modèle spécial BEGIN (voir la section 6.7 [Modèles spéciaux BEGIN et END], page 53). Par exemple :

```
awk 'BEGIN { RS = "/" } ; { print $0 }' Liste BBS
```

Avant toute lecture d'entrée, la valeur de RS est modifiée en « / ». Cette chaîne de caractères commence par une barre oblique ; les enregistrements sont donc séparés par des barres obliques. Ensuite, le fichier d'entrée est lu, et la seconde règle du programme awk (l'action sans motif) affiche chaque enregistrement. Chaque instruction d'affichage ajoutant un saut de ligne à la fin de sa sortie, ce programme awk a pour effet de copier l'entrée en remplaçant chaque barre oblique par un saut de ligne.

Une autre façon de modifier le séparateur d'enregistrements consiste à utiliser la ligne de commande, en utilisant la variable : fonction d'affectation (voir chapitre 14 [Invocation d'awk], page 105).

```
awk '{ print $0 }' RS="/" Liste BBS
```

Cela configure RS sur '/' avant de traiter 'BBS-list'.

L'atteinte de la fin d'un fichier d'entrée met fin à l'enregistrement d'entrée actuel, même si le dernier caractère du fichier n'est pas le caractère présent dans RS.

La chaîne vide, "" (une chaîne ne contenant aucun caractère), a une signification particulière en tant que valeur de RS : elle indique que les enregistrements sont séparés uniquement par des lignes vides. Voir la section 3.6 [Enregistrements sur plusieurs lignes], page 29, pour plus de détails.

L'utilitaire awk comptabilise le nombre d'enregistrements lus dans le fichier d'entrée courant. Cette valeur est stockée dans une variable intégrée appelée FNR. Elle est remise à zéro à chaque ouverture d'un nouveau fichier. Une autre variable intégrée, NR, correspond au nombre total d'enregistrements lus dans tous les fichiers. Initialisée à zéro, elle n'est jamais automatiquement remise à zéro.

Si vous modifiez la valeur de RS au milieu d'une exécution awk, la nouvelle valeur est utilisée pour délimiter les enregistrements suivants, mais l'enregistrement en cours de traitement (et les enregistrements déjà traités) ne sont pas affectés.

3.2 Examen des domaines

Lorsqu'awk lit un enregistrement d'entrée, celui-ci est automatiquement divisé ou analysé par l'interpréteur en segments appelés champs. Par défaut, les champs sont séparés par des espaces, comme les mots sur une ligne.
Dans awk, les espaces blancs désignent toute chaîne d'un ou plusieurs espaces et/ou tabulations ; les autres caractères tels que les sauts de ligne, les sauts de formulaire, etc., qui sont considérés comme des espaces blancs par d'autres langages, ne sont pas considérés comme des espaces blancs par awk.

L'objectif des champs est de vous permettre de consulter plus facilement ces éléments du dossier.
Vous n'êtes pas obligé de les utiliser — vous pouvez opérer sur l'enregistrement entier si vous le souhaitez — mais ce sont les champs qui rendent les programmes awk simples si puissants.

Pour faire référence à un champ dans un programme awk, on utilise le signe dollar, « $ », suivi du numéro du champ souhaité. Ainsi, $1 fait référence au premier champ, $2 au deuxième, et ainsi de suite. Par exemple, supposons que la ligne d'entrée suivante soit :

Cela semble être un assez bon exemple.

Ici, le premier champ, ou $1, est « This » ; le deuxième champ, ou $2, est « seems » ; et ainsi de suite. Notez que le dernier champ, $7, est « example ». Comme il n'y a pas d'espace entre le « e » et le point, ce dernier est considéré comme faisant partie du septième champ.

Quel que soit le nombre de champs, le dernier champ d'un enregistrement peut être représenté par $NF. Ainsi, dans l'exemple ci-dessus, $NF correspond à $7, soit « exemple ». Le fonctionnement de ce mécanisme est expliqué ci-dessous (voir la section 3.3 [Numéros de champs non constants], page 23). Si vous tentez de faire référence à un champ au-delà du dernier, par exemple $8 alors que l'enregistrement ne comporte que 7 champs, vous obtiendrez une chaîne vide.

La variable NF simple, sans le symbole '$', est une variable intégrée dont la valeur correspond au nombre de champs dans l'enregistrement actuel.

$0, qui semble faire référence au champ zéro, est un cas particulier : il représente le Enregistrement d'entrée complet. C'est ce que vous utiliseriez si les champs ne vous intéressaient pas.

Voici d'autres exemples :

```
awk '$1 ~ /foo/ { print $0 }' Liste BBS
```

Cet exemple affiche chaque enregistrement du fichier « BBS-list » dont le premier champ contient la chaîne « foo ». une chaîne (ici, L'opérateur de correspondance (voir section 8.5 [Expressions de comparaison], page 62) vérifie si le champ $1) correspond à une expression régulière donnée.

En revanche, voici un exemple :

```
awk '/foo/ { print $1, $NF }' Liste BBS
```

recherche 'foo' dans l'enregistrement entier et imprime le premier champ et le dernier champ de chaque enregistrement d'entrée contenant une correspondance.

3.3 Numéros de champ non constants

Le numéro d'un champ n'a pas besoin d'être constant. Toute expression du langage awk peut être utilisée après le symbole « $ » pour faire référence à un champ. La valeur de l'expression spécifie le numéro du champ. Si la valeur est une chaîne de caractères et non un nombre, elle est convertie en nombre. Prenons l'exemple suivant :

```
awk '{ print $NR }'
```

Rappelons que NR représente le nombre d'enregistrements lus jusqu'à présent : 1 pour le premier enregistrement, 2 pour le deuxième, etc. Cet exemple affiche donc le premier champ du premier enregistrement, le deuxième champ du deuxième enregistrement, et ainsi de suite. Pour le vingtième enregistrement, le champ numéro 20 est affiché ; or, il est fort probable que cet enregistrement comporte moins de 20 champs, ce qui explique l'affichage d'une ligne vide.

Voici un autre exemple d'utilisation d'expressions comme numéros de champ :

```
awk '{ print $(2*2) }' Liste BBS
```

Le langage awk doit évaluer l'expression (2*2) et utiliser sa valeur comme numéro du champ à afficher. Le signe '*' représente la multiplication, donc l'expression 2*2 est évaluée à 4. Les parenthèses permettent d'effectuer la multiplication avant l'opération '$' ; elles sont nécessaires dès qu'un opérateur binaire est présent dans l'expression de numéro de champ. Cet exemple affiche donc les heures de fonctionnement (le quatrième champ) pour chaque ligne du fichier « BBS-list ».

Si le numéro de champ calculé est égal à zéro, vous obtenez l'enregistrement entier. Ainsi, $(2-2) a la même valeur que $0. Les numéros de champ négatifs ne sont pas autorisés.

Le nombre de champs de l'enregistrement courant est stocké dans la variable intégrée NF (voir chapitre 13 [Variables intégrées], page 101). L'expression $NF n'est pas une fonctionnalité particulière : elle résulte directement de l'évaluation de NF et de l'utilisation de sa valeur comme numéro de champ.

3.4 Modification du contenu d'un champ

Vous pouvez modifier le contenu d'un champ tel qu'il est interprété par awk au sein d'un programme awk ; cela modifie ce qu'awk considère comme l'enregistrement d'entrée courant. (L'entrée elle-même reste inchangée : awk ne modifie jamais le fichier d'entrée.)

Prenons cet exemple :

```
awk '{ $3 = $2 - 10; print $2, $3 }' inventaire expédié
```

Le signe « - » représente la soustraction, donc ce programme réaffecte le champ trois, $3, à la valeur du champ deux moins dix, $2 - 10. (Voir la section 8.3 [Opérateurs arithmétiques], page 60.) Ensuite, le champ deux et la nouvelle valeur du champ trois sont imprimés.

Pour que cela fonctionne, le texte du champ $2 doit pouvoir être interprété comme un nombre ; la chaîne de caractères doit être convertie en nombre pour que l'ordinateur puisse effectuer le calcul. Le nombre résultant de la soustraction est reconverti en chaîne de caractères, qui devient alors le champ 3. Voir la section 8.9 [Conversion de chaînes et de nombres], page 67.

Lorsque vous modifiez la valeur d'un champ (telle que perçue par awk), le texte de l'enregistrement d'entrée est recalculé pour inclure la nouvelle valeur du champ à la place de l'ancienne. Par conséquent, $0 est modifié pour refléter le champ modifié. Ainsi,

```
awk '{ $2 = $2 - 10; print $0 }' inventaire-expédié
```

imprime une copie du fichier d'entrée, en soustrayant 10 au deuxième champ de chaque ligne.

Vous pouvez également attribuer des valeurs à des champs hors limites. Par exemple :

```
awk '{ $6 = ($5 + $4 + $3 + $2) ; print $6 }' inventaire-expédié
```

Nous venons de créer la variable $6, dont la valeur correspond à la somme des valeurs des champs $2, $3, $4 et $5. Le signe « + » indique une addition. Dans le fichier « inventory-shipped », $6 représente le nombre total de colis expédiés pour un mois donné.

La création d'un nouveau champ modifie la copie interne awk de l'enregistrement d'entrée actuel — la valeur de 0 $. Ainsi, si vous exécutez la commande « print $0 » après avoir ajouté un champ, l'enregistrement imprimé inclut le nouveau champ, avec le nombre approprié de séparateurs de champs entre celui-ci et les champs existants.

Ce recalcul influe sur plusieurs fonctionnalités non encore abordées, notamment le séparateur de champs de sortie (OFS), utilisé pour séparer les champs (voir section 4.3 [Séparateurs de sortie], page 37), et NF (le nombre de champs ; voir section 3.2 [Examen des champs], page 22). Par exemple, la valeur de NF correspond au numéro du champ le plus élevé que vous créez.

Notez toutefois que le simple fait de faire référence à un champ hors plage ne modifie pas la valeur de l'un ou l'autre. $0 ou NF. La référence à un champ hors limites produit simplement une chaîne vide. Par exemple :

```
si ($(NF+1) != "") afficher
        "impossible" sinon
```

imprimer « tout est normal »

Il faudrait afficher « tout est normal », car NF+1 est forcément hors limites. (Pour plus d'informations sur les instructions if-else d'awk, voir la section 9.1 [L'instruction if], page 73.)

Il est important de noter que l'affectation d'une valeur à un champ modifiera la valeur de $0, mais ne modifiera pas la valeur de $0. La valeur NF, même lorsque vous affectez la chaîne nulle à un champ. Par exemple :

```
echo abcd | awk '{ OFS = ":"; $2 = "" ; imprimer ; imprimer NF }'
```

impressions

```
a::c:d
4
```

Le champ existe toujours, mais sa valeur est vide. On le remarque car il y a deux points dans un « a ».
rangée.

3.5 Spécification de la manière dont les champs sont séparés

(Cette section est assez longue ; elle décrit l'une des opérations les plus fondamentales d'awk. Si vous débutez avec awk, nous vous recommandons de relire cette section après avoir étudié la section sur les expressions régulières, section 6.2 [Expressions régulières en tant que modèles], page 47.)

La manière dont awk divise un enregistrement d'entrée en champs est contrôlée par le séparateur de champs, qui est un caractère unique ou une expression régulière. awk analyse l'enregistrement d'entrée à la recherche de correspondances avec le séparateur ; les champs eux-mêmes correspondent au texte situé entre ces correspondances. Par exemple, si le séparateur de champs est « oo », la ligne suivante sera :

```
moo goo gai pan
```

serait divisé en trois champs : 'm', 'g' et 'gai pan'.

Le séparateur de champs est représenté par la variable intégrée FS. Avis aux programmeurs shell ! awk n'utilise pas le nom IFS, utilisé par le shell.

Vous pouvez modifier la valeur de FS dans le programme awk à l'aide de l'opérateur d'affectation '=' (voir la section 8.7 [Expressions d'affectation], page 64). Il est souvent préférable de le faire au début de l'exécution, avant tout traitement d'entrée, afin que le premier enregistrement soit lu avec le séparateur approprié. Pour ce faire, utilisez le modèle spécial BEGIN (voir la section 6.7 [Modèles spéciaux BEGIN et END], page 53). Par exemple, ici, nous attribuons à FS la valeur de la chaîne ",":

```
awk 'BEGIN { FS = "," } ; { print $2 }'
```

Étant donné la ligne d'entrée,

```
John Q. Smith, 29 Oak St., Walamazoo, MI 42139
```

Ce programme awk extrait la chaîne ' 29, rue Oak.

Il arrive que vos données d'entrée contiennent des caractères de séparation qui ne séparent pas les champs comme prévu. Par exemple, le nom de la personne dans l'exemple utilisé peut comporter un titre ou un suffixe, comme « John Q. Smith, LXIX ». Voici un exemple de données d'entrée contenant un tel nom :

John Q. Smith, LXIX, 29 Oak St., Walamazoo, MI 42139

Le programme d'exemple précédent extrairait « LXIX » au lieu de « 29 Oak St. ». Si vous vous attendiez à ce que le programme affiche l'adresse, vous seriez surpris. Choisissez donc soigneusement la mise en forme de vos données et vos caractères de séparation pour éviter ce genre de problème.

Comme vous le savez, par défaut, les champs sont séparés par des séquences d'espaces (espaces et tabulations), et non par des espaces simples : deux espaces consécutifs ne délimitent pas un champ vide. La valeur par défaut du séparateur de champs est une chaîne de caractères contenant un seul espace. Si cette valeur était interprétée de manière classique, chaque espace séparerait les champs, de sorte que deux espaces consécutifs créeraient un champ vide entre eux. En réalité, cela ne se produit pas car la valeur « espace simple » du séparateur de champs constitue un cas particulier : elle spécifie le mode de délimitation des champs par défaut.

Si FS est un autre caractère unique, comme ",", chaque occurrence de ce caractère sépare deux champs. Deux occurrences consécutives délimitent un champ vide. Si le caractère apparaît au début ou à la fin de la ligne, il délimite également un champ vide. L'espace est le seul caractère unique qui ne suit pas ces règles.

Plus généralement, la valeur de FS peut être une chaîne de caractères contenant n'importe quelle expression régulière. Ensuite, chaque La correspondance dans l'enregistrement pour l'expression régulière sépare les champs. Par exemple, l'affectation :

FS = ", \t"

Transforme chaque segment d'une ligne de saisie composé d'une virgule, d'un espace et d'une tabulation en séparateur de champs. (« \t » représente une tabulation.)

Prenons un exemple plus complexe d'expression régulière : supposons que vous souhaitiez utiliser des espaces simples pour séparer les champs, comme les virgules simples l'ont fait précédemment. Vous pouvez définir FS sur « [ ] ». Cette expression régulière correspond à un espace unique, et à rien d'autre.

Le système de fichiers peut être défini en ligne de commande. Utilisez l'argument « -F » pour cela. Par exemple :

awk -F, 'programme' fichiers-entrée

L'option `-f` définit le système de fichiers (FS) comme étant la virgule (,). Notez que l'argument utilise un « F » majuscule. À l'inverse, l'option `-f` spécifie un fichier contenant un programme awk. La casse est importante pour les options de commande : les options `-F` et `-f` sont indépendantes. Vous pouvez utiliser les deux simultanément pour définir l'argument `FS` et exécuter un programme awk depuis un fichier.

La valeur utilisée comme argument de « -F » est traitée exactement de la même manière que les affectations à la variable intégrée FS. Cela signifie que si le séparateur de champs contient des caractères spéciaux, ceux-ci doivent être échappés correctement. Par exemple, pour utiliser « \ » comme séparateur de champs, vous devez saisir :

```
# identique à FS = "\\"
awk -F\\\\ '. . .' fichiers . . .
```

Puisque '\' est utilisé pour les guillemets dans le shell, awk interprétera '-F\\'. Ensuite, awk traitera '\\' comme caractère d'échappement. caractères (voir section 8.1 [Expressions constantes], page 57), donnant finalement un seul '\' à utiliser pour le séparateur de champs.

Dans le cas particulier où le mode de compatibilité est utilisé (voir le chapitre 14 [Invocation d'awk], page 105), si le Si l'argument de '-F' est 't', alors FS est défini sur le caractère de tabulation. (En effet, si vous tapez '-F\t', Sans les guillemets, dans le shell, le caractère '\' est supprimé, donc awk en déduit que vous souhaitez réellement vos champs. Les caractères doivent être séparés par des tabulations, et non par des « t ». Utilisez l'option « -v FS="t" » sur la ligne de commande si vous le souhaitez vraiment. pour séparer vos champs par des « t ».

Par exemple, utilisons un fichier de programme awk appelé 'baud.awk' qui contient le modèle /300/, et l'action « imprimer 1 $ ». Voici le programme :

```
/300/ { imprimer $1 }
```

Définissons également FS sur le caractère '-', et exécutons le programme sur le fichier 'BBS-list'. La commande suivante affiche une liste des noms des forums fonctionnant à 300 bauds et les trois premiers chiffres de leurs numéros de téléphone :

```
awk -F- -f baud.awk Liste BBS
```

Il produit le résultat suivant :

```
fourmilier alpo 555

morsures de 555
mouches de bar 555
Camelot 555
cœur 555
pied de 555
foire 555
Macfoo 555
sdace 555
sabafoo 555
```

Notez la deuxième ligne de sortie. Si vous consultez le fichier original, vous verrez que la deuxième ligne Cela ressemblait à ceci :

```
alpo-net 555-3412 2400/1200/300 UN
```

Le tiret « - » figurant dans le nom du système a été utilisé comme séparateur de champs, au lieu du tiret « - » du numéro de téléphone initialement prévu. Cela démontre pourquoi il faut être prudent. choisir vos séparateurs de champs et d'enregistrements.

Le programme suivant recherche dans le fichier des mots de passe du système et affiche les entrées des utilisateurs qui Je n'ai pas de mot de passe :

```
awk -F: '$2 ==  "" ' /etc/passwd
```

Ici, l'option « -F » est utilisée en ligne de commande pour définir le séparateur de champs. Notez que les champs du fichier « /etc/passwd » sont séparés par des deux-points. Le deuxième champ représente le mot de passe chiffré d'un utilisateur ; s'il est vide, cela signifie que l'utilisateur n'a pas de mot de passe.

Conformément à la norme POSIX, awk doit se comporter comme si chaque enregistrement était divisé en champs au moment de sa lecture. Concrètement, cela signifie que vous pouvez modifier la valeur de FS après la lecture d'un enregistrement, mais avant que l'un de ses champs ne soit référencé. La valeur des champs (c'est-à-dire leur mode de division) doit refléter l'ancienne valeur de FS, et non la nouvelle.

Cependant, de nombreuses implémentations d'awk ne procèdent pas ainsi. Au lieu de cela, elles reportent la séparation des champs jusqu'à ce qu'une référence à un champ soit effectivement effectuée, en utilisant la valeur actuelle de FS ! Ce comportement peut être difficile à diagnostiquer. L'exemple suivant illustre les résultats des deux méthodes. (La commande sed affiche uniquement la première ligne de '/etc/passwd'.)

```
sed 1q /etc/passwd | awk '{ FS = ":" ; print $1 }'
```

imprimera généralement

```
racine
```

en cas d'implémentation incorrecte d'awk, gawk affichera quelque chose comme

```
racine:nSijPlPhZZwgE:0:0:Racine:/:
```

entre les deux cas de `FS = "[ \t]+"` (qui est une expression régulière Il existe une différence importante correspondant à un ou plusieurs espaces ou tabulations). Pour les deux valeurs de `FS`, les champs sont séparés par des suites d'espaces et/ou de tabulations. Cependant, lorsque la valeur de `FS` est `""`, awk supprime les espaces de début et de fin de l'enregistrement, puis détermine l'emplacement des champs.

Par exemple, l'expression suivante affiche 'b' :

```
écho '  abcd '  | awk '{ print $2 }'
```

Cependant, les impressions suivantes affichent « a » :

```
écho '  abcd '  | awk 'BEGIN { FS = "[ \t]+" } ; { print $2 }'
```

Dans ce cas, le premier champ est nul.

La suppression des espaces blancs de début et de fin entre également en jeu chaque fois que 0 $ est recalculé. Par exemple, ce pipeline

```
écho '  abc d' | awk '{ print; $2 = $2; print }'
```

produit le résultat suivant :

```
abcd
```

abcd

La première instruction d'impression affiche l'enregistrement tel qu'il a été lu, en conservant les espaces initiaux. L'affectation à $2 reconstruit $0 en concaténant $1 à $NF, séparés par la valeur de OFS. Comme les espaces initiaux ont été ignorés lors de la recherche de $1, ils ne font pas partie du nouveau $0.
Enfin, la dernière instruction d'impression affiche le nouveau montant de 0 $.

Le tableau suivant récapitule la répartition des champs en fonction de la valeur de FS.

FS == Les champs sont séparés par des séquences d'espaces. Les espaces en début et en fin de champ sont ignorés.
Il s'agit de la valeur par défaut.

FS == n'importe quel caractère unique.
Les champs sont séparés par chaque occurrence du caractère. Plusieurs occurrences successives délimitent des champs vides, de même que les occurrences en début et en fin de champ.

FS == regexp Les
champs sont séparés par les occurrences de caractères correspondant à l'expression régulière. Les correspondances en début et en fin d'expression régulière délimitent les champs vides.

3.6 Enregistrements multilignes

Dans certaines bases de données, une seule ligne ne peut pas contenir facilement toutes les informations dans une seule entrée.
Dans de tels cas, vous pouvez utiliser des enregistrements multilignes.

La première étape consiste à choisir le format de vos données : lorsque les enregistrements ne sont pas définis comme des enregistrements uniques
Comment définir les lignes ? Qu'est-ce qui doit séparer les enregistrements ?

Une technique consiste à utiliser un caractère ou une chaîne de caractères inhabituel(le) pour séparer les enregistrements. Par exemple, vous pouvez utiliser le caractère de saut de page (écrit \f en awk, comme en C) pour les séparer, chaque enregistrement correspondant alors à une page du fichier. Pour ce faire, il suffit d'affecter à la variable RS la valeur « \f » (une chaîne de caractères contenant le caractère de saut de page). Tout autre caractère peut être utilisé, à condition qu'il ne fasse pas partie des données d'un enregistrement.

Une autre technique consiste à utiliser des lignes vides pour séparer les enregistrements. Par exception, une chaîne vide comme valeur de RS indique que les enregistrements sont séparés par une ou plusieurs lignes vides. Si vous définissez RS sur une chaîne vide, un enregistrement se termine toujours à la première ligne vide rencontrée. L'enregistrement suivant ne commence qu'à la première ligne non vide qui suit ; peu importe le nombre de lignes vides consécutives, elles sont considérées comme un seul séparateur d'enregistrements. (La fin du fichier est également considérée comme un séparateur d'enregistrements.)

La deuxième étape consiste à séparer les champs de l'enregistrement. Une façon d'y parvenir est de placer chaque champ sur une ligne distincte : pour ce faire, il suffit d'affecter à la variable FS la chaîne « \n ». (Cette expression régulière simple correspond à un saut de ligne.)

Une autre façon de séparer les champs consiste à diviser chacune des lignes en champs de manière classique.
Ce comportement est automatique grâce à une fonctionnalité particulière : lorsque RS est défini sur une chaîne vide, le caractère de nouvelle ligne sert systématiquement de séparateur de champs. Ceci s'ajoute aux séparations de champs résultant de FS.

La motivation initiale de cette exception particulière était probablement de permettre un comportement utile dans le cas par défaut (c.-à-d. FS == " "). Cette fonctionnalité peut poser problème si vous ne souhaitez pas que

Le caractère de nouvelle ligne est utilisé pour séparer les champs, car il est impossible de l'empêcher. Cependant, vous pouvez contourner ce problème en utilisant la fonction split pour diviser manuellement l'enregistrement (voir la section 11.3 [Fonctions intégrées pour la manipulation de chaînes], page 90).

3.7 Saisie explicite avec getline

Jusqu'à présent, nous avons utilisé le flux d'entrée principal d'awk pour récupérer nos fichiers d'entrée : soit l'entrée standard (généralement votre terminal), soit les fichiers spécifiés sur la ligne de commande. Le langage awk possède une commande intégrée spéciale, `getline`, qui permet de lire des entrées de manière contrôlée.

Cette commande est assez complexe et ne devrait pas être utilisée par les débutants. Elle est abordée ici car ce chapitre est consacré à la saisie. Les exemples qui suivent l'explication de la commande `getline` contiennent des notions qui n'ont pas encore été traitées. Par conséquent, revenez à la commande `getline` après avoir consulté le reste de ce manuel et acquis une bonne compréhension du fonctionnement d'`awk`.

La fonction getline renvoie 1 si elle trouve un enregistrement, et 0 si elle rencontre la fin du fichier. S'il y a un en cas d'erreur lors de la récupération d'un enregistrement, par exemple un fichier qui ne peut pas être ouvert, getline renvoie −1.

Dans les exemples suivants, « command » représente une chaîne de caractères correspondant à une commande shell.

La commande `getline` peut être utilisée sans argument pour lire les données du fichier d'entrée courant. Dans ce cas, elle lit simplement l'enregistrement suivant et le divise en champs. Ceci est utile si vous avez terminé le traitement de l'enregistrement courant, mais que vous souhaitez effectuer un traitement spécifique sur l'enregistrement suivant. Voici un exemple :

```
awk '{ if
        (t = index($0, "/*")) { if (t > 1) tmp =
                substr($0, 1, t
                        - 1)
                autre
                                ""
                        tmp = u
                = index(substr($0, t + 2), "*/") tant que (u == 0) { getline t = -1 u
                = index($0, "*/")




                }
                si (u <= longueur($0) - 2) $0 = tmp
                        substr($0, t + u + 3)
                autre
                        $0 = tmp
        }
        imprimer $0
}'
```

Ce programme awk supprime tous les commentaires de style C, '/* . . . */', de l'entrée. En remplaçant 'print $0' par d'autres instructions, vous pouvez effectuer des traitements plus complexes sur l'entrée décommentée, comme la recherche d'occurrences d'une expression régulière. (Ce programme comporte un problème subtil ; saurez-vous le repérer ?)

Cette forme de la commande getline définit NF (le nombre de champs ; voir la section 3.2 [Examen des champs], page 22), NR (le nombre d'enregistrements lus jusqu'à présent ; voir la section 3.1 [Comment l'entrée est divisée en enregistrements], page 21), FNR (le nombre d'enregistrements lus à partir de ce fichier d'entrée) et la valeur de $0.

Remarque : la nouvelle valeur de $0 est utilisée pour tester les modèles des règles suivantes. La valeur initiale de $0 qui a déclenché la règle exécutant getline est perdue. En revanche, l'instruction suivante lit un nouvel enregistrement mais commence immédiatement à le traiter normalement, en commençant par la première règle du programme. Voir la section 9.7 [L'instruction suivante], page 78.

La fonction `getline

var` lit un enregistrement et le stocke dans la variable `var`. Ceci est utile lorsque vous souhaitez que votre programme lise l'enregistrement suivant du fichier d'entrée courant, sans pour autant le soumettre au traitement d'entrée habituel.

Par exemple, supposons que la ligne suivante soit un commentaire ou une chaîne de caractères spéciale, et que vous souhaitiez la lire tout en vous assurant qu'elle ne déclenche aucune règle. Cette version de `getline` vous permet de lire cette ligne et de la stocker dans une variable afin qu'elle ne soit jamais traitée par la boucle principale de `awk` qui lit une ligne et vérifie chaque règle.

L'exemple suivant inverse les lignes d'entrée une à deux. Par exemple, étant donné :

```
Van
tew
phore
libre il
```

produit :

```
tew
Van
phore
libre
```

Voici le programme : awk '{ if

```
((getline
        tmp) > 0) { print tmp print $0 } else print $0




}'
```

La fonction getline utilisée de cette manière ne modifie que les variables NR et FNR (et bien sûr, var). L'enregistrement n'est pas divisé en champs ; par conséquent, les valeurs des champs (y compris $0) et la valeur de NF restent inchangées.

La forme `<file>` de

la fonction `getline` prend comme entrée le fichier `file`. Ici, `file` est une expression de type chaîne de caractères qui spécifie le nom du fichier. L'opérateur `<file>` est appelé une redirection car il redirige l'entrée vers une autre source.

Ce formulaire est utile si vous souhaitez lire vos données d'entrée à partir d'un fichier spécifique, plutôt que du flux d'entrée principal. Par exemple, le programme suivant lit son enregistrement d'entrée à partir du fichier « foo.input » lorsqu'il rencontre un premier champ dont la valeur est égale à 10 dans le fichier d'entrée courant. awk '{ if ($1 == 10) { getline < "foo.input"

```
        imprimer }
            sinon imprimer
}'
```

Étant donné que le flux d'entrée principal n'est pas utilisé, les valeurs de NR et FNR ne sont pas modifiées. Mais l'enregistrement lu est divisé en champs de manière classique, ce qui modifie les valeurs de $0 et des autres champs. Il en va de même pour la valeur de NF.

Cela n'entraîne pas la vérification de l'enregistrement par rapport à tous les modèles du programme awk, comme cela se produirait si l'enregistrement était lu normalement par la boucle de traitement principale d'awk. Cependant, le nouvel enregistrement est vérifié par rapport à toutes les règles ultérieures, tout comme lorsque getline est utilisé sans redirection.

La forme `getline` de la

fonction `getline` lit le fichier `file` et le stocke dans la variable `var`. Comme précédemment, `file` est une expression de type chaîne de caractères qui spécifie le fichier à lire.

Dans cette version de getline, aucune des variables intégrées n'est modifiée et l'enregistrement n'est pas divisé en champs. La seule variable modifiée est var.

Par exemple, le programme suivant copie tous les fichiers d'entrée vers le fichier de sortie, à l'exception des enregistrements contenant « @include nom_du_fichier ». Ces enregistrements sont remplacés par le contenu du fichier nom_du_fichier.

```
awk '{ if
    (NF == 2 && $1 == "@include") {
        tant que ((getline line < $2) > 0) afficher la ligne
            close($2) }
        sinon afficher


}'
```

Notez ici que le nom du fichier d'entrée supplémentaire n'est pas intégré au programme ; il est extrait des données, du deuxième champ de la ligne '@include'.

La fonction close est appelée afin de garantir que si deux lignes « @include » identiques apparaissent dans l'entrée, le fichier spécifié soit inclus deux fois. Voir la section 3.8 [Fermeture des fichiers d'entrée et des tubes], page 33.

L'un des défauts de ce programme est qu'il ne traite pas les instructions '@include' imbriquées comme le ferait un véritable préprocesseur de macros.

commande | getline

Vous pouvez rediriger la sortie d'une commande vers `getline`. Un tube (ou pipe) permet de relier la sortie d'un programme à l'entrée d'un autre. Ici, la commande `string` est exécutée comme une commande shell et sa sortie est redirigée vers `awk` pour être utilisée comme entrée. Cette forme de `getline` lit un enregistrement depuis le tube.

Par exemple, le programme suivant copie l'entrée vers la sortie, à l'exception des lignes commençant par « @execute », qui sont remplacées par la sortie produite en exécutant le reste de la ligne comme une commande shell :

```
awk '{ if
    ($1 == "@execute") { tmp =
        substr($0, 10) while ((tmp |
        getline) > 0) print close(tmp) } else
```

```
                imprimer
}'
```

La fonction close est appelée pour garantir que si deux lignes identiques '@execute' apparaissent dans Pour chaque entrée, la commande est exécutée. Voir la section 3.8 [Fermeture des fichiers d'entrée et Tuyaux], page 33.

Compte tenu des données d'entrée :

```
nourriture
bar
base
@exécuter qui
bletch
```

Le programme pourrait produire :

```
nourriture
bar
base
pirater  ttyv0 13 juil. 14:22
pirater  ttyp0 13 juil. 14:23 ttyp1 13 juil.  (gnu:0)
pirater  14:23 ttyp2 13 juil. 14:23 ttyp3     (gnu:0)
pirater  13 juil. 14:23                       (gnu:0)
pirater                                       (gnu:0)
bletch
```

Notez que ce programme a exécuté la commande « who » et a affiché le résultat. (Si vous essayez ceci) Programmez vous-même, vous obtiendrez des résultats différents, vous montrant qui est connecté à votre compte. système.)

Cette variante de getline divise l'enregistrement en champs, définit la valeur de NF et recalcule la valeur de $0. Les valeurs de NR et FNR restent inchangées.

commande | obtenir la variable de ligne

La sortie de la commande est envoyée via un tube à getline et dans le variable var. Par exemple, le programme suivant lit la date et l'heure actuelles dans la variable current_time, en utilisant l'utilitaire de date, puis l'affiche.

```
awk 'DÉBUT {
        "date" | getline heure_actuelle
        fermer("date")
        imprimer « Rapport imprimé sur " heure_actuelle
}'
```

Dans cette version de getline, aucune des variables intégrées n'est modifiée, et l'enregistrement n'est pas divisé en champs.

3.8 Fermeture des fichiers d'entrée et des canaux

Si le même nom de fichier ou la même commande shell est utilisé plusieurs fois avec getline pendant Lors de l'exécution d'un programme awk, le fichier est ouvert (ou la commande est exécutée) uniquement en premier. à ce moment-là, le premier enregistrement d'entrée est lu à partir de ce fichier ou de cette commande. La fois suivante, Si le même fichier ou la même commande est utilisé dans getline, un autre enregistrement est lu à partir de celui-ci, et ainsi de suite.

Cela signifie que si vous souhaitez recommencer à lire le même fichier depuis le début, ou si vous Si vous souhaitez réexécuter une commande shell (plutôt que de lire davantage de résultats de la commande), vous devez Vous devez prendre des mesures particulières. Il vous suffit d'utiliser la fonction de fermeture, comme suit :

```
fermer(nom_de_fichier)
```

ou

```
fermer(commande)
```

L'argument `filename` ou `command` peut être n'importe quelle expression. Sa valeur doit correspondre exactement à la chaîne de caractères utilisée pour ouvrir le fichier ou lancer la commande ; par exemple, si vous ouvrez un tube avec ceci :

```
"sort -r noms" | getline foo
```

vous devez alors le terminer ainsi :

```
fermer("sort -r noms")
```

Une fois cet appel de fonction exécuté, le prochain getline de ce fichier ou de cette commande rouvrira le fichier ou réexécutera la commande.

La fonction close renvoie la valeur zéro si la fermeture a réussi. Sinon, la valeur sera différente de zéro.

4 Sortie d'impression

L'une des fonctions les plus courantes des actions est d'afficher ou d'imprimer tout ou partie des données d'entrée. Pour un affichage simple, utilisez l'instruction print. Pour une mise en forme plus élaborée, utilisez l'instruction printf. Les deux sont décrits dans ce chapitre.

4.1 Déclaration d'impression

L'instruction d'affichage produit un résultat avec une mise en forme simple et standardisée. Il suffit de spécifier les chaînes de caractères ou les nombres à afficher, dans une liste séparée par des virgules. Ils sont affichés séparés par des espaces, suivis d'un saut de ligne. Voici à quoi ressemble l'instruction :

```
imprimer l'élément 1, l'élément 2, . . .
```

La liste complète des éléments peut être placée entre parenthèses. Ces parenthèses sont nécessaires si l'une des expressions d'un élément utilise un opérateur relationnel ; sinon, il pourrait y avoir confusion avec une redirection (voir la section 4.6 [Redirection de la sortie de print et printf], page 42). Les opérateurs relationnels sont : '==', '!=', '<', '>', '>=', '<=', '~' et '!~' (voir la section 8.5 [Expressions de comparaison], page 62).

Les éléments affichés peuvent être des chaînes de caractères ou des nombres constants, des champs de l'enregistrement courant (comme $1), des variables ou des expressions awk. L'instruction `print` est totalement générique et permet de déterminer les valeurs à afficher. À deux exceptions près, il est impossible de spécifier la mise en forme : le nombre de colonnes, l'utilisation ou non de la notation exponentielle, etc. (Voir la section 4.3 [Séparateurs de sortie], page 37, et la section 4.4 [Contrôle de l'affichage numérique avec `print`], page 37.) Pour cela, il faut utiliser l'instruction `printf` (voir la section 4.5 [Utilisation des instructions `printf` pour un affichage plus précis], page 38).

La simple instruction « print » sans élément est équivalente à « print $0 » : elle imprime l'intégralité du texte. Enregistrement actuel. Pour afficher une ligne vide, utilisez « print "" », où "" représente la chaîne vide.

Pour afficher un texte fixe, utilisez une chaîne de caractères constante comme « Bonjour » comme élément. Si vous oubliez les guillemets, votre texte sera interprété comme une expression awk et vous obtiendrez probablement une erreur. Notez qu'un espace est inséré entre chaque élément.

Le plus souvent, chaque instruction d'affichage produit une seule ligne. Cependant, ce n'est pas limité à une seule ligne. Si la valeur d'un élément est une chaîne de caractères contenant un saut de ligne, ce dernier est affiché avec le reste de la chaîne. Une seule instruction d'affichage peut ainsi produire un nombre quelconque de lignes.

4.2 Exemples d'instructions d'impression

Voici un exemple d'affichage d'une chaîne de caractères contenant des sauts de ligne intégrés :

```
awk 'BEGIN { print "ligne une\nligne deux\nligne trois" }'
```

produit un résultat comme celui-ci :

```
ligne une
```

```
ligne deux
ligne trois
```

Voici un exemple qui affiche les deux premiers champs de chaque enregistrement d'entrée, séparés par un espace :

```
awk '{ print $1, $2 }' inventaire expédié
```

Son résultat ressemble à ceci :

```
13 janvier
15 février
15 mars
. . .
```

Une erreur fréquente lors de l'utilisation de l'instruction `print` est d'omettre la virgule entre deux éléments. Cela a souvent pour effet d'afficher les éléments collés les uns aux autres, sans espace. En effet, juxtaposer deux expressions de type chaîne de caractères dans awk revient à les concaténer. Par exemple, sans la virgule :

```
awk '{ print $1 $2 }' inventaire expédié
```

impressions :

```
13 janvier
15 février
15 mars
. . .
```

Le résultat de ces deux exemples n'est pas très compréhensible pour une personne qui ne connaît pas le fichier « inventory-shipped ». Un titre en début de tableau le rendrait plus clair. Ajoutons des titres à notre tableau des mois (1 $) et des caisses vertes expédiées (2 $). Pour cela, nous utilisons le modèle BEGIN (voir section 6.7 [Modèles spéciaux BEGIN et END], page 53) afin que les titres ne soient imprimés qu'une seule fois.

```
awk 'BEGIN { print "Caisses du mois" print
                                        "----- ------" } { print $1,
                $2 }' inventaire expédié
```

Avez-vous déjà deviné ce qui se passe ? Ce programme affiche le résultat suivant :

```
Caisses mensuelles
— —
13 janvier
15 février
15 mars
. . .
```

Les en-têtes et les données du tableau ne sont pas alignés ! Nous pouvons corriger cela en ajoutant des espaces entre les deux champs :

```
awk 'BEGIN { print "Month Crates" "----- ------" } print
                inventory-{ print $1, " ", $2 }'
            shipped
```

Vous imaginez bien que cette méthode d'alignement des colonnes peut vite devenir complexe lorsqu'il y a de nombreuses colonnes à aligner. Compter les espaces pour deux ou trois colonnes est simple, mais au-delà, on peut facilement s'y perdre. C'est pourquoi l'instruction `printf` a été créée (voir la section 4.5 [Utilisation des instructions `printf` pour une impression plus soignée], page 38) ; l'une de ses particularités est l'alignement des colonnes de données.

4.3 Séparateurs de sortie

Comme indiqué précédemment, une instruction d'impression contient une liste d'éléments séparés par des virgules. Dans le résultat, les éléments sont généralement séparés par un espace. Toutefois, il n'est pas obligatoire d'utiliser des espaces ; l'espace est le comportement par défaut. Vous pouvez spécifier n'importe quelle chaîne de caractères à utiliser comme séparateur de champs de sortie en définissant la variable intégrée OFS. La valeur initiale de cette variable est la chaîne « », c'est-à-dire un simple espace.

Le résultat d'une instruction d'affichage complète est appelé un enregistrement de sortie. Chaque instruction d'affichage produit un enregistrement de sortie, suivi d'une chaîne de caractères appelée séparateur d'enregistrements de sortie. La variable intégrée ORS spécifie cette chaîne. Sa valeur initiale est la chaîne « \n » contenant un caractère de nouvelle ligne ; ainsi, chaque instruction d'affichage crée généralement une ligne distincte.

Vous pouvez modifier la façon dont les champs et les enregistrements de sortie sont séparés en attribuant de nouvelles valeurs aux variables OFS et/ou ORS. Il est généralement conseillé de le faire dans la règle BEGIN (voir la section 6.7 [Modèles spéciaux BEGIN et END], page 53), afin que cette opération soit effectuée avant tout traitement des données d'entrée. Vous pouvez également le faire en effectuant des affectations sur la ligne de commande, avant les noms de vos fichiers d'entrée.

L'exemple suivant affiche le premier et le deuxième champ de chaque enregistrement d'entrée, séparés par un point-virgule, avec une ligne vide ajoutée après chaque ligne :

```
awk 'BEGIN { OFS = ";"; ORS = "\n\n" } { print $1, $2 }' Liste
            BBS
```

Si la valeur de ORS ne contient pas de saut de ligne, toutes vos sorties seront exécutées ensemble sur une seule ligne. ligne, sauf si vous générez les sauts de ligne d'une autre manière.

4.4 Contrôle de la sortie numérique avec impression

Lorsque vous utilisez l'instruction print pour afficher des valeurs numériques, awk convertit en interne le nombre en une chaîne de caractères, puis affiche cette chaîne. awk utilise la fonction sprintf pour effectuer cette conversion.
Pour l'instant, il suffit de dire que la fonction `printf` accepte une spécification de format qui lui indique comment formater les nombres (ou les chaînes de caractères), et qu'il existe plusieurs façons de formater les nombres. Les différentes spécifications de format sont décrites plus en détail dans la section 4.5 [Utilisation des instructions `printf` pour un affichage plus soigné], page 38.

La variable intégrée OFMT contient la spécification de format par défaut utilisée par print avec sprintf lorsqu'il souhaite convertir un nombre en chaîne de caractères pour l'impression. En fournissant des spécifications de format différentes

En modifiant la valeur de OFMT, vous pouvez changer la façon dont vos nombres seront imprimés. Voici un bref exemple :

```
awk 'BEGIN { OFMT = "%d" # afficher les nombres sous forme d'entiers print
                         17.23 }'
```

imprimera « 17 ».

4.5 Utilisation des instructions printf pour une impression plus sophistiquée

Si vous souhaitez un contrôle plus précis du format de sortie que celui offert par la commande print, utilisez printf. Avec `printf`, vous pouvez définir la largeur de chaque élément et personnaliser l'affichage des nombres (par exemple, la base, l'affichage de l'exposant et du signe, et le nombre de chiffres après la virgule). Pour cela, vous spécifiez une chaîne de caractères, appelée chaîne de format, qui détermine comment et où afficher les autres arguments.

4.5.1 Introduction à l'instruction printf

L'instruction printf ressemble à ceci :

```
format printf, élément1, élément2, . . .
```

L'ensemble des arguments peut être placé entre parenthèses. Ces parenthèses sont nécessaires si l'une des expressions d'élément utilise un opérateur relationnel ; sinon, il pourrait y avoir confusion avec une redirection (voir la section 4.6 [Redirection de la sortie de print et printf], page 42). Les opérateurs relationnels sont : '==', '!=', '<', '>', '>=', '<=', '~' et '!~' (voir la section 8.5 [Expressions de comparaison], page 62).

La différence entre `printf` et `print` réside dans le format des arguments. Il s'agit d'une expression dont la valeur est interprétée comme une chaîne de caractères ; elle spécifie comment afficher chacun des autres arguments. On l'appelle la chaîne de format.

La chaîne de format est identique à celle de la fonction `printf` de la bibliothèque ANSI C. Elle contient principalement du texte à afficher tel quel. Ce texte est parsemé de spécificateurs de format, un par élément. Chaque spécificateur indique d'afficher l'élément suivant à cet emplacement.

L'instruction printf n'ajoute pas automatiquement de saut de ligne à sa sortie. Elle n'affiche que le contenu spécifié dans le format. Par conséquent, si vous souhaitez un saut de ligne, vous devez l'inclure dans le format. Les variables de séparation de sortie OFS et ORS sont sans effet sur les instructions printf.

4.5.2 Lettres de contrôle de format

Un spécificateur de format commence par le caractère « % » et se termine par une lettre de contrôle de format ; il indique à l'instruction printf comment afficher un élément. (Si vous souhaitez afficher un « % », écrivez « %% ».)
La lettre de contrôle de format spécifie le type de valeur à imprimer. Le reste du spécificateur de format est composé de modificateurs optionnels, tels que la largeur du champ à utiliser.

Voici la liste des lettres de contrôle de format :

'c' Cela affiche un nombre sous forme de caractère ASCII. Ainsi, « printf "%c", 65 » affiche le résultat suivant : Lettre « A ». Le résultat pour une valeur de chaîne est le premier caractère de la chaîne.

'd' Cela affiche un entier décimal.

'je' Cela affiche également un entier décimal.

'et' Ceci affiche un nombre en notation scientifique (exponentielle). Par exemple,

```
printf "%4.3e", 1950
```

affiche « 1,950e+03 », avec un total de quatre chiffres significatifs dont trois suivent le virgule décimale. Les « 4,3 » sont des modificateurs, expliqués ci-dessous.

'f' Ceci affiche un nombre en notation à virgule flottante.

'g' Cela affiche un nombre en notation scientifique ou en notation à virgule flottante, selon le cas. utilise moins de caractères.

'le' Ceci affiche un entier octal non signé.

's' Ceci imprime une chaîne de caractères.

'x' Ceci affiche un entier hexadécimal non signé.

'X' Cela affiche un entier hexadécimal non signé. Cependant, pour les valeurs de 10 à 15, il utilise les lettres « A » à « F » au lieu de « a » à « f ».

'%' Il ne s'agit pas à proprement parler d'une lettre de contrôle de format, mais elle a une signification lorsqu'elle est utilisée après une '%' : la séquence '%%' produit un seul '%'. Elle ne prend pas d'argument.

4.5.3 Modificateurs pour les formats printf

Une spécification de format peut également inclure des modificateurs permettant de contrôler la valeur de l'élément. est imprimé et l'espace qui lui est alloué. Les modificateurs se placent entre le symbole « % » et le contrôle de format. lettre. Voici les modificateurs possibles, dans l'ordre où ils peuvent apparaître :

'-' Le signe moins, placé avant le modificateur de largeur, indique de justifier à gauche l'argument à l'intérieur sa largeur spécifiée. Normalement, l'argument est imprimé aligné à droite dans la largeur spécifiée. largeur. Ainsi,

```
printf "%-4s", "foo"
```

imprime 'foo'.

'largeur' Il s'agit d'un nombre représentant la largeur souhaitée d'un champ. Saisissez n'importe quel nombre. L'insertion du caractère « % » entre le signe « % » et le caractère de contrôle de format force l'expansion du champ. cette largeur. La méthode par défaut consiste à ajouter des espaces à gauche. Par exemple,

```
printf "%4s", "foo"
```

La ' foo'.

valeur de largeur affichée est une largeur minimale, et non maximale. Si la valeur de l'élément nécessite Plus qu'une simple question de largeur de caractères, sa largeur peut être aussi importante que nécessaire. Ainsi,

```
printf "%4s", "foobar"
```

imprime 'foobar'.

Faire précéder la largeur d'un signe moins permet d'ajouter des espaces à la sortie. à droite, et non à gauche.

'.prec' Ce nombre spécifie la précision d'impression. Il indique le nombre de chiffres à afficher après la virgule. Pour une chaîne de caractères, il spécifie le nombre maximal de caractères à imprimer.

La capacité de la fonction printf de la bibliothèque C à gérer la largeur et la précision dynamiques (par exemple, "%*.*s") est prise en charge. Au lieu de fournir explicitement les valeurs de largeur et/ou de précision dans la chaîne de format, vous les transmettez dans la liste d'arguments. Par exemple :

```
w = 5
p = 3 s
= "abcdefg" printf
"<%*.*s>\n", w, p, s
```

est exactement équivalent à

```
s = "abcdefg" printf
"<%5.3s>\n", s
```

Les deux programmes affichent « <••abc> ». (Nous avons utilisé le symbole « • » pour représenter un espace, afin de bien montrer qu'il y a deux espaces dans le résultat.)

Les versions précédentes d'awk ne prenaient pas en charge cette fonctionnalité. Vous pouvez la simuler en utilisant concatenation pour construire la chaîne de format, comme ceci :

```
w = 5 p
= 3 s =
"abcdefg" printf "<%"
w "." p "s>\n", s
```

Cependant, ce n'est pas particulièrement facile à lire.

4.5.4 Exemples d'utilisation de printf

Voici comment utiliser printf pour créer un tableau aligné :

```
awk '{ printf "%-10s %s\n", $1, $2 }' Liste BBS
```

Affiche les noms des forums ($1) du fichier « BBS-list » sous forme d'une chaîne de 10 caractères, alignée à gauche. Affiche ensuite les numéros de téléphone ($2) sur la même ligne. On obtient ainsi un tableau à deux colonnes alignées, contenant les noms et les numéros de téléphone.

```
oryctérope 555-5553
alpo-net 555-3412
barfly 555-7685
morsures 555-1675
Camelot 555-0542
cœur 555-2912
pied de 555-1234
foire 555-6699
Macfoo 555-6480
Sdace 555-3430
excusez-moi 555-2127
```

Avez-vous remarqué que nous n'avons pas précisé que les numéros de téléphone devaient être imprimés en chiffres ? Il a fallu les imprimer sous forme de chaînes de caractères car les nombres sont séparés par un tiret. Ce tiret serait interprété comme un signe moins si nous avions essayé d'imprimer les numéros de téléphone sous forme de chiffres. ont abouti à des résultats assez déroutants.

Nous n'avons pas spécifié de largeur pour les numéros de téléphone car ce sont les derniers éléments affichés. lignes. Il n'est pas nécessaire de mettre d'espaces après.

Nous pourrions embellir encore davantage notre tableau en ajoutant des en-têtes de colonnes. À faire Pour cela, utilisez le modèle BEGIN (voir la section 6.7 [Modèles spéciaux BEGIN et END], page 53) pour forcer le En-tête à imprimer une seule fois, au début du programme awk :

```
awk 'BEGIN { print "Nom print "Nombre"
                   { printf "----" ------" }
        "%-10s %s\n", $1, $2 }' Liste BBS
```

Avez-vous remarqué que nous avons mélangé les instructions print et printf dans l'exemple ci-dessus ? Nous pourrions J'ai utilisé uniquement des instructions printf pour obtenir les mêmes résultats :

```
awk 'BEGIN { printf "%-10s %s\n", "Nom", "Numéro"
             printf "%-10s %s\n", "----", "------" }
     { printf "%-10s %s\n", $1, $2 }' Liste BBS
```

En affichant chaque en-tête de colonne avec la même spécification de format que celle utilisée pour les éléments de Dans la colonne, nous avons veillé à ce que les en-têtes soient alignés de la même manière que les colonnes.

Le fait que la même spécification de format soit utilisée trois fois peut être mis en évidence en la stockant. dans une variable, comme ceci :

```
awk 'BEGIN { format = "%-10s %s\n"
             format printf, "Nom", "Numéro"
             format printf, "----", "------" }
     Liste BBS { printf format, $1, $2 }
```

Essayez d'utiliser l'instruction printf pour aligner les en-têtes et les données du tableau. L'exemple « stock expédié » a été traité précédemment dans la section sur le relevé d'impression (voir section 4.1 [Le relevé d'impression], page 35).

4.6 Redirection de la sortie de print et printf

Jusqu'à présent, nous n'avons traité que des sorties affichées sur la sortie standard, généralement votre terminal. Les fonctions `print` et `printf` peuvent également rediriger leur sortie vers d'autres emplacements. On parle alors de redirection.

Une redirection apparaît après l'instruction `print` ou `printf`. Les redirections dans awk s'écrivent simplement `redirect`. comme les redirections dans les commandes shell, sauf qu'elles sont écrites à l'intérieur du programme awk.

4.6.1 Redirection de la sortie vers des fichiers et des tubes

Voici les trois formes de redirection de sortie. Elles sont toutes présentées pour l'instruction print, mais elles fonctionnent de la même manière pour printf.

L'instruction `print items > output-
file` redirige les éléments vers le fichier de sortie `output-file`. Ce fichier peut être nommé avec n'importe quelle expression. Sa valeur est convertie en chaîne de caractères, puis utilisée comme nom de fichier (voir le chapitre 8 [Expressions comme instructions d'action], page 57).

Lorsqu'on utilise ce type de redirection, le fichier de sortie est effacé avant l'écriture de la première donnée. Les écritures suivantes n'effacent pas le fichier de sortie, mais y ajoutent des données. Si le fichier de sortie n'existe pas, il est créé.

Par exemple, voici comment un programme awk peut écrire une liste de noms de BBS dans un fichier « name-list » et une liste de numéros de téléphone dans un fichier « phone-list ». Chaque fichier de sortie contient un nom ou un numéro par ligne.

```
La commande `awk '{ print $2 > "phone-list"
        print $1 > "name-list" }' BBS-list print items >> output-
```

file` redirige les éléments du fichier
de sortie vers le fichier `output-file`. Contrairement à la redirection simple `>`, le contenu existant du fichier `output-file` n'est pas effacé ; la sortie de `awk` est simplement ajoutée à la fin.

Afficher les éléments | commande
Il est également possible d'envoyer la sortie via un tube plutôt que dans un fichier. Ce type de redirection ouvre un tube vers la commande et écrit les valeurs des éléments via ce tube, vers un autre processus créé pour exécuter la commande.

La commande de redirection est en réalité une expression awk. Sa valeur est convertie en une chaîne de caractères dont le contenu indique la commande shell à exécuter.

Par exemple, ceci produit deux fichiers : une liste non triée de noms de BBS et une liste triée par ordre alphabétique inverse : awk '{ print $1 >

```
"names.unsorted"
        print $1 | "sort -r > names.sorted" }' BBS-list Ici, la liste non triée est écrite
```

avec une redirection ordinaire tandis que la liste triée est écrite en la faisant passer par l'utilitaire de tri.

Voici un exemple d'utilisation de la redirection pour envoyer un message à la liste de diffusion « bug-system ». Cela peut s'avérer utile en cas de problème lors de l'exécution périodique d'un script awk pour la maintenance du système.

```
rapport = "mail bug-system" print "Échec
du script Awk :", $0 | rapport print "à l'enregistrement numéro",
FNR, "de", FILENAME | rapport
```

```
fermer(rapport)
```

Nous utilisons la fonction `close` car il est recommandé de fermer le tube dès que toutes les données de sortie prévues y ont été envoyées. Pour plus d'informations, consultez la section 4.6.2 [Fermeture des fichiers et tubes de sortie], page 43. Cet exemple illustre également l'utilisation d'une variable pour représenter un fichier ou une commande : il n'est pas toujours nécessaire d'utiliser une constante de type chaîne de caractères. L'utilisation d'une variable est généralement conseillée, car `awk` exige que la valeur de la chaîne soit toujours identique.

La redirection de la sortie à l'aide de '>', '>>' ou '|' demande au système d'ouvrir un fichier ou un tube uniquement si le fichier ou la commande que vous avez spécifié n'a pas déjà été utilisé par votre programme, ou s'il a été fermé depuis sa dernière utilisation.

4.6.2 Fermeture des fichiers de sortie et des tubes

Lorsqu'un fichier ou un tube est ouvert, le nom du fichier ou la commande associée est mémorisé par awk, et les écritures suivantes dans ce même fichier ou avec cette même commande sont ajoutées aux précédentes. Le fichier ou le tube reste ouvert jusqu'à la fermeture d'awk. C'est généralement pratique.

Il peut parfois être nécessaire de fermer un fichier de sortie ou un tube de sortie plus tôt. Pour ce faire, utilisez la fonction close, comme suit :

```
fermer(nom_de_fichier)
```

ou

```
fermer(commande)
```

L'argument nom_de_fichier ou commande peut être n'importe quelle expression. Sa valeur doit être exactement égale à chaîne de caractères utilisée pour ouvrir le fichier ou le tube au départ — par exemple, si vous ouvrez un tube avec ceci :

```
imprimer $1 | "sort -r > names.sorted"
```

vous devez alors le terminer ainsi :

```
fermer("sort -r > noms.triés")
```

Voici quelques raisons pour lesquelles vous pourriez avoir besoin de fermer un fichier de sortie :

- Pour écrire un fichier et le relire ultérieurement dans le même programme awk. Fermez le fichier une fois l'écriture terminée ; vous pourrez ensuite le lire avec getline (voir la section 3.7 [Entrée explicite avec getline], page 30).
- Pour écrire plusieurs fichiers successivement dans le même programme awk. Si vous ne fermez pas les fichiers, vous risquez de dépasser la limite système du nombre de fichiers ouverts dans un même processus. Fermez donc chaque fichier une fois l'écriture terminée. • Pour terminer l'exécution

d'une commande. Lorsque vous redirigez la sortie via un tube (pipe), la commande qui lit ce tube continue généralement d'essayer de lire l'entrée tant que le tube est ouvert. Souvent, cela signifie que la commande ne peut pas s'exécuter correctement tant que le tube n'est pas fermé. Par exemple, si vous redirigez la sortie vers le programme mail, le message n'est envoyé que lorsque le tube est fermé.

- Exécuter le même programme une seconde fois, avec les mêmes arguments. Ce n'est pas la même chose.
  comme pour apporter davantage d'informations à la première exécution !

  Par exemple, supposons que vous redirigiez la sortie vers le programme de messagerie. Si vous envoyez plusieurs lignes redirigées vers ce tube sans le fermer, elles formeront un seul message de plusieurs lignes. En revanche, si vous fermez le tube après chaque ligne, chaque ligne constituera un message distinct.

La fonction close renvoie la valeur zéro si la fermeture a réussi. Sinon, la valeur sera différente de zéro.

4.7 Flux d'E/S standard

Les programmes en cours d'exécution disposent généralement de trois flux d'entrée et de sortie pour la lecture et l'écriture : l'entrée standard, la sortie standard et la sortie d'erreur standard. Par défaut, ces flux sont utilisés pour l'entrée et la sortie du terminal, mais ils sont souvent redirigés par l'interpréteur de commandes, via les opérateurs « < », « << », « > », « >> », « >& » et « | ». La sortie d'erreur standard sert uniquement à l'affichage des messages d'erreur ; la présence de deux flux distincts, la sortie standard et la sortie d'erreur standard, permet de les rediriger séparément.

Dans d'autres implémentations d'awk, la seule façon d'écrire un message d'erreur sur la sortie d'erreur standard est de… Un programme awk est le suivant :

```
print "Erreur grave détectée !\n" | "cat 1>&2"
```

Cela fonctionne en ouvrant un pipeline vers une commande shell qui peut accéder au flux d'erreur standard hérité du processus awk. Cette méthode est loin d'être élégante et s'avère également inefficace, car elle nécessite un processus distinct. C'est pourquoi les développeurs de programmes awk ont souvent négligé cette étape.
Au lieu de cela, ils ont envoyé les messages d'erreur au terminal, comme ceci :

```
NF != 4 {
      printf("ligne %d ignorée : ne contient pas 4 champs\n", FNR) > "/dev/tty"
}
```

Cela produit généralement le même effet, mais pas toujours : bien que le flux d'erreur standard soit habituellement le terminal, il peut être redirigé, et dans ce cas, l'écriture dans le terminal est incorrecte. En fait, si awk est exécuté par une tâche en arrière-plan, il se peut qu'aucun terminal ne soit accessible. L'ouverture de '/dev/tty' échouera alors.

5 phrases percutantes et efficaces

Les programmes awk utiles sont souvent courts, une ou deux lignes suffisent. Voici une sélection de programmes courts et pratiques pour vous initier. Certains contiennent des constructions qui n'ont pas encore été abordées. La description du programme vous donnera une bonne idée de son fonctionnement, mais lisez le reste du manuel pour maîtriser awk !

awk '{ si (NF > max) max = NF }
FIN { print max }' Ce
programme affiche le nombre maximal de champs sur n'importe quelle ligne d'entrée.

La commande `awk 'length($0)
> 80'` affiche toutes les lignes de plus de 80 caractères. Cette règle unique utilise une expression relationnelle comme modèle et ne comporte aucune action (l'action par défaut, l'affichage de l'enregistrement, est donc utilisée).

La commande
`awk 'NF > 0'` affiche chaque ligne contenant au moins un champ. C'est une méthode simple pour supprimer les lignes vides d'un fichier (ou plutôt, pour créer un nouveau fichier similaire à l'original, mais débarrassé des lignes vides).

awk '{ if (NF > 0) print }'
Ce programme imprime également chaque ligne contenant au moins un champ. Ici, la règle vérifie que chaque ligne correspond bien à une autre, puis l'action décide s'il faut imprimer ou non.

awk 'BEGIN { for (i = 1; i <= 7; i++) print int(101 *
rand()) }'
Ce programme imprime 7 nombres aléatoires compris entre 0 et 100.

ls -l fichiers | awk '{ x += $4 } ; END { print "total octets : " x }'
Ce programme affiche le nombre total d'octets utilisés par les fichiers.

développer le fichier | awk '{ si (x < longueur()) x = longueur() }
FIN { print "la longueur maximale de ligne est " x }'
Ce programme affiche la longueur maximale des lignes d'un fichier. Les données d'entrée sont traitées par le programme d'expansion afin de convertir les tabulations en espaces ; les largeurs comparées correspondent donc en réalité aux colonnes de la marge de droite.

awk 'BEGIN { FS = ":" } { print
$1 | "sort" }' /etc/passwd Ce programme imprime
une liste triée des noms de connexion de tous les utilisateurs.

awk '{ nlines++ }
FIN { print nlines }' Ce
programme compte les lignes d'un fichier.

awk 'END { print NR }' Ce
programme compte également les lignes d'un fichier, mais laisse awk faire le travail.

awk '{ print NR, $0 }' Ce
programme ajoute des numéros de ligne à tous ses fichiers d'entrée, similaire à 'cat -n'.

6 motifs

Dans awk, les modèles contrôlent l'exécution des règles : une règle est exécutée lorsque son modèle correspond au modèle. Enregistrement d'entrée actuel. Ce chapitre explique comment écrire des modèles.

6.1 Types de motifs

Voici un résumé des types de modèles pris en charge par awk.

/expression régulière/
: Une expression régulière utilisée comme modèle. Elle correspond à une valeur lorsque le texte de l'enregistrement d'entrée correspond à l'expression régulière. (Voir la section 6.2 [Expressions régulières utilisées comme modèles], page 47.)

expression
: Une expression unique. Elle correspond lorsque sa valeur, convertie en nombre, est différente de zéro (s'il s'agit d'un nombre) ou non nulle (s'il s'agit d'une chaîne de caractères). (Voir la section 6.5 [Expressions comme modèles], page 52.)

pat1, pat2
: Une paire de modèles séparés par une virgule, spécifiant une plage d'enregistrements. (Voir la section 6.6 [Spécification des plages d'enregistrements avec des modèles], page 53.)

COMMENCER
FIN
: Modèles spéciaux pour fournir des informations de démarrage ou de nettoyage à awk. (Voir la section 6.7 [Modèles spéciaux BEGIN et END], page 53.)

nul
: Le modèle vide correspond à chaque enregistrement d'entrée. (Voir la section 6.8 [Le modèle vide], page 54.)

6.2 Les expressions régulières comme modèles

Une expression régulière, ou regexp, permet de décrire une classe de chaînes de caractères. Une expression régulière encadrée par des barres obliques (« / ») est un motif awk qui correspond à chaque enregistrement d'entrée dont le texte appartient à cette classe.

L'expression régulière la plus simple est une séquence de lettres, de chiffres ou des deux. Une telle expression régulière correspond à toute chaîne de caractères contenant cette séquence. Ainsi, l'expression régulière « foo » correspond à toute chaîne contenant « foo ». Par conséquent, le modèle /foo/ correspond à tout enregistrement d'entrée contenant « foo ». D'autres types d'expressions régulières permettent de spécifier des classes de chaînes plus complexes.

6.2.1 Comment utiliser les expressions régulières

Une expression régulière peut servir de modèle en l'encadrant de barres obliques. Elle est ensuite comparée à l'intégralité du texte de chaque enregistrement. (En général, il suffit qu'une partie du texte corresponde pour que la comparaison réussisse.) Par exemple, le code suivant affiche le deuxième champ de chaque enregistrement contenant « foo » :

```
Liste BBS awk '/foo/ { print $2 }'
```

Les expressions régulières peuvent également être utilisées dans les expressions de comparaison. Vous pouvez alors spécifier la chaîne à comparer ; il n'est pas nécessaire qu'il s'agisse de l'intégralité de l'enregistrement d'entrée courant. Ces expressions de comparaison peuvent être utilisées comme modèles ou dans des instructions if, while, for et do.

Ceci ~ /regexp/ exp
est vrai si l'expression exp (prise comme une chaîne de caractères) correspond à regexp.
L'exemple suivant fait correspondre, ou sélectionne, tous les enregistrements d'entrée contenant la lettre majuscule « J » quelque part dans le premier champ :

```
awk '$1 ~ /J/' inventaire expédié
```

Ceci aussi :

```
awk '{ if ($1 ~ /J/) print }' inventaire expédié
```

exp !~ /regexp/
Ceci est vrai si l'expression exp (prise comme une chaîne de caractères) ne correspond pas à regexp. L'exemple suivant sélectionne tous les enregistrements d'entrée dont le premier champ ne contient pas la lettre majuscule « J » : awk '$1 !

```
~ /J/' inventory-shipped
```

Le côté droit d'un '~' L'opérateur '!~' n'est pas nécessairement une expression régulière constante (c'est-à-dire une chaîne de caractères entre barres obliques). Il peut s'agir de n'importe quelle expression. L'expression est évaluée et, si nécessaire, convertie en chaîne de caractères ; le contenu de cette chaîne est ensuite utilisé comme expression régulière. Une expression régulière calculée de cette manière est appelée expression régulière dynamique. Par exemple :

```
identifier_regexp = "[A-Za-z_][A-Za-z_0-9]+" $0 ~
identifier_regexp
```

définit identifier_regexp sur une expression régulière qui décrit les noms de variables awk, et vérifie si l'enregistrement d'entrée correspond à cette expression régulière.

6.2.2 Opérateurs d'expressions régulières

Vous pouvez combiner les expressions régulières avec les caractères suivants, appelés expressions régulières. opérateurs, ou métacaractères, pour accroître la puissance et la polyvalence des expressions régulières.

Voici un tableau des métacaractères. Tous les caractères non répertoriés dans le tableau sont considérés individuellement.

^ Cela correspond au début de la chaîne ou au début d'une ligne à l'intérieur de la chaîne.
Par exemple :

```
^@chapitre
```

correspond à '@chapter' en début de chaîne et peut être utilisé pour identifier les débuts de chapitres dans les fichiers sources Texinfo.

$ Ceci est similaire à '^', mais ne correspond qu'à la fin d'une chaîne ou à la fin d'une ligne au sein de cette chaîne. Par exemple : p$

correspond à un enregistrement se terminant par un 'p'.

. Cela correspond à n'importe quel caractère unique, sauf un saut de ligne. Par exemple :

.P

correspond à n'importe quel caractère unique suivi d'un « P » dans une chaîne. Grâce à la concaténation, nous pouvons Créez des expressions régulières comme 'U.A', qui correspond à toute séquence de trois caractères. commence par « U » et se termine par « A ».

[...] On appelle cela un jeu de caractères. Il correspond à n'importe lequel des caractères qui sont inclus. entre crochets. Par exemple :

[MVX]

correspond à l'un des caractères 'M', 'V' ou 'X' dans une chaîne.

Les plages de caractères sont indiquées par un trait d'union entre le début et la fin. caractères, et en encadrant le tout entre parenthèses. Par exemple :

[0-9]

correspond à n'importe quel chiffre.

Pour inclure le caractère '\', ']', '-', exemple : ou '^' Dans un jeu de caractères, placez un '\' devant.

[d\]]

correspond soit à 'd', soit à ']'.

Ce traitement du caractère '\' est compatible avec d'autres implémentations d'awk et est également imposé par la norme POSIX (Command Language and Utilities). Les expressions régulières d'awk constituent un sur-ensemble de la spécification POSIX pour les expressions régulières étendues. (ERE). Les ERE POSIX sont basés sur les expressions régulières acceptées par la norme traditionnelle Utilitaire egrep.

En syntaxe egrep, la barre oblique inverse n'a pas de caractère syntaxique particulier à l'intérieur des crochets. Cela signifie que des astuces spéciales doivent être utilisées pour représenter les caractères ']', '-' et '^' comme appartenant à un jeu de caractères.

En syntaxe egrep, pour faire correspondre '-', écrivez-le comme '---', ce qui est une plage contenant uniquement '-'. peut également être le premier ou le dernier caractère de l'ensemble. Pour correspondre à '^', placez-le n'importe où. sauf comme premier caractère d'un ensemble. Pour correspondre à un ']', placez-le en premier caractère de l'ensemble. ensemble. Par exemple :

[]d^]

correspond à ']', 'd' ou '^'.

[^ . . .] Il s'agit d'un jeu de caractères complémentaire. Le premier caractère après le '[' doit être un '^'. correspond à tous les caractères sauf ceux entre crochets (ou au saut de ligne). Par exemple :

[^0-9]

correspond à tout caractère qui n'est pas un chiffre.

| Il s'agit de l'opérateur d'alternance, utilisé pour spécifier des alternatives. Par exemple :

^P|[0-9]

correspond à toute chaîne de caractères correspondant à '^P' ou '[0-9]'. Cela signifie qu'elle correspond à n'importe quelle chaîne de caractères. chaîne de caractères contenant un chiffre ou commençant par « P ».

Cette alternance s'applique aux expressions régulières les plus grandes possibles de chaque côté.

(. . .) Les parenthèses servent à regrouper les éléments dans les expressions régulières, comme en arithmétique. Elles peuvent être utilisé pour concaténer des expressions régulières contenant l'opérateur d'alternance, '|'.

* Ce symbole signifie que l'expression régulière précédente doit être répétée autant de fois que nécessaire. autant de fois que possible pour trouver une correspondance. Par exemple :

ph*

applique le symbole '*' au 'h' précédent et recherche les correspondances avec un 'p' suivi de Accepte un nombre quelconque de « h ». Cela correspond également à la simple présence de « p » si aucun « h » n'est présent.

L'astérisque (*) répète la plus petite expression précédente possible. (Utilisez des parenthèses si vous souhaitez répéter une expression plus longue.) Il trouve autant de répétitions que possible. Par exemple :

```
L'exemple awk '/\(c[ad][ad]*rx\)/ { print }' imprime chaque
```

enregistrement de l'entrée contenant une chaîne de la forme '(car x)', '(cdr x)', '(cadr x)', etc.

+ Ce symbole est similaire à '*', mais l'expression précédente doit correspondre au moins une fois. Cela signifie que :

```
wh+y
```

correspondrait à « why » et « why » mais pas à « wy », tandis que « why*y » correspondrait à ces trois chaînes. Voici une façon plus simple d'écrire le dernier exemple avec « * » : awk '/\(c[ad]+rx\)/

```
{ print }'
```

? Ce symbole est similaire à '*', mais l'expression précédente peut correspondre une fois ou pas du tout. Par exemple :

```
fé?d
```

Correspondra à « fed » et « fd », mais à rien d'autre.

\ Ceci permet de masquer la signification particulière d'un caractère lors de la mise en correspondance. Par exemple :

```
$
```

correspond au caractère '$'.

Les séquences d'échappement utilisées pour les constantes de chaîne (voir la section 8.1 [Expressions constantes], page 57) sont également valides dans les expressions régulières ; elles sont également introduites par un '\'.

Dans les expressions régulières, les opérateurs « * », « + » et « ? » sont prioritaires, suivis de la concaténation, puis de l'opérateur « | ». Comme en arithmétique, les parenthèses peuvent modifier l'ordre de groupement des opérateurs.

6.2.3 Sensibilité à la casse dans l'appariement

La casse est généralement importante dans les expressions régulières, aussi bien pour la correspondance de caractères ordinaires (c'est-à-dire, à l'exclusion des métacaractères) qu'au sein d'un même jeu de caractères. Ainsi, un « w » dans une expression régulière correspond uniquement à un « w » minuscule et non à un « W » majuscule.

La méthode la plus simple pour effectuer une correspondance insensible à la casse consiste à utiliser un jeu de caractères : « [Ww] ». Cependant, cette solution peut s'avérer fastidieuse en cas d'utilisation fréquente et rendre les expressions régulières plus difficiles à lire. Deux autres alternatives pourraient vous convenir.

Une façon d'effectuer une correspondance insensible à la casse à un endroit précis du programme consiste à convertir les données en une seule casse, à l'aide des fonctions intégrées de manipulation de chaînes de caractères `tolower` ou `toupper` (que nous n'avons pas encore abordées ; voir la section 11.3 [Fonctions intégrées de manipulation de chaînes de caractères], page 90). Par exemple :

```
tolower($1) ~ /foo/ { . . . }
```

convertit le premier champ en minuscules avant de le comparer.

```
x = "aB" si (x
~ /ab/) . . . # ce test échouera
```

6.3 Expressions de comparaison en tant que modèles

Les modèles de comparaison testent des relations telles que l'égalité entre deux chaînes de caractères ou deux nombres. sont un cas particulier de modèles d'expressions (voir section 6.5 [Expressions en tant que modèles], page 52). sont écrites avec des opérateurs relationnels, qui constituent un sur-ensemble de ceux du C. Voici un tableau les répertoriant :

x < y	Vrai si x est inférieur à y.
x <= y	Vrai si x est inférieur ou égal à y.
x > y	Vrai si x est supérieur à y.
x ≥ y	Vrai si x est supérieur ou égal à y.
x == y	Vrai si x est égal à y.
x ≠ y	Vrai si x n'est pas égal à y.
x ~ et	Vrai si x correspond à l'expression régulière décrite par y.
x !~ y	Vrai si x ne correspond pas à l'expression régulière décrite par y.

Les opérandes d'un opérateur relationnel sont comparés comme des nombres s'ils sont tous deux des nombres. Sinon, elles sont converties en chaînes de caractères et comparées comme telles (voir section 8.9 [Conversion des chaînes de caractères]). (et les nombres], page 67, pour les règles détaillées). Les chaînes de caractères sont comparées en comparant la première lettre. on compare d'abord le premier caractère de chaque caractère, puis le second, et ainsi de suite, jusqu'à ce qu'il y ait une différence. Si le Deux cordes sont égales jusqu'à ce que la plus courte s'épuise ; la plus courte est alors considérée comme plus courte que la plus longue. La plus longue. Ainsi, « 10 » est inférieur à « 9 », et « abc » est inférieur à « abcd ».

L'opérande de gauche des opérateurs '~' et '!~' est une chaîne de caractères. L'opérande de droite est soit une constante, soit une constante. une expression régulière encadrée par des barres obliques (/regexp/), ou toute expression dont la valeur de chaîne est utilisée comme une expression régulière dynamique (voir la section 6.2.1 [Comment utiliser les expressions régulières], page 47).

L'exemple suivant affiche le deuxième champ de chaque enregistrement d'entrée dont le premier champ est précisément 'foo'.

```
awk '$1 == "foo" { print $2 }' Liste BBS
```

Comparez cela avec la correspondance d'expression régulière suivante, qui accepterait tout enregistrement contenant un premier champ contenant 'foo' :

```
awk '$1 ~ "foo" { print $2 }' Liste BBS
```

ou, de manière équivalente, celui-ci :

```
awk '$1 ~ /foo/ { print $2 }' Liste BBS
```

6.4 Opérateurs booléens et modèles

Un motif booléen est une expression qui combine d'autres motifs à l'aide des opérateurs booléens. « ou » (« || »), « et » (« && ») et « non » (« ! »). Indique si le modèle booléen correspond à un enregistrement d'entrée. Cela dépend de la correspondance de ses sous-modèles.

Par exemple, la commande suivante affiche tous les enregistrements du fichier d'entrée « BBS-list » qui contiennent à la fois '2400' et 'foo'.

```
Liste BBS awk '/2400/ && /foo/'
```

La commande suivante imprime tous les enregistrements du fichier d'entrée 'BBS-list' qui contiennent soit '2400', soit 'foo', soit les deux.

```
Liste de BBS awk '/2400/ || /foo/'
```

La commande suivante imprime tous les enregistrements du fichier d'entrée 'BBS-list' qui ne contiennent pas la chaîne 'foo'.

```
Liste BBS awk '! /foo/'
```

Notez que les motifs booléens constituent un cas particulier des motifs d'expression (voir la section 6.5 [Expressions comme motifs], page 52) ; ce sont des expressions qui utilisent les opérateurs booléens. Pour des informations complètes sur les opérateurs booléens, voir la section 8.6 [Expressions booléennes], page 64.

Les sous-motifs d'un motif booléen peuvent être des expressions régulières constantes, des comparaisons ou toute autre expression awk. Les motifs de plage ne sont pas des expressions et ne peuvent donc pas figurer à l'intérieur de motifs booléens. De même, les motifs spéciaux BEGIN et END, qui ne correspondent jamais à un enregistrement d'entrée, ne sont pas des expressions et ne peuvent donc pas figurer à l'intérieur de motifs booléens.

6.5 Les expressions comme modèles

Toute expression awk est également valide en tant que motif awk. Le motif « correspond » alors si l'expression…
La valeur de sion est non nulle (s'il s'agit d'un nombre) ou non nulle (s'il s'agit d'une chaîne de caractères).

L'expression est réévaluée à chaque fois que la règle est testée sur un nouvel enregistrement. Si l'expression utilise des champs comme $1, sa valeur dépend directement du texte du nouvel enregistrement ; sinon, elle dépend uniquement du déroulement de l'exécution du programme awk, ce qui peut néanmoins s'avérer utile.

Les modèles de comparaison constituent un cas particulier. Par exemple, l'expression `$5 == "foo"` vaut 1 lorsque `$5` vaut "foo" et 0 sinon ; par conséquent, cette expression, en tant que modèle, correspond lorsque les deux valeurs sont égales.

Les modèles booléens sont également des cas particuliers de modèles d'expression.

Une expression régulière constante utilisée comme motif est également un cas particulier de motif d'expression. L'expression /foo/ vaut 1 si « foo » apparaît dans l'enregistrement d'entrée actuel ; ainsi, en tant que motif, /foo/ correspond à tout enregistrement contenant « foo ».

D'autres implémentations d'awk qui ne sont pas encore conformes à posix sont moins générales que gawk : elles autorisent les expressions de comparaison et les combinaisons booléennes de celles-ci (éventuellement avec des parenthèses), mais pas nécessairement d'autres types d'expressions.

6.6 Spécification des plages d'enregistrements avec des modèles

Un motif de plage est composé de deux motifs séparés par une virgule, de la forme begpat, endpat.
Il correspond à des plages d'enregistrements d'entrée consécutifs. Le premier motif, begpat, détermine le début de la plage, et le second, endpat, détermine sa fin. Par exemple :

```
awk '$1 == "on", $1 == "off"'
```

Imprime tous les enregistrements entre les paires « on »/« off », inclus.

Un modèle de plage commence par comparer la valeur de début (begpat) à chaque enregistrement d'entrée ; lorsqu'un enregistrement correspond à begpat, le modèle de plage est activé. Le modèle de plage correspond alors à cet enregistrement. Tant que la fonction reste activée, elle compare automatiquement chaque enregistrement d'entrée lu. Elle compare également la position de fin (endpat) à chaque enregistrement d'entrée ; si cette comparaison réussit, la recherche par plage est désactivée pour l'enregistrement suivant. Elle reprend ensuite la vérification de la position de début (begpat) pour chaque enregistrement.

L'enregistrement qui active le modèle de plage et celui qui le désactive correspondent tous deux à ce modèle. Si vous ne souhaitez pas agir sur ces enregistrements, vous pouvez utiliser des instructions conditionnelles dans l'action de la règle pour les distinguer.

Il est possible qu'un motif soit activé et désactivé par le même disque, si les deux conditions sont réunies.
sont satisfaites par cet enregistrement. L'action est alors exécutée uniquement pour cet enregistrement.

6.7 Motifs spéciaux de début et de fin

BEGIN et END sont des modèles spéciaux. Ils ne servent pas à faire correspondre les enregistrements d'entrée, mais plutôt à fournir des informations de démarrage ou de nettoyage à votre script awk. Une règle BEGIN est exécutée une seule fois, avant la lecture du premier enregistrement d'entrée. Une règle END est exécutée une seule fois, après la lecture de toutes les données d'entrée. Par exemple :

```
awk 'BEGIN { print "Analyse de 'foo'" } /foo/ { ++foobar }

     FIN { print "'foo' apparaît " foobar " fois." }' Liste BBS
```

Ce programme compte le nombre d'enregistrements du fichier d'entrée « BBS-list » contenant la chaîne de caractères « foo ». La règle BEGIN affiche un titre pour le rapport. Il est inutile d'utiliser la règle BEGIN pour initialiser le compteur foobar à zéro, car awk le fait automatiquement (voir la section 8.2 [Variables], page 59).

La deuxième règle incrémente la variable foobar à chaque fois qu'un enregistrement contenant le motif 'foo'
La fonction est lue. La règle END affiche la valeur de foobar à la fin de l'exécution.

Les modèles spéciaux BEGIN et END ne peuvent pas être utilisés dans des plages ou avec des opérateurs booléens (en effet, (ils ne peuvent pas être utilisés avec n'importe quel opérateur).

Un programme awk peut comporter plusieurs règles BEGIN et/ou END. Elles sont exécutées dans l'ordre.
Elles apparaissent, toutes les règles BEGIN au démarrage et toutes les règles END à la fin.

L'utilisation de plusieurs sections BEGIN et END est utile pour écrire des fonctions de bibliothèque, car chaque bibliothèque peut avoir ses propres règles BEGIN ou END pour son initialisation et/ou son nettoyage. Notez que l'ordre dans lequel les fonctions de bibliothèque sont nommées sur la ligne de commande détermine l'ordre d'exécution de leurs règles BEGIN et END. Il est donc important d'écrire ces règles dans les fichiers de bibliothèque de manière à ce que leur ordre d'exécution soit indifférent. Pour plus d'informations sur l'utilisation des fonctions de bibliothèque, consultez le chapitre 14 [Invocation d'awk], page 105.

Si un programme awk ne contient qu'une règle BEGIN, et aucune autre, il se termine après l'exécution de cette règle. (Les anciennes versions d'awk continuaient à lire et à ignorer les entrées jusqu'à la fin du fichier.) Cependant, si une règle END est également présente, les entrées seront lues, même en l'absence d'autres règles. Ceci est nécessaire au cas où la règle END vérifierait la variable NR.

Les règles BEGIN et END doivent comporter des actions ; il n'existe pas d'action par défaut pour ces règles.
Aucun enregistrement actuel au moment de leur exécution.

6.8 Le motif vide

Un motif vide est considéré comme correspondant à chaque enregistrement d'entrée. Par exemple, le programme :

```
awk '{ print $1 }' Liste BBS
```

Imprime le premier champ de chaque enregistrement.

7 Aperçu des actions

Un programme ou script awk est constitué d'une série de règles et de définitions de fonctions, entremêlées. (Les fonctions sont décrites plus loin. Voir le chapitre 12 [Fonctions définies par l'utilisateur], page 95.)

Une règle contient un motif et une action, dont l'un ou l'autre peut être omis. L'action sert à indiquer à awk ce qu'il doit faire lorsqu'une correspondance avec le motif est trouvée. Le programme complet ressemble donc à ceci :

```
[modèle] [{ action }]
[modèle] [{ action }]
. . .
nom de la fonction (args) { . . . }
. . .
```

Une action est composée d'une ou plusieurs instructions awk, encadrées par des accolades (« { » et « } »). Chaque instruction spécifie une action à effectuer. Les instructions sont séparées par des sauts de ligne ou des points-virgules.

Les accolades autour d'une action sont obligatoires, même si l'action ne contient qu'une seule instruction, voire aucune. En revanche, si vous omettez complètement l'action, vous devez également omettre les accolades. (Une action omise équivaut à '{ print $0 }'.)

Voici les types d'instructions prises en charge par awk :

Les expressions, qui peuvent appeler des fonctions ou affecter des valeurs à des variables (voir le chapitre 8 [Expressions comme instructions d'action], page 57), permettent de calculer la valeur de l'expression puis de l'ignorer. Ceci est utile lorsque l'expression a des effets de bord (voir la section 8.7 [Expressions d'affectation], page 64).

• Les instructions de contrôle, qui définissent le flux d'exécution des programmes awk. Le langage awk propose des constructions similaires à celles du C (if, for, while, etc.) ainsi que quelques instructions spécifiques (voir le chapitre 9 [Instructions de contrôle dans les actions], page 73). • Les instructions composées, constituées d'une ou plusieurs instructions entre accolades. Une instruction composée permet de regrouper plusieurs instructions dans le corps d'une instruction if, while, do ou for. • Le contrôle des entrées, via la commande getline (voir la section 3.7 [Entrée explicite avec getline], page 30) et l'instruction next (voir la section 9.7 [L'instruction next], page 78). • Les instructions d'affichage, print et printf. Voir le chapitre 4 [Affichage de la sortie], page 35. • Les instructions de suppression, pour supprimer des éléments d'un tableau. Voir la section 10.6 [L'instruction delete], page 85.

Les deux chapitres suivants traitent en détail des expressions et des instructions de contrôle. Nous aborderons ensuite les tableaux et les fonctions intégrées, tous deux utilisés dans les expressions. Enfin, nous verrons comment définir vos propres fonctions.

8 Expressions comme énoncés d'action

Les expressions sont les éléments de base des actions awk. Une expression est évaluée et produit une valeur que vous pouvez afficher, tester, stocker dans une variable ou passer à une fonction. De plus, une expression peut affecter une nouvelle valeur à une variable ou à un champ grâce à un opérateur d'affectation.

Une expression peut constituer une instruction à elle seule. La plupart des autres types d'instructions contiennent une ou plusieurs expressions qui spécifient les données à manipuler. Comme dans d'autres langages, les expressions en awk incluent les variables, les références de tableaux, les constantes et les appels de fonctions, ainsi que leurs combinaisons avec divers opérateurs.

8.1 Expressions constantes

Le type d'expression le plus simple est la constante, qui a toujours la même valeur. Il existe trois types de constantes : les constantes numériques, les constantes de chaîne et les constantes d'expression régulière.

Une constante numérique représente un nombre. Ce nombre peut être un entier, une fraction décimale ou un nombre en notation scientifique (exponentielle). Notez que toutes les valeurs numériques sont représentées dans awk en virgule flottante double précision. Voici quelques exemples de constantes numériques, qui ont toutes la même valeur :

```
105
1,05e+2
1050e-1
```

Une constante de chaîne est constituée d'une séquence de caractères placée entre guillemets doubles. Par exemple :

```
"perroquet"
```

représente la chaîne de caractères contenant « perroquet ». Les chaînes de caractères dans gawk peuvent avoir n'importe quelle longueur et contenir tous les caractères ASCII 8 bits possibles, y compris le caractère nul. D'autres implémentations d'awk peuvent rencontrer des difficultés avec certains codes de caractères.

Certains caractères ne peuvent pas être inclus littéralement dans une constante de chaîne. Vous les représentez à la place. avec des séquences d'échappement, qui sont des séquences de caractères commençant par une barre oblique inverse (« \ »).

L'une des utilisations d'une séquence d'échappement est d'inclure un guillemet double dans une constante de chaîne. Puisqu'un guillemet double simple terminerait la chaîne, il faut utiliser '\"' pour représenter un seul guillemet double dans la chaîne. Le caractère barre oblique inverse lui-même ne peut pas être inclus normalement ; on écrit '\\' pour insérer une barre oblique inverse dans la chaîne. Ainsi, la chaîne contenant les deux caractères '"\' doit être écrite "\"\\".

L'antislash sert également à représenter des caractères non imprimables, comme le saut de ligne. Bien qu'il soit possible d'écrire directement la plupart de ces caractères dans une constante de chaîne, le résultat peut s'avérer inesthétique.

Voici un tableau de toutes les séquences d'échappement utilisées dans awk :

\\	Représente une barre oblique inverse littérale, '\'.
\un	Représente le caractère « alerte », contrôle-g, code ASCII 7.
b	Représente la touche de retour arrière, Ctrl+H, code ASCII 8.
\f	Représente un saut de page, contrôle-l, code ASCII 12.
\n	Représente un saut de ligne, Ctrl+J, code ASCII 10.
\r	Représente un retour chariot, contrôle-m, code ASCII 13.
\t	Représente une tabulation horizontale, contrôle-i, code ASCII 9.
\v	Représente une tabulation verticale, contrôle-k, code ASCII 11.
\nnn	Représente la valeur octale nnn, où nnn représente un à trois chiffres compris entre 0 et 7. Par exemple, le code du caractère ASCII ESC (échappement) est '\033'.
\xhh. . .	Représente la valeur hexadécimale hh, où les hh sont des chiffres hexadécimaux (de '0' à '9'). '9' et soit 'A' à 'F', soit 'a' à 'f'). Comme la même construction en C ANSI, La séquence d'échappement se poursuit jusqu'à l'apparition du premier chiffre non hexadécimal. Cependant, L'utilisation de plus de deux chiffres hexadécimaux produit des résultats indéfinis. (L'échappement '\x' (La séquence n'est pas autorisée dans POSIX awk.)

Une expression régulière constante est une description d'expression régulière encadrée par des barres obliques, telle que /^début et end$/. La plupart des expressions régulières utilisées dans les programmes awk sont constantes, mais les opérateurs '~' et '!~' peuvent correspondent également aux expressions régulières calculées ou « dynamiques » (voir la section 6.2.1 [Comment utiliser les expressions régulières], page 47).

Les expressions régulières constantes peuvent être utilisées comme des expressions simples. Lorsqu'une expression régulière constante ne figure pas sur le Le côté droit des opérateurs '~' ou '!~' a la même signification que s'il apparaissait dans un motif. c'est-à-dire '($0 ~ /foo/)' (voir section 6.5 [Expressions comme modèles], page 52). Cela signifie que les deux segments de code,

```
si ($0 ~ /barfly/ || $0 ~ /camelot/)
        imprimer « trouvé »
```

et

```
si (/barfly/ || /camelot/)
        imprimer « trouvé »
```

sont exactement équivalentes. Une conséquence assez bizarre de cette règle est que la valeur booléenne suivante L'expression est légale, mais ne produit pas l'effet escompté par l'utilisateur :

```
si (/foo/ ~ $1) afficher "foo trouvé"
```

Ce code teste « manifestement » si $1 correspond à l'expression régulière /foo/. Mais en réalité, L'expression (/foo/ ~ $1) signifie en réalité (($0 ~ /foo/) ~ $1). Autrement dit, il faut d'abord faire correspondre le L'enregistrement d'entrée est comparé à l'expression régulière /foo/. Le résultat sera soit 0, soit 1, selon le cas. Succès ou échec du match. Comparez ensuite ce résultat avec le premier champ enregistré.

Une autre conséquence de cette règle est que l'instruction d'affectation

```
correspondances = /foo/
```

attribuera la valeur 0 ou 1 à la variable matches, en fonction du contenu de l'enregistrement d'entrée actuel.

Les expressions régulières constantes sont également utilisées comme premier argument des fonctions sub et gsub. (voir la section 11.3 [Fonctions intégrées pour la manipulation de chaînes], page 90).

Cette caractéristique du langage n'a jamais été bien documentée avant la spécification POSIX.

Vous vous demandez peut-être quand aura lieu cet événement.

```
$1 ~ /foo/ { . . . }
```

préférable à

```
$1 ~ "foo" { . . . }
```

Comme les opérandes des deux opérateurs '~' sont constants, il est plus efficace d'utiliser la forme '/foo/' : awk peut ainsi identifier l'expression régulière fournie et la stocker en interne sous une forme qui optimise la correspondance de motifs. Avec la seconde forme, awk doit d'abord convertir la chaîne dans ce format interne, puis effectuer la correspondance. La première forme est également plus élégante ; elle indique clairement qu'il s'agit d'une correspondance avec une expression régulière.

8.2 Variables

Les variables permettent de nommer des valeurs et d'y faire référence ultérieurement. Vous avez déjà rencontré des variables dans de nombreux exemples. Le nom d'une variable doit être une suite de lettres, de chiffres et de tirets bas, mais ne peut pas commencer par un chiffre. La casse est importante : « a » et « A » sont des variables distinctes.

Le nom d'une variable constitue une expression valide en soi ; il représente la valeur actuelle de la variable. Les variables reçoivent de nouvelles valeurs grâce aux opérateurs d'affectation et d'incrémentation. Voir la section 8.7 [Expressions d'affectation], page 64.

Certaines variables ont des significations intégrées spécifiques, comme FS, le séparateur de champs, et NF, le nombre de champs dans l'enregistrement d'entrée courant. Consultez le chapitre 13 [Variables intégrées], page 101, pour en obtenir la liste. Ces variables intégrées peuvent être utilisées et affectées comme toutes les autres variables, mais leurs valeurs sont également utilisées ou modifiées automatiquement par awk. Le nom de chaque variable intégrée est composé exclusivement de lettres majuscules.

Dans awk, les variables peuvent se voir attribuer des valeurs numériques ou des chaînes de caractères. Par défaut, elles sont initialisées à la chaîne vide, qui équivaut à zéro. Il n'est pas nécessaire d'initialiser explicitement chaque variable dans awk, contrairement à ce qui se fait en C ou dans la plupart des autres langages traditionnels.

8.2.1 Affectation de variables en ligne de commande

Vous pouvez définir n'importe quelle variable awk en incluant une affectation de variable parmi les arguments de la ligne de commande lors de l'appel à awk (voir le chapitre 14 [Appel à awk], page 105). Une telle affectation a la forme suivante :

```
variable=text
```

Grâce à cela, vous pouvez définir une variable soit au début de l'exécution d'awk, soit entre les fichiers d'entrée.

Si vous faites précéder l'affectation de l'option '-v', comme ceci :

```
-v variable=text
```

La variable est alors initialisée dès le début, avant même l'exécution des instructions BEGIN. L'option « -v » et son affectation doivent précéder tous les arguments de nom de fichier, ainsi que le texte du programme.

Sinon, l'affectation de la variable est effectuée à un moment déterminé par sa position parmi les Arguments du fichier d'entrée : après le traitement de l'argument du fichier d'entrée précédent. Par exemple :

```
awk '{ print $n }' n=4 inventaire expédié n=2 liste BBS
```

Affiche la valeur du champ numéro n pour tous les enregistrements d'entrée. Avant la lecture du premier fichier, la ligne de commande initialise la variable n à 4. Le quatrième champ est alors affiché dans les lignes du fichier « inventory-shipped ». Une fois le premier fichier traité, mais avant le début du second, n est réinitialisé à 2, afin que le deuxième champ soit affiché dans les lignes du fichier « BBS-list ».

Les arguments de la ligne de commande sont mis à disposition pour un examen explicite par le programme awk dans un tableau nommé ARGV (voir chapitre 13 [Variables intégrées], page 101).

awk traite les valeurs des affectations de ligne de commande pour les séquences d'échappement (voir section 8.1). [Expressions constantes], page 57).

8.3 Opérateurs arithmétiques

Le langage awk utilise les opérateurs arithmétiques courants lors de l'évaluation des expressions. Ces opérateurs respectent les règles de priorité habituelles et fonctionnent comme prévu.
Cet exemple divise le champ trois par le champ quatre, additionne le champ deux, stocke le résultat dans le champ un et imprime l'enregistrement d'entrée modifié résultant :

```
awk '{ $1 = $2 + $3 / $4; print }' inventaire expédié
```

Les opérateurs arithmétiques en awk sont :

x + y	Ajout.
x - y	Soustraction.
- x	Négation.

+ x	Unaire plus. Aucun effet réel sur l'expression.
x * y	Multiplication.
x / y	Division. Étant donné que tous les nombres dans awk sont des nombres à virgule flottante double précision, le résultat n'est pas arrondi à l'entier le plus proche : 3 / 4 a pour valeur 0,75.
x % y	Reste. Le quotient est arrondi à l'entier le plus proche de zéro, multiplié par y et Ce résultat est soustrait de x. Cette opération est parfois appelée « troncature modulo ». La relation suivante est toujours vérifiée : b * int(a / b) + (a % b) == a Un effet potentiellement indésirable de cette définition du reste est que x % y est négatif. si x est négatif. Ainsi, -17 % 8 = -1 Dans d'autres implémentations d'awk, le signe du reste peut dépendre de la machine.
x ^ et x ** y	Exponentiation : x élevé à la puissance y. 2^3 vaut 8. La séquence de caractères '**' est équivalente à '^'. (La norme POSIX spécifie uniquement l'utilisation de '^' pour exponentiation.)

8.4 Concaténation de chaînes

Il n'existe qu'une seule opération sur les chaînes de caractères : la concaténation. Elle ne possède pas d'opérateur spécifique pour… le représenter. La concaténation s'effectue plutôt en écrivant les expressions les unes à côté des autres, avec aucun opérateur. Par exemple :

```
awk '{ print "Champ numéro un : " $1 }' Liste BBS
```

produit, pour le premier disque de la « liste BBS » :

```
Champ numéro un : oryctérope
```

Sans l'espace après le caractère « : » dans la constante de chaîne, la ligne serait continue. Par exemple :

```
awk '{ print "Champ numéro un :" $1 }' Liste BBS
```

produit, pour le premier disque de la « liste BBS » :

```
Champ numéro un : oryctérope
```

Comme la concaténation de chaînes ne possède pas d'opérateur explicite, il est souvent nécessaire de s'assurer que Cela se produit là où vous le souhaitez en encadrant les éléments à concaténer entre parenthèses.
Par exemple, le fragment de code suivant ne concatène pas le fichier et le nom comme on pourrait s'y attendre :

```
fichier = "fichier"
nom = "nom"
imprimer « quelque chose de significatif » > nom de fichier
```

Il est nécessaire d'utiliser les éléments suivants :

imprimer « quelque chose de significatif » > (nom de fichier)

Nous vous recommandons d'utiliser des parenthèses autour de la concaténation dans tous les contextes, sauf les plus courants. (comme dans l'opérande de droite de '=').

8.5 Expressions de comparaison

Les expressions de comparaison comparent des chaînes de caractères ou des nombres pour établir des relations telles que l'égalité. sont écrites à l'aide d'opérateurs relationnels, qui constituent un sur-ensemble de ceux du C. Voici un tableau les répertoriant :

x < y	Vrai si x est inférieur à y.
x <= y	Vrai si x est inférieur ou égal à y.
x > y	Vrai si x est supérieur à y.
x ≥ y	Vrai si x est supérieur ou égal à y.
x == y	Vrai si x est égal à y.
x ≠ y	Vrai si x n'est pas égal à y.
x ~ et	Vrai si la chaîne x correspond à l'expression régulière désignée par y.
x !~ y	Vrai si la chaîne x ne correspond pas à l'expression régulière désignée par y.
indice dans le tableau	Vrai si le tableau contient un élément avec l'indice spécifié.

Les expressions de comparaison ont la valeur 1 si elles sont vraies et 0 si elles sont fausses.

Les règles utilisées par Gawk pour effectuer les comparaisons sont basées sur celles du brouillon 11.2 du POSIX. La norme POSIX a introduit le concept de chaîne numérique, qui est simplement une chaîne de caractères. cela ressemble à un nombre, par exemple, « +2 ».

Lors de l'exécution d'une opération relationnelle, gawk considère que le type d'un opérande est le type qu'il reçu lors de sa dernière affectation, plutôt que le type de sa dernière utilisation (voir section 8.10 [Numérique et [Valeurs de chaîne], page 68). Ce type est inconnu lorsque l'opérande provient d'une source « externe » : champ variables, arguments de ligne de commande, éléments de tableau résultant d'une opération de division et la valeur d'un élément ENVIRON. Dans ce cas uniquement, si l'opérande est une chaîne numérique, alors il est considéré comme être à la fois de type chaîne et de type numérique. Si au moins un opérande d'une comparaison est de type chaîne Ensuite seulement, une comparaison de chaînes de caractères est effectuée. Tout opérande numérique sera converti en chaîne de caractères. en utilisant la valeur de CONVFMT (voir section 8.9 [Conversion de chaînes et de nombres], page 67). L'un des opérandes d'une comparaison est numérique, et l'autre opérande est soit numérique, soit les deux sont numériques. et une chaîne de caractères, puis awk effectue une comparaison numérique. Si les deux opérandes sont des deux types, alors le La comparaison est numérique. Les chaînes de caractères sont comparées en comparant le premier caractère de chacune, puis le reste. deuxième caractère de chaque, et ainsi de suite. Ainsi, « 10 » est inférieur à « 9 ». S'il existe deux chaînes où L'un est un préfixe de l'autre, la chaîne la plus courte est inférieure à la plus longue. Ainsi, « abc » est inférieur à "abcd".

Voici quelques exemples d'expressions, la manière dont awk les compare et le résultat de la comparaison.

1,5 <= 2,0

comparaison numérique (vrai)

Comparaison de
chaînes « abc » >= « xyz » (faux)

1,5 ≠ " +2"
comparaison de chaînes (vrai)

"1e2" < "3"
comparaison de chaînes (vrai)

a = 2 ; b = "2" a ==
b
comparaison de chaînes (vrai)

```
echo 1e2 3 | awk '{ print ($1 < $2) ? "true" : "false" }'
```

affiche « false » puisque $1 et $2 sont tous deux des chaînes numériques et ont donc à la fois le type chaîne et le type numérique, ce qui impose une comparaison numérique.

L'objectif des règles de comparaison et de l'utilisation de chaînes numériques est de tenter de produire le comportement « le moins surprenant », tout en « faisant ce qu'il faut ».

Les comparaisons de chaînes de caractères et les comparaisons d'expressions régulières sont très différentes. Par exemple,

```
$1 == "foo"
```

prend la valeur 1, ou est vrai, si le premier champ de l'enregistrement d'entrée actuel est précisément « foo ». En revanche,

```
1 $ ~ /foo/
```

prend la valeur 1 si le premier champ contient 'foo', comme 'foobar'.

L'opérande de droite des opérateurs '~' et '!~' peut être soit une expression régulière constante (/. . ./), soit une expression ordinaire, auquel cas la valeur de l'expression sous forme de chaîne est une expression régulière dynamique (voir la section 6.2.1 [Comment utiliser les expressions régulières], page 47).

Dans les implémentations très récentes d'awk, une expression régulière constante entre barres obliques est également une expression. L'expression régulière /regexp/ est une abréviation de cette expression de comparaison :

```
$0 ~ /regexp/
```

Dans certains contextes, il peut être nécessaire d'encadrer l'expression régulière de parenthèses pour éviter toute confusion avec l'analyseur syntaxique awk. Par exemple, `(/x/ - /y/) > threshold` n'est pas autorisé, mais `((/x/) - (/y/)) > threshold` est correctement interprété.

Un cas particulier où /foo/ n'est pas une abréviation de $0 ~ /foo/ est lorsqu'il s'agit de l'opérande droit ou de '!~'! Voir la section 8.1 [Expressions constantes], page 57, où cela est discuté plus en détail dans l'opérande de '~'.

8.6 Expressions booléennes

Une expression booléenne est une combinaison d'expressions de comparaison ou d'appariement, utilisant les opérateurs booléens « ou » (« || »), « et » (« && ») et « non » (« ! »), ainsi que des parenthèses pour contrôler l'imbrication. La valeur de vérité de l'expression booléenne est calculée en combinant les valeurs de vérité des expressions qui la composent.

Les expressions booléennes peuvent être utilisées partout où des expressions de comparaison et de correspondance peuvent être utilisées. Elles peuvent être utilisées dans les instructions if, while, do et for. Elles ont des valeurs numériques (1 si vrai, 0 si faux), qui interviennent si le résultat de l'expression booléenne est stocké dans une variable ou utilisé dans des opérations arithmétiques.

De plus, toute expression booléenne est également un motif booléen valide, vous pouvez donc l'utiliser comme un modèle permettant de contrôler l'exécution des règles.

Vous trouverez ci-dessous la description des trois opérateurs booléens, accompagnée d'un exemple pour chacun. Il peut être instructif de comparer ces exemples avec les exemples analogues de motifs booléens (voir la section 6.4 [Opérateurs booléens et motifs], page 51), qui utilisent les mêmes opérateurs booléens dans des motifs plutôt que dans des expressions.

booléen1 && booléen2
: Vrai si boolean1 et boolean2 sont tous deux vrais. Par exemple, l'instruction suivante affiche l'enregistrement d'entrée actuel s'il contient à la fois '2400' et 'foo'. if ($0 ~ /2400/ && $0 ~ /foo/) print

  La sous-expression boolean2 n'est évaluée que si boolean1 est vrai. Cela peut avoir une incidence lorsque boolean2 contient des expressions ayant des effets de bord : dans le cas de $0 ~ /foo/ && ($2 == bar++), la variable bar n'est pas incrémentée s'il n'y a pas de 'foo' dans l'enregistrement.

boolean1 || boolean2 Renvoie
: vrai si au moins l'un des booleans 1 ou 2 est vrai. Par exemple, la commande suivante affiche tous les enregistrements du fichier d'entrée « BBS-list » contenant soit « 2400 », soit « foo », soit les deux.

  La commande `awk '{ if ($0 ~ /2400/ || $0 ~ /foo/) print }'` (liste BBS) indique que la sous-expression `boolean2` n'est évaluée que si `boolean1` est faux. Cela peut avoir une incidence lorsque `boolean2` contient des expressions ayant des effets de bord.

!boolean Renvoie vrai si la valeur booléenne est fausse. Par exemple, le programme suivant affiche tous les enregistrements du fichier d'entrée « BBS-list » qui ne contiennent pas la chaîne « foo ».

```
awk '{ if (! ($0 ~ /foo/)) print }' Liste BBS
```

8.7 Expressions d'affectation

Une affectation est une expression qui stocke une nouvelle valeur dans une variable. Par exemple, affectons `nil` à une variable. la valeur 1 pour la variable z :

```
z = 1
```

Une fois cette expression exécutée, la variable z prend la valeur 1. L'ancienne valeur de z avant cette affectation est oubliée.

Les affectations peuvent également stocker des chaînes de caractères. Par exemple, le code suivant stockerait la valeur « this food is good » dans la variable message :

```
chose = "nourriture"
prédicat = "bon" message =
"cette " chose " est " prédicat
```

(Ceci illustre également la concaténation de chaînes de caractères.)

Le signe « = » est appelé opérateur d'affectation. C'est l'opérateur d'affectation le plus simple car la valeur de l'opérande de droite est conservée sans modification.

La plupart des opérateurs (addition, concaténation, etc.) n'ont d'autre effet que de calculer une valeur. Si l'on ignore cette valeur, l'opérateur est inutile. L'opérateur d'affectation est différent : il produit une valeur, mais même si on l'ignore, l'affectation a un effet de bord, puisqu'elle modifie la variable. On parle alors d'effet de bord.

L'opérande de gauche d'une affectation n'est pas nécessairement une variable (voir la section 8.2 [Variables], page 59) ; il peut également s'agir d'un champ (voir la section 3.4 [Modification du contenu d'un champ], page 24) ou d'un élément de tableau (voir le chapitre 10 [Tableaux dans awk], page 81). Ces éléments sont appelés lvalues, ce qui signifie qu'ils peuvent figurer à gauche d'un opérateur d'affectation. L'opérande de droite peut être n'importe quelle expression ; il produit la nouvelle valeur que l'affectation stocke dans la variable, le champ ou l'élément de tableau spécifié.

Il est important de noter que les variables n'ont pas de type permanent. Le type d'une variable correspond simplement au type de la valeur qu'elle contient à un instant donné. Dans l'extrait de programme suivant, la variable foo a d'abord une valeur numérique, puis une valeur de type chaîne de caractères :

```
foo = 1
imprimer foo
foo = "bar"
imprimer foo
```

Lorsque la deuxième affectation attribue à foo une valeur de type chaîne de caractères, le fait qu'elle ait précédemment eu une valeur numérique est oublié.

Une affectation est une expression, elle a donc une valeur : la même valeur qui lui est affectée. Ainsi, l'expression z = 1 a la valeur 1. Il en résulte que l'on peut écrire plusieurs affectations à la suite.

```
x = y = z = 0
```

La fonction stocke la valeur 0 dans les trois variables. Cela s'explique par le fait que la valeur de z = 0, qui est 0, est stockée dans y, puis la valeur de y = z = 0, qui est 0, est stockée dans x.

On peut utiliser une affectation partout où une expression est requise. Par exemple, il est correct d'écrire `x != (y = 1)` pour affecter la valeur 1 à `y` puis vérifier si `x` est égal à 1. Cependant, ce style a tendance à rendre les programmes difficiles à lire ; sauf dans le cas d'un programme ponctuel, il est préférable de le réécrire pour éviter ces imbrications d'affectations. Ce n'est jamais très compliqué.

Outre l'opérateur « = », plusieurs autres opérateurs d'affectation permettent d'effectuer des opérations arithmétiques sur l'ancienne valeur de la variable. Par exemple, l'opérateur « += » calcule une nouvelle valeur en ajoutant la valeur de droite à l'ancienne valeur de la variable. Ainsi, l'affectation suivante ajoute 5 à la valeur de foo :

```
foo += 5
```

Cela équivaut précisément à ce qui suit :

```
foo = foo + 5
```

Utilisez celle qui rend le sens de votre programme le plus clair.

Voici un tableau des opérateurs d'affectation arithmétique. Dans chaque cas, l'opérande de droite est une expression dont la valeur est convertie en un nombre.

lvalue += incrément
: Ajoute un incrément à la valeur de lvalue pour obtenir la nouvelle valeur de lvalue.

lvalue -= décrémentation
: Soustrait le décrément de la valeur de lvalue.

lvalue *= coefficient
: Multiplie la valeur de lvalue par le coefficient.

lvalue /= quotient Divise
: la valeur de lvalue par le quotient.

lvalue %= module
: Définit la valeur de lvalue à son reste par le modulo.

lvalue ^= puissance lvalue

**= puissance Élève lvalue à
: la puissance puissance. (Seul l'opérateur ^= est spécifié par posix.)

8.8 Opérateurs d'incrémentation

Les opérateurs d'incrémentation augmentent ou diminuent la valeur d'une variable de 1. On pourrait faire la même chose avec un opérateur d'affectation ; les opérateurs d'incrémentation n'ajoutent donc aucune puissance au langage awk ; mais ce sont des abréviations pratiques pour quelque chose de très courant.

L'opérateur d'ajout de 1 s'écrit '++'. Il peut être utilisé pour incrémenter une variable avant ou après son écriture. après en avoir pris la valeur.

Pour pré-incrémenter une variable v, écrivez ++v. Cela ajoute 1 à la valeur de v et cette nouvelle valeur est La valeur de cette expression est également valable. L'expression d'affectation v += 1 est parfaitement équivalente.

L'ajout du symbole « ++ » après la variable permet d'effectuer une post-incrémentation. La valeur de la variable est incrémentée de la même manière ; la différence réside dans le fait que la valeur de l'expression d'incrémentation correspond à l'ancienne valeur de la variable. Ainsi, si `foo` vaut 4, l'expression `foo++` vaut également 4, mais elle modifie la valeur de `foo` à 5.

L'incrémentation de `foo++` est presque équivalente à l'écriture `(foo += 1) - 1`. Ce n'est pas parfaitement équivalent car tous les nombres utilisés dans awk sont des nombres à virgule flottante : en virgule flottante, `foo + 1 - 1` n'est pas nécessairement égal à `foo`. Mais la différence est minime tant que l'on utilise des nombres relativement petits (inférieurs à un billion).

Toute valeur lvalue peut être incrémentée. Les champs et les éléments de tableau sont incrémentés de la même manière que les variables. (Utilisez '$(i++)' lorsque vous souhaitez effectuer une référence de champ et une incrémentation de variable en même temps. Les parenthèses sont nécessaires en raison de la priorité de l'opérateur de référence de champ, '$'.)

L'opérateur de décrémentation '--' fonctionne comme '++', sauf qu'il soustrait 1 au lieu d'ajouter. Tout comme '++', il peut être utilisé avant la lvalue pour pré-décrémenter ou après pour post-décrémenter.

Voici un résumé des expressions d'incrémentation et de décrémentation.

++lvalue Cette expression incrémente lvalue et la nouvelle valeur devient la valeur de cette expression.
sion.

lvalue++ Cette expression incrémente la valeur de lvalue. La valeur renvoyée est l'ancienne valeur de lvalue.

--lvalue Comme ++lvalue, mais au lieu d'additionner, il soustrait. Il décrémente lvalue et renvoie
la valeur qui en résulte.

lvalue-- Comme lvalue++, mais au lieu d'additionner, il soustrait. Il décrémente lvalue. La valeur de
l'expression correspond à l'ancienne valeur de lvalue.

8.9 Conversion de chaînes de caractères et de nombres

Les chaînes de caractères sont converties en nombres, et les nombres en chaînes, si le contexte du programme awk l'exige. Par exemple, si la valeur de `foo` ou de `bar` dans l'expression `foo + bar` est une chaîne de caractères, elle est convertie en nombre avant l'addition. Si des valeurs numériques apparaissent dans la concaténation de chaînes, elles sont également converties en chaînes. Prenons un exemple :

```
deux = 2 ; trois = 3 afficher
(deux trois) + 4
```

Cela affiche finalement la valeur (numérique) 27. Les valeurs numériques des variables deux et trois sont converties en chaînes de caractères et concaténées, et la chaîne résultante est reconvertie en nombre 23, auquel 4 est ensuite ajouté.

Si, pour une raison quelconque, vous devez forcer la conversion d'un nombre en chaîne de caractères, concaténez le caractère nul. Une chaîne de caractères contenant ce nombre. Pour forcer la conversion d'une chaîne en nombre, ajoutez zéro à cette chaîne.

Une chaîne de caractères est convertie en nombre en interprétant son préfixe numérique comme un chiffre : « 2.5 » devient 2.5, « 1e3 » devient 1000 et « 25fix » a la valeur numérique de 25. Les chaînes qui ne peuvent pas être interprétées comme des nombres valides sont converties en zéro.

La méthode exacte de conversion des nombres en chaînes de caractères est contrôlée par la variable intégrée CONVFMT d'awk (voir chapitre 13 [Variables intégrées], page 101). Cette conversion s'effectue à l'aide d'une version spéciale de la fonction sprintf (voir chapitre 11 [Fonctions intégrées], page 89) avec CONVFMT comme spécificateur de format.

La valeur par défaut de CONVFMT est « %.6g », ce qui permet d'afficher une valeur avec au moins six chiffres significatifs. Pour certaines applications, il est nécessaire de la modifier afin d'obtenir une précision accrue. La double précision, sur la plupart des ordinateurs modernes, offre une précision de 16 ou 17 chiffres décimaux.

Des résultats inattendus peuvent se produire si vous définissez CONVFMT sur une chaîne de caractères qui n'indique pas à sprintf comment formater correctement les nombres à virgule flottante. Par exemple, si vous oubliez le symbole « % » dans le format, tous les nombres seront convertis en une seule et même chaîne de caractères.

Dans le cas particulier où un nombre est un entier, le résultat de sa conversion en chaîne de caractères est toujours un entier, quelle que soit la valeur de CONVFMT. Prenons l'exemple du fragment de code suivant :

```
CONVFMT = "%2.2f" a =
12
b = a ""
```

b a la valeur « 12 », et non « 12.00 ».

Avant la norme POSIX, awk spécifiait que la valeur de OFMT était utilisée pour convertir les nombres en chaînes de caractères. OFMT spécifie le format de sortie à utiliser lors de l'affichage de nombres avec la commande print. CONVFMT a été introduit afin de dissocier la sémantique des conversions de celle de l'affichage.
Les fonctions CONVFMT et OFMT ont toutes deux la même valeur par défaut : « %.6g ». Dans la grande majorité des cas, les anciens programmes awk ne modifieront pas leur comportement. Toutefois, il est important de tenir compte de cette utilisation d'OFMT si vous devez porter votre programme vers d'autres implémentations d'awk ; nous vous recommandons plutôt que de modifier vos programmes, de porter directement gawk !

8.10 Valeurs numériques et chaînes de caractères

Dans la majeure partie de ce manuel, les valeurs awk (telles que les constantes, les champs ou les variables) sont présentées sous forme de nombres ou de chaînes de caractères. Cette approche est pratique car, généralement, elles ne sont utilisées que d'une seule manière.

En réalité, les valeurs awk peuvent être à la fois des chaînes de caractères et des nombres. En interne, awk représente les valeurs par une chaîne de caractères, un nombre (à virgule flottante) et une indication précisant si l'une, l'autre ou les deux représentations de la valeur sont valides.

Il est important de conserver une trace des deux types de valeurs pour optimiser l'exécution : une variable peut prendre la valeur d'une chaîne de caractères lors de sa première utilisation, et cette valeur peut ensuite être utilisée jusqu'à ce qu'une nouvelle valeur lui soit attribuée. Ainsi, si une variable ne contenant qu'une valeur numérique est utilisée dans plusieurs concaténations successives, il suffit de lui attribuer une représentation sous forme de chaîne une seule fois. La valeur numérique reste valide, de sorte qu'aucune conversion en nombre n'est nécessaire si la variable est utilisée ultérieurement dans une expression arithmétique.

Le suivi de ces deux types de valeurs est également important pour des calculs numériques précis. Prenons l'exemple suivant :

```
a = 123,321
CONVFMT = "%3,1f" b = a
"
est un nombre"
c = a + 1,654
```

Lors de la concaténation et de l'affectation à la variable b, celle-ci reçoit une chaîne de caractères : « 123,3 ». Si la valeur numérique de a est perdue lors de sa conversion en chaîne, l'utilisation numérique de a dans la dernière instruction sera erronée. La variable c se verra alors attribuer la valeur 124,954 au lieu de 124,975. De telles erreurs s'accumulent rapidement et affectent gravement les calculs numériques.

Une fois qu'une valeur numérique se voit attribuer une valeur de chaîne correspondante, cette dernière reste valide jusqu'à une nouvelle attribution. Si CONVFMT (voir section 8.9 [Conversion de chaînes et de nombres], page 67) est modifié entre-temps, l'ancienne valeur de chaîne sera conservée. Par exemple :

```
DÉBUT
    { CONVFMT = "%2.2f" a = 123.456
    b = a ""                    # Forcer 'a' à avoir également une valeur de chaîne
    printf "a = %s\n", a CONVFMT =
    "%.6g" printf "a = %s\n", aa
    += 0 printf "a = %s\n", a
                                # Rendre 'a' uniquement numérique # Utiliser
                                'a' comme chaîne de caractères
}
```

Ce programme affiche « a = 123,46 » deux fois, puis « a = 123,456 ».

Voir la section 8.9 [Conversion des chaînes de caractères et des nombres], page 67, pour les règles qui spécifient comment Les valeurs de type chaîne de caractères sont composées de valeurs numériques.

8.11 Expressions conditionnelles

Une expression conditionnelle est un type particulier d'expression à trois opérandes. Elle vous permet de Utiliser la valeur d'une expression pour sélectionner l'une des deux autres expressions.

L'expression conditionnelle est identique à celle du langage C :

```
sélecteur ? si-exp-vrai : si-exp-faux
```

Il y a trois sous-expressions. La première, le sélecteur, est toujours calculée en premier. Si elle est vraie (ni nulle ni égale à zéro), alors l'expression « si vraie » est calculée ensuite et sa valeur devient celle de l'expression entière. Sinon, c'est l'expression « si fausse » qui est calculée ensuite et sa valeur devient celle de l'expression entière.

Par exemple, cette expression donne la valeur absolue de x :

```
x > 0 ? x : -x
```

À chaque calcul de l'expression conditionnelle, seul le résultat de `if-true-exp` ou `if-false-exp` est calculé ; l'autre est ignoré. Ceci est important lorsque les expressions ont des effets de bord. Par exemple, cette expression conditionnelle examine l'élément `i` du tableau `a` ou du tableau `b`, et incrémente `i`.

```
x == y ? a[i++] : b[i++]
```

Cela garantit que i sera incrémenté exactement une fois, car à chaque fois, l'une ou l'autre des deux expressions d'incrémentation est exécutée, et l'autre ne l'est pas.

8.12 Appels de fonction

Une fonction désigne un calcul particulier. Grâce à son nom, on peut l'appeler par son nom à n'importe quel moment du programme. Par exemple, la fonction sqrt calcule la racine carrée d'un nombre.

Un ensemble fixe de fonctions est intégré, ce qui signifie qu'elles sont disponibles dans tous les programmes awk. La fonction sqrt en fait partie. Consultez le chapitre 11 [Fonctions intégrées], page 89, pour obtenir la liste des fonctions intégrées et leur description. De plus, vous pouvez définir vos propres fonctions dans le programme pour les utiliser ailleurs dans ce même programme. Consultez le chapitre 12 [Fonctions définies par l'utilisateur], page 95, pour savoir comment procéder.
Faites ceci.

Pour utiliser une fonction, il faut utiliser une expression d'appel de fonction, composée du nom de la fonction suivi d'une liste d'arguments entre parenthèses. Les arguments sont des expressions qui fournissent les données nécessaires au calcul effectué par la fonction. S'il y a plusieurs arguments, ils sont séparés par des virgules. S'il n'y a pas d'arguments, il suffit d'écrire « () » après le nom de la fonction.
Voici quelques exemples :

```
sqrt(x^2 + y^2) atan2(y,    # Un argument
x) rand()                   # Deux arguments
                            # Aucun argument
```

N'insérez aucun espace entre le nom de la fonction et la parenthèse ouvrante ! Le nom d'une fonction définie par l'utilisateur ressemble au nom d'une variable ; un espace donnerait l'impression que l'expression est la concaténation d'une variable et d'une expression entre parenthèses. L'espace avant la parenthèse est sans conséquence pour les fonctions intégrées, mais il est préférable de ne pas prendre l'habitude d'en utiliser pour éviter les erreurs avec les fonctions définies par l'utilisateur.

Chaque fonction attend un nombre précis d'arguments. Par exemple, la fonction sqrt doit…
être appelée avec un seul argument, le nombre dont on veut calculer la racine carrée :

```
racine carrée(argument)
```

Certaines fonctions intégrées permettent d'omettre le dernier argument. Dans ce cas, une valeur par défaut appropriée est utilisée. Pour plus de détails, consultez le chapitre 11 [Fonctions intégrées], page 89. Si des arguments sont omis lors d'appels à des fonctions définies par l'utilisateur, ils sont traités comme des variables locales, initialisées à la chaîne vide (voir le chapitre 12 [Fonctions définies par l'utilisateur], page 95).

Comme toute autre expression, l'appel de fonction a une valeur, calculée par la fonction en fonction des arguments qui lui sont fournis. Dans cet exemple, la valeur de `sqrt(argument)` est la racine carrée de l'argument. Une fonction peut également avoir des effets de bord, comme l'affectation de valeurs à certaines variables ou la réalisation d'opérations d'entrée/sortie.

Voici une commande permettant de lire des nombres, un nombre par ligne, et d'afficher la racine carrée de chacun.
un:

```
awk '{ print "La racine carrée de", $1, "est", sqrt($1) }'
```

8.13 Priorité des opérateurs (Comment les opérateurs s'imbriquent)

La priorité des opérateurs détermine leur regroupement lorsqu'ils apparaissent côte à côte dans une même expression. Par exemple, l'opérateur « * » est prioritaire sur l'opérateur « + » ; ainsi, a + b * c signifie multiplier b et c, puis ajouter a au produit (c'est-à-dire a + (b * c)).

Vous pouvez modifier la priorité des opérateurs en utilisant des parenthèses. Ces règles de priorité indiquent où les parenthèses sont implicites si vous ne les écrivez pas. En fait, il est judicieux d'utiliser systématiquement des parenthèses lorsqu'une combinaison d'opérateurs est inhabituelle, car les autres personnes qui liront le programme pourraient oublier la priorité dans ce cas. Vous pourriez vous aussi l'oublier et commettre une erreur. L'utilisation explicite de parenthèses permet d'éviter ce type d'erreur.

Lorsque des opérateurs de même priorité sont utilisés ensemble, l'opérateur le plus à gauche est groupé en premier, à l'exception des opérateurs d'affectation, de condition et d'exponentiation, qui sont groupés dans l'ordre inverse.
Ainsi, a - b + c se regroupent comme (a - b) + c ; a = b = c se regroupent comme a = (b = c).

La priorité des opérateurs unaires préfixes est sans importance tant que seuls des opérateurs unaires sont utilisés, car il n'y a qu'une seule façon de les analyser : en commençant par les plus internes. Ainsi, `$++i` signifie `$(++i)` et `++$x` signifie `++($x)`. Cependant, lorsqu'un autre opérateur suit l'opérande, la priorité des opérateurs unaires peut avoir une incidence. Ainsi, `$x^2` signifie `($x)^2`, mais `-x^2` signifie `-(x^2)`, car `-` a une priorité inférieure à `^`, tandis que `$` a une priorité supérieure.

Voici un tableau des opérateurs d'awk, par ordre de priorité croissante :

Les opérateurs
: d'affectation '=', '+=', '-=', '*=', '/=', '%=', '^=', '**=' sont utilisés. Ces opérateurs s'appliquent de droite à gauche. (L'opérateur '**=' n'est pas spécifié par POSIX.)

conditionnel
: '?:'. Cet opérateur groupe les caractères de droite à gauche.

Opérateur logique
: « ou ». '||'.

« et » logique.
: « && ».

appartenance au tableau
: 'dans'.

correspondance '~', '!~'.

Les opérateurs relationnels et les
: redirections ont le même niveau de priorité. Les caractères tels que « > » peuvent servir à la fois d'opérateurs relationnels et de redirections ; le contexte permet de distinguer les deux significations.

  Les opérateurs relationnels sont '<', '<=', '==', '!=', '>=' et '>'.

  Les opérateurs de redirection d'E/S sont '<', '>', '>>' et '|'.

  Notez que les opérateurs de redirection d'E/S dans les instructions print et printf appartiennent au niveau de l'instruction, et non aux expressions. La redirection ne produit pas d'expression.

qui pourrait être l'opérande d'un autre opérateur. Par conséquent, il est incorrect d'utiliser un opérateur de redirection à proximité d'un autre opérateur de priorité inférieure, sans parenthèses. De telles combinaisons, par exemple « print foo > a ? b : c », entraînent des erreurs de syntaxe.

enchaînement
: Aucun symbole particulier n'est utilisé pour indiquer la concaténation. Les opérandes sont simplement écrits côte à côte.

ajouter,
soustraire '+', '-'.

multiplier, diviser, modulo
'*', '/', '%'.

plus, moins, « non » '+', '-', '!'.

L'exponentiation est
représentée par les opérateurs '^' et '**'. Ces opérateurs sont groupés de droite à gauche. (L'opérateur '**' n'est pas spécifié par POSIX.)

incrémenter, décrémenter
'++', '--'.

champ '$'.

9 instructions de contrôle dans les actions

Les instructions de contrôle telles que if, while, etc. contrôlent le flux d'exécution des programmes awk.
La plupart des instructions de contrôle dans awk sont calquées sur des instructions similaires en C.

Toutes les instructions de contrôle commencent par des mots-clés spéciaux tels que if et while, afin de les distinguer.
les transformer en expressions simples.

De nombreuses instructions de contrôle contiennent d'autres instructions ; par exemple, l'instruction « if » contient une autre instruction qui peut être exécutée ou non. L'instruction contenue est appelée le corps du dispositif.
Si vous souhaitez inclure plusieurs instructions dans le corps du texte, regroupez-les en une seule instruction composée à l'aide d'accolades, en les séparant par des sauts de ligne ou des points-virgules.

9.1 L'instruction if

L'instruction if-else est l'instruction de prise de décision d'awk. Elle ressemble à ceci :

```
si (condition) alors-corps [sinon sinon-corps]
```

La condition est une expression qui détermine le comportement du reste de l'instruction. Si la condition est vraie, le corps de la clause « then » est exécuté ; sinon, c'est le corps de la clause « else » qui est exécuté (si la clause « else » est présente).
La partie « else » de l'instruction est facultative. La condition est considérée comme fausse si sa valeur est zéro ou la chaîne vide, et vraie sinon.

Voici un exemple :

```
si (x % 2 == 0) afficher "x
        est pair" sinon afficher "x est
impair"
```

Dans cet exemple, si l'expression x % 2 == 0 est vraie (c'est-à-dire que la valeur de x est divisible par 2),
alors la première instruction d'impression est exécutée, sinon c'est la deuxième instruction d'impression qui est exécutée.

Si la clause else apparaît sur la même ligne que le corps de la clause then, et que ce dernier n'est pas une instruction composée (c'est-à-dire non entourée d'accolades), un point-virgule doit séparer le corps de la clause else. Pour illustrer cela, réécrivons l'exemple précédent :

```
awk '{ if (x % 2 == 0) print "x est pair"; else print "x est impair" }'
```

Si vous oubliez le point-virgule (;), awk ne pourra pas analyser l'instruction et vous obtiendrez une erreur de syntaxe.

Nous n'écririons pas cet exemple de cette façon, car un lecteur humain pourrait ne pas le voir.
sinon, si ce n'était pas la première chose sur sa ligne.

9.2 L'instruction while

En programmation, une boucle désigne une partie d'un programme qui est (ou du moins peut être) exécutée deux fois ou plus. plusieurs fois de suite.

L'instruction `while` est l'instruction de boucle la plus simple en awk. Elle exécute une instruction de manière répétée tant qu'une condition est vraie. Elle ressemble à ceci :

```
tandis que (condition)
    corps
```

Ici, « body » est une instruction que nous appelons le corps de la boucle, et « condition » est une expression qui contrôle la durée d'exécution de la boucle.

La première action de la boucle `while` est de tester une condition. Si la condition est vraie, le corps de la boucle est exécuté. (La condition est vraie si la valeur est différente de zéro et n'est pas une chaîne vide.) Une fois le corps de la boucle exécuté, la condition est testée à nouveau ; si elle est toujours vraie, le corps de la boucle est exécuté une seconde fois. Ce processus se répète jusqu'à ce que la condition devienne fausse. Si la condition est initialement fausse, le corps de la boucle n'est jamais exécuté.

Cet exemple affiche les trois premiers champs de chaque enregistrement, un par ligne.

```
awk '{ i = 1
        tant que (i <= 3) { afficher $i i+
                +

        }
}'
```

Ici, le corps de la boucle est une instruction composée encadrée par des accolades, contenant deux instructions.

La boucle fonctionne ainsi : la valeur de i est initialement initialisée à 1. Ensuite, la boucle `while` vérifie si i est inférieur ou égal à 3. C'est le cas lorsque i vaut 1, donc le i-ème champ est affiché. Puis, `i++` incrémente la valeur de i et la boucle se répète. La boucle s'arrête lorsque i atteint 4.

Comme vous pouvez le constater, un saut de ligne n'est pas obligatoire entre la condition et le corps du programme ; toutefois, son utilisation améliore la lisibilité, sauf si le corps du programme est une instruction composée ou très simple. Le saut de ligne après l'accolade ouvrante qui introduit l'instruction composée n'est pas non plus obligatoire, mais le programme serait difficile à lire sans lui.

9.3 L'instruction do-while

La boucle do est une variante de la boucle while. La boucle do exécute son corps une seule fois.
Le corps du programme est ensuite répété tant que la condition est vraie. Voici à quoi cela ressemble :

```
faire le
corps pendant (condition)
```

Même si la condition est fausse au départ, le corps de la fonction est exécuté au moins une fois (et une seule fois, sauf si L'exécution du corps de la boucle rend la condition vraie. Comparez cela avec l'instruction while correspondante :

```
tandis que (condition)
    corps
```

Cette instruction n'exécute pas son corps, même une seule fois, si la condition est fausse dès le départ.

Voici un exemple d'instruction do :

```
awk '{ i = 1 do { print
                $0 i++ }
                    while (i <= 10)

}'
```

Affiche chaque enregistrement d'entrée dix fois. Ce n'est pas un exemple très réaliste, car dans ce cas, une simple boucle `while` suffirait. Mais cela reflète la pratique : l'utilisation d'une instruction `do` est rarement vraiment utile.

9.4 Déclaration « pour »

L'instruction for facilite le comptage des itérations d'une boucle. La forme générale de La boucle for ressemble à ceci :

```
pour (initialisation; condition; incrément)
    corps
```

Cette instruction commence par exécuter une initialisation. Ensuite, tant que la condition est vraie, elle exécute successivement le corps du programme, puis l'incrémentation. Généralement, l'initialisation attribue la valeur zéro ou un à une variable, l'incrémentation lui ajoute 1, et la condition compare le résultat au nombre d'itérations souhaité.

Voici un exemple d'instruction for :

```
awk '{ for (i = 1; i <= 3; i++) print $i

}'
```

Ceci imprime les trois premiers champs de chaque enregistrement d'entrée, un champ par ligne.

Dans une boucle `for`, le corps représente n'importe quelle instruction, tandis que l'initialisation, la condition et l'incrémentation sont de simples expressions. Il est impossible d'affecter plusieurs variables à l'initialisation, sauf en utilisant une instruction d'affectation multiple telle que `x = y = 0`, ce qui n'est possible que si toutes les valeurs initiales sont égales. (Vous pouvez cependant initialiser des variables supplémentaires en écrivant leurs affectations dans des instructions séparées avant la boucle `for`.)

Il en va de même pour l'incrémentation : pour incrémenter d'autres variables, il faut ajouter des instructions distinctes à la fin de la boucle. L'expression composée C, utilisant l'opérateur virgule, serait utile dans ce contexte, mais elle n'est pas prise en charge par awk.

Le plus souvent, l'incrément est une expression d'incrémentation, comme dans l'exemple ci-dessus. Mais ce n'est pas obligatoire ; il peut s'agir de n'importe quelle expression. Par exemple, cette instruction affiche toutes les puissances de 2 comprises entre 1 et 100 :

```
pour (i = 1 ; i <= 100 ; i *= 2) afficher i
```

Chacune des trois expressions entre parenthèses suivant la boucle « for » peut être omise si elle est vide. Ainsi, « for (;x > 0;) » est équivalent à « while (x > 0) ». Si la condition est omise, elle est considérée comme vraie, ce qui crée une boucle infinie (c'est-à-dire une boucle qui ne se termine jamais).

Dans la plupart des cas, une boucle for est l'abréviation d'une boucle while, comme illustré ici :

```
initialisation tant
que (condition) { incrémentation
    du
    corps
}
```

La seule exception est lorsque l'instruction continue (voir la section 9.6 [L'instruction continue], page 77) est utilisée à l'intérieur de la boucle ; le remplacement d'une instruction for par une instruction while de cette manière peut modifier l'effet de l'instruction continue à l'intérieur de la boucle.

Il existe une autre version de la boucle for, permettant de parcourir tous les indices d'un tableau :

```
pour (i dans le tableau)
        faire quelque chose avec tableau[i]
```

Voir le chapitre 10 [Tableaux dans awk], page 81, pour plus d'informations sur cette version de la boucle for.

Le langage awk propose une instruction `for` en plus d'une instruction `while`, car la boucle `for` est souvent plus concise et plus intuitive. Compter le nombre d'itérations est une opération courante dans les boucles. Il est plus simple de considérer ce comptage comme faisant partie intégrante de la boucle plutôt que comme une opération à effectuer à l'intérieur de celle-ci.

La section suivante présente des exemples plus complexes de boucles for.

9.5 Déclaration de rupture

L'instruction break sort de la boucle for, while ou do-while la plus interne qui la contient.
L'exemple suivant trouve le plus petit diviseur de n'importe quel entier et identifie également les nombres premiers :

```
awk '# trouver le plus petit diviseur de num
```

```
{ num = 1 $
    pour (div = 2; div*div <= num; div++)
        si (num % div == 0)
            casser
    if (num % div == 0) printf "Le plus
        petit diviseur de %d est %d\n", num, div else printf "%d est premier\n", num }'
```

Lorsque le reste de la première condition `if` est nul, `awk` sort immédiatement de la boucle `for` qui la contient. Autrement dit, `awk` passe directement à l'instruction suivant la boucle et poursuit son exécution. (Ce comportement est très différent de l'instruction `exit` qui arrête complètement le programme `awk`. Voir la section 9.8 [L'instruction `exit`], page 79.)

Voici un autre programme équivalent au précédent. Il illustre comment la condition d'un
On pourrait tout aussi bien remplacer « for » ou « while » par une pause à l'intérieur d'un « if » :

```
awk '# trouver le plus petit diviseur de num
        { num = $1
            pour (div = 2; ; div++) { si (num % div
                == 0) {
                    printf "Le plus petit diviseur de %d est %d\n", num, div break

                }
                si (div*div > num) { printf "%d
                    est premier\n", num
                    casser
                }
            }
}'
```

9.6 Déclaration de poursuite

L'instruction `continue`, tout comme `break`, s'utilise uniquement à l'intérieur des boucles `for`, `while` et `do-while`. Elle ignore le reste du corps de la boucle, ce qui provoque le démarrage immédiat de l'itération suivante. À l'inverse, la fonction `break` sort complètement de la boucle. Voici un exemple :

```
# Afficher les noms qui ne contiennent pas la chaîne « ignore »

# Tout d'abord, enregistrez le texte de chaque ligne
{ names[NR] = $0 }

# Imprimer ce qui nous intéresse
FIN
    { pour (x dans noms) { si
            (noms[x] ~ /ignorer/)
                continuer
            imprimer les noms[x]
     }
}
```

Si l'un des enregistrements d'entrée contient la chaîne « ignorer », cet exemple ignore l'instruction d'impression pour cet enregistrement et revient à la première instruction de la boucle.

Ce n'est pas un exemple pratique de continue, car il serait tout aussi simple d'écrire la boucle comme ceci :

```
pour (x dans noms) si
    (noms[x] !~ /ignorer/) afficher noms[x]
```

L'instruction `continue` dans une boucle `for` indique à awk de passer outre le reste du corps de la boucle et de reprendre l'exécution à l'étape d'incrémentation de la boucle `for`. Le programme suivant illustre ce comportement :

```
awk 'BEGIN { for
        (x = 0; x <= 20; x++) { if (x == 5)

                    continuer
            printf ("%d ", x)
        }
                ""
        imprimer
}'
```

Ce programme affiche tous les nombres de 0 à 20, sauf 5, pour lequel la fonction printf est ignorée.
Comme l'incrémentation x++ n'est pas ignorée, x ne reste pas bloqué à 5. Comparez la boucle for ci-dessus avec la boucle while :

```
awk 'DÉBUT { x = 0 tant
        que (x <=
        20) { si (x == 5) continue printf ("%d
            ", x)


            x++
        }
                ""
        imprimer
}'
```

Ce programme boucle indéfiniment une fois que x atteint 5.

Comme décrit ci-dessus, l'instruction continue n'a aucun sens lorsqu'elle est utilisée en dehors du corps d'une boucle.

9.7 La déclaration suivante

L'instruction suivante force awk à interrompre immédiatement le traitement de l'enregistrement courant et à passer au suivant. Cela signifie qu'aucune autre règle n'est exécutée pour l'enregistrement courant. Le reste de l'action de la règle en cours n'est pas non plus exécuté.

Comparez cela à l'effet de la fonction `getline` (voir section 3.7 [Entrée explicite avec `getline`], page 30). Celle-ci aussi provoque la lecture immédiate de l'enregistrement suivant par `awk`, mais sans modifier le flux d'exécution. Ainsi, le reste de l'action en cours s'exécute avec un nouvel enregistrement d'entrée.

Au plus haut niveau, l'exécution d'un programme awk se présente comme une boucle qui lit un enregistrement d'entrée, puis teste le modèle de chaque règle par rapport à cet enregistrement. Si l'on considère cette boucle comme une instruction `for` dont le corps contient les règles, alors l'instruction suivante est analogue à une instruction `continue` : elle passe à la fin du corps de cette boucle implicite et exécute l'incrémentation (qui consiste à lire un autre enregistrement).

Par exemple, si votre programme awk ne fonctionne que sur les enregistrements comportant quatre champs, et que vous ne le souhaitez pas Pour que le programme échoue en cas d'entrée incorrecte, vous pouvez utiliser cette règle au début du programme :

```
NF != 4
    { printf("ligne %d ignorée : ne contient pas 4 champs", FNR) > "/dev/stderr"
    suivant
}
```

Ainsi, les règles suivantes ne pourront pas traiter l'enregistrement erroné. Le message d'erreur est redirigé vers le flux de sortie d'erreur standard, comme il se doit. Voir la section 4.7 [Flux d'E/S standard], page 44.

Selon la norme POSIX, le comportement est indéfini si l'instruction suivante est utilisée dans une règle BEGIN ou END. gawk la traitera comme une erreur de syntaxe.

Si l'instruction suivante entraîne la fin de la saisie, alors le code des règles END,
Le cas échéant, elles seront exécutées. Voir la section 6.7 [Modèles spéciaux BEGIN et END], page 53.

9.8 Déclaration de sortie

L'instruction de sortie provoque l'arrêt immédiat de l'exécution de la règle en cours par awk et son arrêt.
Traitement des données saisies ; toute autre donnée saisie est ignorée.

Si une instruction exit est exécutée à partir d'une règle BEGIN, le programme cesse immédiatement tout traitement. Aucun enregistrement d'entrée n'est lu. Cependant, si une règle END est présente, elle est exécutée (voir la section 6.7 [Modèles spéciaux BEGIN et END], page 53).

Si la commande exit est utilisée dans une règle END, elle provoque l'arrêt immédiat du programme.

Une instruction `exit` faisant partie d'une règle ordinaire (c'est-à-dire, n'appartenant pas à une règle `BEGIN` ou `END`) interrompt l'exécution de toutes les règles automatiques suivantes, mais la règle `END` est exécutée si elle existe. Si vous ne souhaitez pas que la règle `END` s'applique dans ce cas, vous pouvez affecter une valeur différente de zéro à une variable avant l'instruction `exit`, puis vérifier cette variable dans la règle `END`.

Si un argument est fourni à la commande exit, sa valeur est utilisée comme code d'état de sortie pour le processus awk.
Si aucun argument n'est fourni, la fonction exit renvoie le statut zéro (succès).

Par exemple, imaginons que vous ayez découvert une erreur que vous ne savez pas vraiment comment gérer.
Par convention, les programmes signalent cette erreur en se terminant avec un code de sortie différent de zéro. Votre programme awk peut le faire en utilisant une instruction exit avec un argument différent de zéro. Voici un exemple :

```
DÉBUT { si
        (("date" | getline date_now) < 0) {
            print "Impossible d'obtenir la date système" > "/dev/stderr"
            sortie 4
        }
}
```

10 tableaux en awk

Un tableau est une table de valeurs, appelées éléments. Les éléments d'un tableau se distinguent par leurs indices. Les indices peuvent être des nombres ou des chaînes de caractères. Chaque tableau possède un nom, qui ressemble à un nom de variable, mais ne doit pas être utilisé comme nom de variable dans le même programme awk.

10.1 Introduction aux tableaux

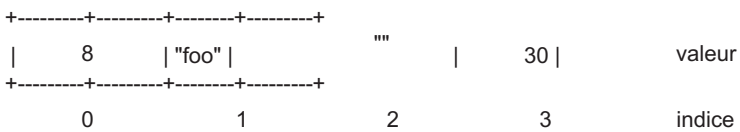
Le langage awk utilise des tableaux unidimensionnels pour stocker des groupes de chaînes de caractères ou de nombres apparentés.

Chaque tableau awk doit avoir un nom. Les noms de tableaux ont la même syntaxe que les noms de variables ; n'importe quel nom Un nom de variable valide serait également un nom de tableau valide. Mais vous ne pouvez pas utiliser le même nom des deux manières. (sous forme de tableau et de variable) dans un seul programme awk.

Les tableaux dans awk ressemblent superficiellement aux tableaux d'autres langages de programmation ; mais il existe Différences fondamentales. Avec awk, il n'est pas nécessaire de spécifier la taille d'un tableau avant de commencer à… Utilisez-le. De plus, n'importe quel nombre ou chaîne de caractères dans awk peut être utilisé comme index de tableau.

Dans la plupart des autres langages, il faut déclarer un tableau et spécifier le nombre d'éléments ou composants qu'il contient. Dans de tels langages, la déclaration provoque la mise en œuvre d'un bloc de mémoire contigu. être alloué pour autant d'éléments. Un index dans le tableau doit être un entier positif ; par exemple, L'indice 0 spécifie le premier élément du tableau, qui est en fait stocké au début du tableau. bloc de mémoire. L'indice 1 spécifie le deuxième élément, qui est stocké en mémoire juste après le premier élément, et ainsi de suite. Il est impossible d'ajouter d'autres éléments au tableau, car il n'y a plus d'espace disponible. pour autant d'éléments que vous l'avez déclaré.

Conceptuellement, un tableau contigu de quatre éléments pourrait ressembler à ceci si les valeurs des éléments sont 8, "foo", "" et 30 :



Seules les valeurs sont stockées ; les indices sont implicites et déduits de l'ordre des valeurs. La valeur à l'indice 8 est celle de l'indice 8. L'indice est 0, car le chiffre 8 apparaît à la position qui ne comporte aucun élément avant lui.

Les tableaux dans awk sont différents : ils sont associatifs. Cela signifie que chaque tableau est une collection de paires : un index et la valeur correspondante de l'élément du tableau :

Élément 4	Valeur 30
Élément 2	Valeur "foo"
Élément 1	Valeur 8
Élément 3	Valeur ""

Nous avons présenté les paires dans un ordre aléatoire car leur ordre n'a aucune importance.

L'un des avantages d'un tableau associatif est que de nouvelles paires peuvent y être ajoutées à tout moment. Par exemple, supposons que nous ajoutions au tableau ci-dessus un dixième élément dont la valeur est « numéro dix ». Le résultat est le suivant :

```
Élément 10      Valeur « numéro dix »
Élément 4       Valeur 30
Élément 2       Valeur "foo"
Élément 1       Valeur 8
Élément 3       Valeur ""
```

Le tableau est maintenant clairsemé (c'est-à-dire que certains indices sont manquants) : il contient les éléments 1 à 4 et 10, mais pas les éléments 5, 6, 7, 8 ou 9.

Une autre conséquence des tableaux associatifs est que les indices n'ont pas besoin d'être des entiers positifs. N'importe quel nombre, voire une chaîne de caractères, peut servir d'index. Par exemple, voici un tableau qui traduit des mots de l'anglais vers le français :

```
Élément « chien » Valeur « chien »
Élément « chat » Valeur « chat »
Élément « un » Valeur « un »
Élément 1       Valeur « un »
```

Nous avons décidé ici de traduire le nombre 1 à la fois en toutes lettres et sous forme numérique, illustrant ainsi qu'un même tableau peut contenir à la fois des nombres et des chaînes de caractères comme indices.

Lorsque awk crée un tableau pour vous, par exemple avec la fonction intégrée split, les indices de ce tableau sont des entiers consécutifs commençant à 1. (Voir la section 11.3 [Fonctions intégrées pour la manipulation de chaînes], page 90.)

10.2 Référence à un élément de tableau

La principale façon d'utiliser un tableau est de faire référence à l'un de ses éléments. Une référence de tableau est un expression qui ressemble à ceci :

```
tableau[index]
```

Ici, « array » désigne le nom d'un tableau. L'expression « index » correspond à l'indice de l'élément du tableau que vous recherchez.

La valeur de la référence de tableau correspond à la valeur actuelle de l'élément correspondant. Par exemple, foo[4.3] représente l'élément du tableau foo situé à l'indice 4.3.

Si vous faites référence à un élément de tableau sans valeur enregistrée, la valeur de la référence est la chaîne vide. Cela concerne aussi bien les éléments auxquels vous n'avez attribué aucune valeur que les éléments supprimés (voir la section 10.6 [Instruction DELETE], page 85). Une telle référence crée automatiquement l'élément correspondant dans le tableau, avec la chaîne vide comme valeur. (Dans certains cas, cela peut être problématique, car cela risque de gaspiller de la mémoire dans awk).

Vous pouvez déterminer si un élément existe dans un tableau à un certain index grâce à l'expression :

index dans le tableau

Cette expression vérifie si l'élément à l'index spécifié existe, sans créer l'élément s'il n'est pas présent. Elle renvoie 1 (vrai) si l'élément `index` existe, et 0 (faux) sinon.

Par exemple, pour vérifier si le tableau `frequency` contient l'indice « 2 », vous pouvez écrire l'instruction suivante :

si ("2" dans les fréquences) imprimer "L'indice \"2\" est présent."

Notez que ceci ne vérifie pas si le tableau `frequency` contient un élément de valeur « 2 ». (Il n'y a pas d'autre moyen de le faire que de parcourir tous les éléments.) De plus, cela ne crée pas `frequency["2"]`, contrairement à l'alternative (incorrecte) suivante :

if (frequencies["2"] != "") print "L'indice \"2\" est présent."

10.3 Affectation des éléments du tableau

Les éléments d'un tableau sont des lvalues : on peut leur attribuer des valeurs, tout comme aux variables awk :

tableau[indice] = valeur

Ici, « array » est le nom de votre tableau. L'indice de l'expression correspond à l'index de l'élément du tableau auquel vous souhaitez attribuer une valeur. La valeur de l'expression est la valeur que vous attribuez à cet élément du tableau.

10.4 Exemple simple de tableau

Le programme suivant prend en entrée une liste de lignes, chacune commençant par un numéro, et les affiche dans l'ordre de ces numéros. Cependant, à la première lecture, les numéros de ligne sont mélangés. Ce programme trie les lignes en créant un tableau dont les numéros servent d'indices. Il affiche ensuite les lignes dans l'ordre de ces numéros. C'est un programme très simple, qui rencontre des difficultés avec les numéros répétés, les espaces vides ou les lignes ne commençant pas par un numéro.

```
{
    si ($1 > max) max
        = $1 arr[$1]
    = $0
}

FIN
    { pour (x = 1; x <= max; x++) imprimer
        arr[x]
}
```

La première règle conserve la trace du numéro de ligne le plus élevé rencontré jusqu'à présent ; elle stocke également chaque ligne dans le tableau arr, à un index qui est le numéro de la ligne.

La deuxième règle s'exécute une fois que toutes les données d'entrée ont été lues, afin d'imprimer toutes les lignes.

Lorsque ce programme est exécuté avec les entrées suivantes :

```
5 Je suis l'homme Cinq 2 Qui es-
tu ? Le nouveau numéro deux ! 4 .
        . Et quatre sur le sol
1 Qui est le numéro un ?
3 Je vous trois.
```

Voici le résultat :

```
1 Qui est le numéro un ?
2 Qui es-tu ? Le nouveau numéro deux !
3 Je vous trois. 4 .
        . Et quatre sur le sol
5 Je suis l'homme Cinq
```

Si un numéro de ligne est répété, la dernière ligne portant ce numéro prévaut sur les autres.

Les lacunes dans la numérotation des lignes peuvent être gérées par une simple amélioration de la règle END du programme :

```
FIN
    { pour (x = 1; x <= max; x++)
        si (x dans arr)
            imprimer arr[x]
}
```

10.5 Analyse de tous les éléments d'un tableau

Dans les programmes utilisant des tableaux, on a souvent besoin d'une boucle qui parcourt chaque élément du tableau. Dans d'autres langages, où les tableaux sont contigus et les indices limités aux entiers positifs, c'est simple : l'indice maximal est inférieur de 1 à la longueur du tableau, et on trouve tous les indices valides en comptant de zéro jusqu'à cette valeur. Cette technique ne fonctionne pas avec awk, car n'importe quel nombre ou chaîne de caractères peut servir d'indice. C'est pourquoi awk propose une boucle `for` spécifique pour parcourir un tableau :

```
pour (var dans le tableau)
    corps
```

Cette boucle exécute le corps une fois pour chaque valeur différente que votre programme a précédemment utilisée comme index dans le tableau, la variable var étant définie sur cet index.

Voici un programme qui utilise cette forme de l'instruction for. La première règle parcourt les enregistrements d'entrée et note les mots qui apparaissent (au moins une fois) dans l'entrée, en stockant un 1 dans le tableau utilisé avec

Le mot sert d'index. La deuxième règle parcourt les éléments utilisés pour trouver tous les mots distincts présents dans l'entrée. Elle affiche chaque mot de plus de 10 caractères, ainsi que leur nombre. Pour plus d'informations sur la fonction intégrée `length`, consultez le chapitre 11 [Fonctions intégrées], page 89.

```
# Enregistrez un 1 pour chaque mot utilisé au moins une fois. {

    pour (i = 1 ; i <= NF ; i++)
        utilisé[$i] = 1
}

# Trouver le nombre de mots distincts de plus de 10 caractères.
FIN
    { pour (x dans utilisé)
        si (longueur(x) > 10) { +
            +num_long_words
            afficher x
    }
    print num_long_words, "mots de plus de 10 caractères"
}
```

Voir l'annexe B [Programme d'exemple], page 119, pour un exemple plus détaillé de ce type.

L'ordre dans lequel les éléments du tableau sont accédés par cette instruction est déterminé par l'organisation interne des éléments du tableau au sein d'awk et ne peut être ni contrôlé ni modifié. Cela peut poser problème si de nouveaux éléments sont ajoutés au tableau par des instructions dans le corps de la boucle ; il est impossible de prévoir si la boucle `for` les atteindra. De même, modifier la variable `var` à l'intérieur de la boucle peut produire des résultats inattendus. Il est donc préférable d'éviter ce genre de pratiques.

10.6 L'instruction de suppression

Vous pouvez supprimer un élément individuel d'un tableau à l'aide de l'instruction delete :

```
supprimer le tableau[index]
```

Il est impossible de faire référence à un élément d'un tableau après sa suppression ; c'est comme s'il n'avait jamais été utilisé ni même considéré comme une valeur. On ne peut plus accéder à aucune valeur que cet élément possédait auparavant.

Voici un exemple de suppression d'éléments dans un tableau :

```
pour (i dans les fréquences)
    supprimer les fréquences[i]
```

Cet exemple supprime tous les éléments du tableau des fréquences.

Si vous supprimez un élément, une instruction for ultérieure parcourant le tableau ne signalera pas cela. élément, et l'opérateur in pour vérifier la présence de cet élément renverra 0 :

```
supprimer foo[4]
```

```
si (4 dans foo)
    imprimer "Ceci ne sera jamais imprimé"
```

Supprimer un élément qui n'existe pas n'est pas une erreur.

10.7 Utilisation des nombres pour indexer les tableaux

Il est important de se rappeler que les indices des tableaux sont toujours des chaînes de caractères. Si vous utilisez une valeur numérique comme indice, elle sera convertie en chaîne de caractères avant d'être utilisée pour l'indexation (voir la section 8.9 [Conversion des chaînes de caractères et des nombres], page 67).

Cela signifie que la valeur de CONVFMT peut potentiellement affecter la façon dont votre programme accède aux éléments d'un tableau. Par exemple :

```
a = b = 12.153 data[a] =
1 CONVFMT =
"%2.2f" if (b in data) printf "%s
est dans data", b else
        printf "%s n'est pas dans data", b
```

Le programme devrait afficher « 12.15 n'est pas dans les données ». La première instruction attribue la même valeur numérique à a et b. L'affectation à data[a] donne d'abord à a la valeur de chaîne « 12.153 » (en utilisant la valeur de conversion par défaut de CONVFMT, « %.6g »), puis affecte 1 à data["12.153"]. Le programme modifie ensuite la valeur de CONVFMT. Le test « (b dans les données) » force la conversion de b en une chaîne, cette fois « 12.15 », car la valeur de CONVFMT n'autorise que deux chiffres significatifs. Ce test échoue, car « 12.15 » est une chaîne différente de « 12.153 ».

Conformément aux règles de conversion (voir section 8.9 [Conversion des chaînes et des nombres], page 67), les valeurs entières sont toujours converties en chaînes sous forme d'entiers, quelle que soit la valeur de CONVFMT. Ainsi, dans le cas habituel de

```
pour (i = 1; i <= maxsub; i++) faire quelque
        chose avec tableau[i]
```

fonctionnera, quelle que soit la valeur de CONVFMT.

Comme souvent avec awk, le fonctionnement est généralement conforme aux attentes. Toutefois, une connaissance précise des règles en vigueur est utile, car elles peuvent parfois avoir un impact subtil sur vos programmes.

10.8 Tableaux multidimensionnels

Un tableau multidimensionnel est un tableau dans lequel chaque élément est identifié par une séquence d'indices, et non par un seul indice. Par exemple, un tableau bidimensionnel nécessite deux indices. La manière habituelle (dans la plupart des langages, y compris awk) de faire référence à un élément d'un tableau bidimensionnel nommé `grid` est `grid[x,y]`.

Les tableaux multidimensionnels sont pris en charge dans awk grâce à la concaténation des indices en une seule chaîne de caractères. La commande `awk` convertit les indices en chaînes de caractères (voir la section 8.9 [Conversion de chaînes et de nombres], page 67) et les concatène en les séparant par un caractère. On obtient ainsi une chaîne unique décrivant les valeurs des différents indices. Cette chaîne est ensuite utilisée comme indice dans un tableau unidimensionnel classique. Le séparateur utilisé est la valeur de la variable intégrée `SUBSEP`.

Par exemple, supposons que nous évaluions l'expression foo[5,12]="value" lorsque la valeur de SUBSEP est "@". Les nombres 5 et 12 sont convertis en chaînes de caractères et concaténés avec un @ entre eux, ce qui donne "5@12" ; ainsi, l'élément du tableau foo["5@12"] est défini sur "value".

Une fois la valeur de l'élément stockée, awk ne conserve aucune trace de son mode de stockage (index unique ou séquence d'index). Les deux expressions `foo[5,12]` et `foo[5 SUBSEP 12]` auront toujours la même valeur.

La valeur par défaut de SUBSEP est la chaîne « \034 », qui contient un caractère non imprimable.
il est peu probable qu'il apparaisse dans un programme awk ou dans les données d'entrée.

L'intérêt de choisir un caractère improbable réside dans le fait que les valeurs d'index contenant une chaîne correspondant à SUBSEP produisent des chaînes combinées ambiguës. Supposons que SUBSEP soit « @ » ; alors foo["a@b", "c"] et foo["a", "b@c"] seraient indiscernables car les deux seraient en réalité stockées comme foo["a@b@c"]. Comme SUBSEP est « \034 », une telle confusion ne peut survenir que lorsqu'un index contient le caractère de code ASCII 034, ce qui est rare.

Vous pouvez vérifier si une séquence d'indices particulière existe dans un tableau multidimensionnel en utilisant le même opérateur que pour les tableaux unidimensionnels. Au lieu d'un seul indice comme opérande de gauche, indiquez la séquence complète d'indices, séparés par des virgules, entre parenthèses :

(indice1, indice2, . . .) dans le tableau

L'exemple suivant traite ses données d'entrée comme un tableau bidimensionnel de champs ; il effectue une rotation de ce tableau de 90 degrés dans le sens horaire et affiche le résultat. Il suppose que toutes les lignes contiennent le même nombre d'éléments.

```
awk '{ if
        (max_nf < NF) max_nf =
                NF max_nr = NR
        for (x = 1; x <= NF;
        x++) vector[x, NR] = $x

}

FIN
    { pour (x = 1; x <= max_nf; x++) {
                pour (y = max_nr; y >= 1; --y)
                        printf("%s ", vecteur[x, y]) printf("\n")

        }
}'
```

Lorsqu'on lui fournit les données d'entrée :

```
1 2 3 4 5 6
2 3 4 5 6 1 3 4
5 6 1 2
4 5 6 1 2 3
```

il produit :

```
4 3 2 1
5 4 3 2
6 5 4 3 1
6 5 4
2 1 6 5 3
2 1 6
```

10.9 Analyse de tableaux multidimensionnels

Il n'existe pas d'instruction spéciale pour parcourir un tableau « multidimensionnel » ; il ne peut en exister, car en réalité il n'existe pas de tableaux ou d'éléments multidimensionnels ; il n'existe qu'une manière multidimensionnelle d'accéder à un tableau.

Toutefois, si votre programme utilise un tableau multidimensionnel, vous pouvez simuler son parcours en combinant l'instruction `for` (voir section 10.5 [Parcours de tous les éléments d'un tableau], page 84) avec la fonction intégrée `split` (voir section 11.3 [Fonctions intégrées pour la manipulation de chaînes de caractères], page 90). Voici comment cela fonctionne :

```
pour (combiné dans le tableau)
    { diviser(combiné, séparer, SUBSEP)
    . . .
}
```

Cette fonction recherche chaque index concaténé et combiné dans le tableau, puis le divise en ses indices individuels en le décomposant à l'endroit où apparaît la valeur de SUBSEP. Les indices ainsi obtenus deviennent les éléments du tableau séparés.

Ainsi, supposons que vous ayez précédemment stocké `foo` dans `array[1, "foo"]` ; alors un élément d'indice `1\034foo` existe dans le tableau. (Rappelons que la valeur par défaut de `SUBSEP` contient le caractère de code 034.) Tôt ou tard, la boucle `for` trouvera cet indice et effectuera une itération avec `combined` défini sur `1\034foo`. La fonction `split` est alors appelée comme suit :

```
split("1\034foo", separate, "\034")
```

Le résultat est d'attribuer la valeur 1 à separate[1] et la valeur « foo » à separate[2]. Et voilà, la séquence originale des indices de separate est retrouvée.

11 fonctions intégrées

Les fonctions intégrées sont des fonctions toujours disponibles pour votre programme awk. Ce chapitre définit toutes les fonctions intégrées d'awk ; certaines sont mentionnées dans d'autres sections, mais elles sont résumées ici pour votre commodité. (Vous pouvez également définir vos propres fonctions. Voir le chapitre 12 [Fonctions définies par l'utilisateur], page 95.)

11.1 Appel des fonctions intégrées

Pour appeler une fonction intégrée, écrivez le nom de la fonction suivi des arguments entre parenthèses.
Par exemple, atan2(y + z, 1) est un appel à la fonction atan2, avec deux arguments.

Les espaces sont ignorés entre le nom d'une fonction intégrée et la parenthèse ouvrante, mais nous vous recommandons de les éviter. Les fonctions définies par l'utilisateur n'autorisent pas les espaces de cette manière ; pour éviter les erreurs, il est plus simple de suivre une convention simple et toujours efficace : aucun espace après le nom d'une fonction.

Chaque fonction intégrée accepte un certain nombre d'arguments. Dans la plupart des cas, les arguments supplémentaires sont ignorés. Le comportement par défaut des arguments omis varie d'une fonction à l'autre et est décrit dans la documentation de chaque fonction.

Lorsqu'une fonction est appelée, les expressions qui créent les paramètres effectifs de la fonction sont évaluées. complètement avant l'appel de fonction. Par exemple, dans l'extrait de code :

```
i = 4 j =
sqrt(i++)
```

La variable i est initialisée à 5 avant que la fonction sqrt ne soit appelée avec une valeur de 4 pour son paramètre réel.

11.2 Fonctions numériques intégrées

Voici la liste complète des fonctions intégrées qui fonctionnent avec les nombres :

int(x) Cela vous donne la partie entière de x, tronquée vers 0. Cela produit l'entier le plus proche de x, situé entre x et 0.

Par exemple, int(3) vaut 3, int(3,9) vaut 3, int(-3,9) vaut −3 et int(-3) vaut également −3. sqrt(x) donne la racine carrée positive de x. Elle renvoie une erreur si x est négatif. Ainsi, sqrt(4) vaut 2.

exp(x) Cette fonction calcule l'exponentielle de x, ou signale une erreur si x est hors limites. Les valeurs possibles de x dépendent de la représentation en virgule flottante de votre machine.

log(x) Cela vous donne le logarithme népérien de x, si x est positif ; sinon, cela signale une erreur.

sin(x) Cela vous donne le sinus de x, avec x en radians.

cos(x) Cela vous donne le cosinus de x, x étant exprimé en radians.

atan2(y, x) Cela vous donne l'arctangente de y / x en radians.

rand() Cela vous donne un nombre aléatoire. Les valeurs de rand sont uniformément distribuées entre 0 et 1. La valeur n'est jamais 0 ni 1.

Souvent, vous souhaitez plutôt des entiers aléatoires. Voici une fonction définie par l'utilisateur que vous pouvez utiliser pour obtenir un entier non négatif aléatoire inférieur à n :

```
function randint(n) { return int(n *
        rand()) }
}
```

La multiplication produit un nombre réel aléatoire supérieur à 0 et inférieur à n. Nous le transformons ensuite en un entier (en utilisant int) compris entre 0 et n − 1.

Voici un exemple où une fonction similaire est utilisée pour générer des entiers aléatoires compris entre 1 et n. Notez que ce programme affichera un nouveau nombre aléatoire pour chaque enregistrement saisi.

```
awk ' #
Fonction pour lancer un dé virtuel. function roll(n) { return 1 +
int(rand() * n) }

Lancez 3 dés à six faces et affichez le nombre total de points. {

        printf("%d points\n", roll(6)+roll(6)+roll(6))
}'
```

Remarque : la fonction `rand` génère des nombres à partir du même point initial (ou graine) à chaque exécution d'`awk`. Cela signifie qu'un programme produira les mêmes résultats à chaque exécution.
Les nombres sont aléatoires au sein d'une même exécution d'awk, mais prévisibles d'une exécution à l'autre. C'est pratique pour le débogage, mais si vous souhaitez qu'un programme effectue des actions différentes à chaque utilisation, vous devez modifier la valeur initiale (seed) pour qu'elle soit différente à chaque exécution. Pour ce faire, utilisez la commande `srand`.

srand(x) La fonction srand définit le point de départ, ou graine, pour générer des nombres aléatoires à la valeur x.

Chaque valeur initiale génère une séquence particulière de nombres aléatoires. Ainsi, si vous définissez la valeur initiale une seconde fois, vous obtiendrez à nouveau la même séquence de nombres aléatoires.

Si vous omettez l'argument x, comme dans srand(), la date et l'heure actuelles servent de graine. C'est ainsi que l'on obtient des nombres aléatoires véritablement imprévisibles.

La fonction srand renvoie la graine précédente. Cela facilite le suivi des graines pour la génération de séquences de nombres aléatoires reproduisant fidèlement ces séquences.

11.3 Fonctions intégrées pour la manipulation de chaînes de caractères

Les fonctions de cette section examinent ou modifient le texte d'une ou plusieurs chaînes de caractères.

index(dans, trouver)

Cette fonction recherche la première occurrence de la chaîne « find » dans la chaîne « in » et renvoie sa position, en caractères, au début de cette occurrence. Par exemple :

```
awk 'BEGIN { print index("peanut", "an") }'
```

Affiche « 3 ». Si la chaîne est introuvable, l'index renvoie 0. (N'oubliez pas que les indices de chaînes dans awk commencent

à 1.) longueur(chaîne)

Cette fonction renvoie le nombre de caractères dans la chaîne. Si la chaîne est un nombre, elle renvoie la longueur de la chaîne de chiffres représentant ce nombre. Par exemple : longueur("abcde")

est 5. En revanche, longueur(15 * 35) est égal à 3. Comment ? Eh bien, 15 * 35 = 525, et 525 est ensuite converti en la chaîne '"525"', qui comporte trois caractères.

Si aucun argument n'est fourni, la fonction length renvoie la longueur de $0.

Dans les anciennes versions d'awk, il était possible d'appeler la fonction length sans parenthèses.
Cette pratique est considérée comme « dépréciée » dans la norme POSIX. Cela signifie que, bien que vous puissiez l'utiliser dans vos programmes, cette fonctionnalité pourrait être supprimée dans une version ultérieure de la norme. Par conséquent, pour une portabilité optimale de vos programmes awk, vous devez toujours utiliser les parenthèses. `match(string, regexp)`

La fonction `match` recherche dans la chaîne de caractères `string` la plus longue sous-chaîne, située à gauche, correspondant à l'expression régulière `regexp`. Elle renvoie la position du caractère où commence cette sous-chaîne (1 si elle commence au début de la chaîne). Si aucune correspondance n'est trouvée, elle renvoie 0.

La fonction de correspondance affecte à la variable intégrée RSTART l'index. Elle affecte également à la variable intégrée RLENGTH la longueur, en caractères, de la sous-chaîne correspondante. Si aucune correspondance n'est trouvée, RSTART est initialisée à 0 et RLENGTH à -1.

Par exemple :

```
awk '{
        si ($1 == "FIND") regex =
            $2 sinon
        { where
            = match($0, regex) si (where) print
            "Correspondance
                de", regex, "trouvé à", where, "dans", $0
        }
}'
```

Ce programme recherche les lignes correspondant à l'expression régulière stockée dans la variable `regex`. Cette expression régulière est modifiable. Si le premier mot d'une ligne est « FIND », `regex` est remplacé par le deuxième mot de cette ligne. Par conséquent, étant donné :

```
TROUVER fo*bar

Mon programme était un foobar
Mais rien de tout cela ne ferait de doobar
TROUVER Melvin
JF+KM

Cette ligne est la propriété de The Reality Engineering Co.
Ce fichier a été créé par Melvin. awk
```

affiche :

```
Correspondance de fo*bar trouvée à 18 dans Mon programme était un foobar
Correspondance de Melvin trouvée à 26 dans Ce fichier a été créé par Melvin.
```

diviser(chaîne, tableau, champsep)

Cela divise la chaîne en morceaux séparés par fieldsep, et stocke les morceaux dans un tableau.
Le premier élément est stocké dans array[1], le deuxième dans array[2], et ainsi de suite. La valeur de chaîne du troisième argument, fieldsep, est une expression régulière décrivant où découper la chaîne (de la même manière que FS peut être une expression régulière décrivant où découper les enregistrements d'entrée). Si fieldsep est omis, la valeur de FS est utilisée. split renvoie le nombre d'éléments créés.

La fonction split divise donc les chaînes de caractères en morceaux, de la même manière que les lignes d'entrée sont divisées en champs. Par exemple : split("auto-da-fe",

```
a, "-") divise la chaîne 'auto-da-fe' en trois
```

champs en utilisant le contenu du tableau a, comme suit : '-' comme séparateur. Il définit le

```
a[1] = "auto" a[2] =
"da" a[3] = "fe"
```

La valeur renvoyée par cet appel à split est 3.

Comme pour la division des champs de saisie, lorsque la valeur de fieldsep est " ", les espaces blancs de début et de fin sont ignorés et les éléments sont séparés par des séquences d'espaces blancs.

sprintf(format, expression1,. . .)

Cette fonction renvoie (sans affichage) la chaîne de caractères que `printf` aurait affichée avec les mêmes arguments (voir la section 4.5 [Utilisation des instructions `printf` pour un affichage plus sophistiqué], page 38). Par exemple : `sprintf("pi = %.2f

```
(approx.)", 22/7)`
```

renvoie la chaîne "pi = 3,14 (approx.)".

sous-(regexp, remplacement, cible)

La sous-fonction modifie la valeur de la variable cible. Elle recherche dans cette valeur, qui doit être une chaîne de caractères, la première sous-chaîne correspondant à l'expression régulière `regexp`, en étendant cette correspondance autant que possible. Ensuite, la chaîne entière est modifiée en remplaçant le texte correspondant par le texte de remplacement. La chaîne modifiée devient la nouvelle valeur de la variable cible.

Cette fonction est particulière car la cible ne sert pas simplement à calculer une valeur, et n'importe quelle expression ne convient pas : il doit s'agir d'une variable, d'un champ ou d'une référence de tableau, afin que la fonction puisse y stocker une valeur modifiée. Si cet argument est omis, la valeur par défaut est utilisée et modifiée : $0.

Par exemple :

```
str = "water, water, everywhere" sub(/at/, "ith", str)
```

définit str à "wither, water, everywhere", en remplaçant l'occurrence la plus à gauche et la plus longue de 'at' par 'ith'.

La sous-fonction renvoie le nombre de substitutions effectuées (un ou zéro).

Si le caractère spécial « & » apparaît en remplacement, il représente la sous-chaîne précise qui a été identifiée par l'expression régulière. (Si l'expression régulière peut correspondre à plusieurs chaînes, cette sous-chaîne précise peut varier.) Par exemple : `awk '{ sub(/candidate/, "&

```
and his wife"); print }'` remplace la première occurrence de « candidate » par
```

« candidate and his wife » sur chaque ligne d'entrée.

Voici un autre exemple : awk 'BEGIN

```
{ str = "daabaaa"
        sub(/a*/, "c&c", str) print
        str

}'
```

Affiche « dcaacbaaa ». Ceci montre comment « & » peut représenter une chaîne non constante, et illustre également la règle « l'élément le plus à gauche, le plus long ».

L'effet du caractère spécial « & » peut être désactivé en plaçant une barre oblique inverse avant celui-ci dans la chaîne. Comme d'habitude, pour insérer une barre oblique inverse, il faut en écrire deux. Par conséquent, écrivez « \\& » dans une constante de chaîne pour inclure un « & » littéral dans le remplacement. Par exemple, voici comment remplacer le premier « | » de chaque ligne par un « & » :

```
awk '{ sub(/\|/, "\\&"); print }'
```

Remarque : comme indiqué précédemment, le troisième argument de `sub` doit être une lvalue. Certaines versions d'awk autorisent une expression qui n'est pas une lvalue comme troisième argument.

Dans ce cas, la fonction `sub` rechercherait toujours le motif et renverrait 0 ou 1, mais le résultat de la substitution (le cas échéant) serait ignoré faute de place. Ces versions d'awk acceptent des expressions comme celle-ci : `sub(/USA/, "United States", "the USA and

Canada")`

Mais cela est considéré comme erroné à Gawk.

gsub(regexp, remplacement, cible)

Cette fonction est similaire à la fonction `sub`, à la différence que `gsub` remplace toutes les sous-chaînes correspondantes les plus longues, les plus à gauche et non chevauchantes qu'elle trouve. Le « g » de `gsub` signifie « global », c'est-à-dire qu'il remplace partout. Par exemple :

awk '{ gsub(/Britain/, "United Kingdom"); print }' remplace toutes les occurrences

de la chaîne 'Britain' par 'United Kingdom' pour tous les enregistrements d'entrée.

La fonction gsub renvoie le nombre de substitutions effectuées. Si la variable cible à rechercher et à modifier est omise, l'enregistrement d'entrée complet, $0, est utilisé.

Comme dans sub, les caractères '&' et '\' sont spéciaux, et le troisième argument doit être une lvalue.

sous-chaîne(chaîne, début, longueur)

Cette fonction renvoie une sous-chaîne de la chaîne de caractères de longueur `length`, commençant au caractère numéro `start`. Le premier caractère d'une chaîne est le caractère numéro 1. Par exemple, `substr("washington", 5, 3)` renvoie `ing`.

Si la longueur n'est pas spécifiée, cette fonction renvoie le suffixe complet de la chaîne commençant au numéro de début. Par exemple, substr("washington", 5) renvoie "ington".
C'est également le cas si la longueur est supérieure au nombre de caractères restants dans la chaîne, en comptant à partir du premier caractère. `tolower(string)`

Cette fonction renvoie une copie de la chaîne, où chaque caractère majuscule est remplacé par son caractère minuscule correspondant. Les caractères non alphabétiques restent inchangés.
Par exemple, tolower("MiXeD cAsE 123") renvoie "mixed case 123".

toupper(chaîne)

Cette fonction renvoie une copie de la chaîne, où chaque caractère minuscule est remplacé par sa majuscule correspondante. Les caractères non alphabétiques restent inchangés.
Par exemple, toupper("MiXeD cAsE 123") renvoie "MIXED CASE 123".

11.4 Fonctions intégrées pour les entrées/sorties

fermer(nom_de_fichier)

Ferme le fichier nommé « filename », utilisé en entrée ou en sortie. L'argument peut également être une commande shell servant à rediriger le flux vers ou depuis un tube ; dans ce cas, le tube est fermé.

Voir la section 3.8 [Fermeture des fichiers et tubes d'entrée], page 33, concernant la fermeture des fichiers et tubes d'entrée. Voir la section 4.6.2 [Fermeture des fichiers et tubes de sortie], page 43, concernant la fermeture des fichiers et tubes de sortie.

système(commande)

La fonction système permet à l'utilisateur d'exécuter des commandes du système d'exploitation puis de revenir au programme awk. Elle exécute la commande spécifiée par la chaîne de caractères « command » et renvoie le statut de cette commande.

Par exemple, si le fragment de code suivant est inséré dans votre programme awk :

```
FIN
    { système("mail -s 'awk run done' operator < /dev/null")
}
```

L'opérateur système recevra un courriel lorsque le programme awk aura terminé le traitement des données d'entrée et commencera son traitement de fin de traitement.

Notez qu'un résultat similaire peut être obtenu en redirigeant `print` ou `printf` vers un tube. Cependant, si votre programme `awk` est interactif, `system` est utile pour lancer des programmes autonomes de grande envergure, tels qu'un interpréteur de commandes ou un éditeur.

Certains systèmes d'exploitation ne peuvent pas implémenter la fonction système. Le système provoque une erreur fatale s'il n'est pas pris en charge.

Contrôle de la mise en mémoire tampon de sortie avec le système

De nombreux utilitaires mettent en mémoire tampon leurs données de sortie ; ils stockent en mémoire les informations à écrire sur un fichier disque ou dans un terminal, jusqu'à ce qu'il y en ait suffisamment pour une seule opération. Cette méthode est souvent plus efficace que d'écrire chaque donnée dès qu'elle est prête. Cependant, il est parfois nécessaire de forcer un programme à vider ses tampons ; c'est-à-dire à écrire les informations à destination, même si un tampon n'est pas plein. Vous pouvez le faire depuis votre programme awk en appelant la fonction `system` avec une chaîne vide comme argument :

```
système("") # vider la sortie
```

12 Fonctions définies par l'utilisateur

Les programmes awk complexes peuvent souvent être simplifiés en définissant vos propres fonctions. Les fonctions définies par l'utilisateur peuvent être appelées comme les fonctions intégrées (voir la section 8.12 [Appels de fonctions], page 70), mais c'est à vous de les définir, c'est-à-dire d'indiquer à awk ce qu'elles doivent faire.

12.1 Syntaxe des définitions de fonctions

Les définitions de fonctions peuvent apparaître n'importe où entre les règles du programme awk. Ainsi, la forme générale d'un programme awk est étendue pour inclure des séquences de règles et des définitions de fonctions définies par l'utilisateur.

La définition d'une fonction nommée name ressemble à ceci :

```
nom de la fonction (liste des paramètres) { corps
        de la fonction
}
```

Le nom de la fonction à définir est `name`. Un nom de fonction valide est, comme un nom de variable valide, une suite de lettres, de chiffres et de tirets bas, ne commençant pas par un chiffre. Les fonctions partagent le même ensemble de noms que les variables et les tableaux.

La liste des paramètres contient les arguments de la fonction et les noms des variables locales, séparés par des virgules. Lors de l'appel de la fonction, les noms des arguments sont utilisés pour stocker les valeurs des arguments passés en paramètre. Les variables locales sont initialisées à la chaîne vide.

Le corps de la fonction est constitué d'instructions awk. C'est la partie la plus importante de la définition, car elle décrit ce que la fonction doit faire. Les noms des arguments permettent au corps de la fonction de les mentionner ; les variables locales, quant à elles, permettent de stocker des valeurs temporaires.

Les noms des arguments ne sont pas distingués syntaxiquement des noms des variables locales ; c'est le nombre d'arguments fournis lors de l'appel de la fonction qui détermine le nombre de variables d'arguments. Ainsi, si trois valeurs d'arguments sont données, les trois premiers noms de la liste des paramètres sont des arguments, et les suivants sont des variables locales.

Il s'ensuit que si le nombre d'arguments n'est pas le même dans tous les appels à la fonction, certains noms de la liste des paramètres peuvent être des arguments dans certains cas et des variables locales dans d'autres.
On peut aussi considérer que les arguments omis sont remplacés par défaut par la chaîne nulle.

En général, lorsqu'on écrit une fonction, on sait combien de noms on utilisera comme arguments et combien comme variables locales. Par convention, il est conseillé d'insérer un espace supplémentaire entre les arguments et les variables locales, afin que les autres personnes comprennent comment la fonction est censée être utilisée.

Lors de l'exécution du corps de la fonction, les arguments et les valeurs des variables locales masquent les variables portant le même nom utilisées dans le reste du programme. Ces variables masquées ne sont pas accessibles dans la définition de la fonction, car il est impossible de les nommer tant que leurs noms sont réservés aux variables locales. Toutes les autres variables utilisées dans le programme awk peuvent être référencées ou définies normalement dans la définition de la fonction.

Les arguments et les variables locales ne sont conservés que pendant l'exécution du corps de la fonction. Une fois que
Une fois la carrosserie terminée, les variables restées dans l'ombre réapparaissent.

Le corps d'une fonction peut contenir des expressions qui appellent d'autres fonctions. Ces expressions peuvent même appeler cette fonction, soit directement, soit par l'intermédiaire d'une autre fonction. Dans ce cas, on dit que la fonction est récursive.

En awk, il n'est pas nécessaire de placer la définition d'une fonction avant toutes ses utilisations.
C'est parce qu'awk lit l'intégralité du programme avant de commencer à en exécuter la moindre partie.

12.2 Exemple de définition de fonction

Voici un exemple de fonction définie par l'utilisateur, appelée myprint, qui prend un nombre et l'affiche.
il dans un format spécifique.

```
fonction myprint(num) {

        printf "%6.3g\n", num
}
```

Pour illustrer cela, voici une règle awk qui utilise notre fonction myprint :

```
3 $ > 0          { myprint($3) }
```

Ce programme affiche, dans notre format spécifique, tous les troisièmes champs contenant un nombre positif dans nos données d'entrée. Par conséquent, lorsque l'on nous donne :

```
  1,2 3,4 5,6 7,8 9,10 11,12 -13,14
  15,16
17.18 19.20 21.22 23.24
```

Ce programme, utilisant notre fonction pour formater les résultats, affiche :

```
   5.6
  21.2
```

Voici un exemple assez artificiel de fonction récursive. Elle affiche une chaîne de caractères à l'envers :

```
fonction rev (str, len) { si (len == 0)
        { printf "\n"

                retour

        printf "%c", substr(str, len, 1) rev(str, len - 1)

}
```

12.3 Appel de fonctions définies par l'utilisateur

Appeler une fonction signifie la faire s'exécuter et accomplir sa tâche. Un appel de fonction est un l'expression, et sa valeur est la valeur renvoyée par la fonction.

Un appel de fonction se compose du nom de la fonction suivi des arguments entre parenthèses. Les arguments saisis dans l'appel sont des expressions awk ; à chaque exécution, ces expressions sont évaluées et leurs valeurs constituent les arguments effectifs. Par exemple, voici un appel à la fonction `foo` avec trois arguments (le premier étant une concaténation de chaînes de caractères) :

```
foo(xy, "perdre", 4 * z)
```

Attention : les espaces et les tabulations ne sont pas autorisés entre le nom de la fonction et la parenthèse ouvrante de la liste des arguments. Si vous insérez un espace par erreur, awk pourrait interpréter cela comme une tentative de concaténation d'une variable avec une expression entre parenthèses. Or, il détectera que vous avez utilisé un nom de fonction et non un nom de variable, et signalera une erreur.

Lorsqu'une fonction est appelée, elle reçoit une copie des valeurs de ses arguments. On parle alors d'appel par valeur. L'appelant peut utiliser une variable comme expression pour l'argument, mais la fonction appelée l'ignore : elle connaît uniquement la valeur de l'argument. Par exemple, si vous écrivez le code suivant :

```
foo = "bar"
z = mafonction(foo)
```

Vous ne devez donc pas considérer l'argument de myfunc comme étant « la variable foo », mais plutôt comme la valeur de chaîne « bar ».

Si la fonction myfunc modifie les valeurs de ses variables locales, cela n'a aucun effet sur les autres variables. variables. En particulier, si ma fonction fait ceci :

```
fonction myfunc (win) { print win win =
    "zzz"

    imprimer gagner
}
```

Modifier la variable `win`, son premier argument, ne change pas la valeur de `foo` dans le programme appelant. Le rôle de `foo` dans l'appel à `myfunc` s'est terminé lorsque sa valeur, `bar`, a été calculée. Si `win` existe également en dehors de `myfunc`, le corps de la fonction ne peut pas modifier cette valeur externe, car elle est masquée pendant l'exécution de `myfunc` et ne peut être ni vue ni modifiée depuis cet environnement.

Cependant, lorsque des tableaux sont passés en paramètres à des fonctions, ils ne sont pas copiés. Le tableau lui-même est directement accessible à la fonction pour manipulation. On parle alors d'appel par référence. Les modifications apportées à un tableau en paramètre à l'intérieur du corps d'une fonction sont visibles en dehors de cette fonction. Cela peut s'avérer très dangereux si l'on n'est pas vigilant. Par exemple :

```
fonction changeit (tableau, ind, nvalue) { tableau[ind] = nvalue
```

```
}

COMMENCER {
                a[1] = 1 ; a[2] = 2 ; a[3] = 3 changeit(a, 2,
                "deux") printf "a[1] = %s, a[2] =
                %s, a[3] = %s\n", a[1], a[2], a[3]
        }
```

affiche 'a[1] = 1, a[2] = deux, a[3] = 3', car l'appel à changeit stocke "deux" dans le deuxième élément de a.

12.4 Déclaration de retour

Le corps d'une fonction définie par l'utilisateur peut contenir une instruction `return`. Cette instruction rend le contrôle au reste du programme awk. Elle peut également servir à renvoyer une valeur utilisable par la suite dans le programme awk. Voici à quoi elle ressemble :

```
expression de retour
```

La partie expression est facultative. Si elle est omise, la valeur renvoyée est indéfinie et, Par conséquent, imprévisible.

Une instruction de retour sans expression de valeur est supposée à la fin de chaque définition de fonction. Si l'exécution atteint la fin du corps de la fonction, celle-ci renvoie une valeur imprévisible. awk ne vous avertit pas si vous utilisez la valeur de retour d'une telle fonction ; vous obtiendrez simplement des résultats imprévisibles ou inattendus.

Voici un exemple de fonction définie par l'utilisateur qui renvoie la valeur du plus grand nombre parmi les éléments d'un tableau :

```
fonction maxelt (vec, i, ret) { pour (i dans vec) { si
        (ret == "" || vec[i] > ret)
                ret = vec[i]

        }
        retourner ret
}
```

Vous appelez `maxelt` avec un argument, qui est le nom d'un tableau. Les variables locales `i` et `ret` ne sont pas destinées à être des arguments ; bien que rien ne vous empêche de passer deux ou trois arguments à `maxelt`, le comportement serait inattendu. L'espace supplémentaire avant `i` dans la liste des paramètres de la fonction indique que `i` et `ret` ne doivent pas être des arguments. Il s'agit d'une convention à respecter lors de la définition de fonctions.

Voici un programme qui utilise notre fonction maxelt. Il charge un tableau, appelle maxelt, puis indique le nombre maximal dans ce tableau :

```
awk '
fonction maxelt (vec, i, ret) { for (i in vec) { if (ret == "" ||
        vec[i] > ret) ret = vec[i]


        }
        retourner ret
}
# Charger tous les champs de chaque enregistrement dans nums.
{
                pour (i = 1 ; i <= NF ; i++) nums[NR,
                        i] = $i
}

FIN
    { print maxelt(nums)
}'
```

Compte tenu des données d'entrée suivantes :

```
  1 5 23 8 16 44
3 5 2 8 26
256 291 1396 2962 100
-6 467 998 1101 99385
11 0 225
```

Notre programme nous indique (sans surprise) que :

```
99385
```

est le plus grand nombre de notre tableau.

13 variables intégrées

La plupart des variables awk sont à votre disposition pour vos propres besoins ; elles ne changent jamais sauf lorsque votre programme leur attribue des valeurs, et n'affectent rien sauf lorsque votre programme les examine.

Certaines variables ont des significations intrinsèques spécifiques. Certaines sont analysées automatiquement par awk, ce qui vous permet de lui indiquer comment effectuer certaines opérations. D'autres sont initialisées automatiquement par awk afin de transmettre des informations de son fonctionnement interne à votre programme.

Ce chapitre répertorie toutes les variables intégrées d'awk. La plupart d'entre elles sont également documentées dans les chapitres décrivant leurs domaines d'application.

13.1 Variables intégrées qui contrôlent awk

Voici la liste des variables que vous pouvez modifier pour contrôler le comportement d'awk.

CONVFMT est une chaîne de caractères utilisée par awk pour contrôler la conversion des nombres en chaînes de caractères (voir la section 8.9 [Conversion de chaînes et de nombres], page 67). Elle est passée comme premier argument à la fonction sprintf. Sa valeur par défaut est « %.6g ». CONVFMT a été introduite par la norme POSIX.

FS
FS est le séparateur de champs d'entrée (voir la section 3.5 [Spécification du mode de séparation des champs], page 25). Sa valeur est une chaîne de caractères unique ou une expression régulière à plusieurs caractères correspondant aux séparations entre les champs d'un enregistrement d'entrée.

La valeur par défaut est « » (une chaîne de caractères composée d'un seul espace). Par exception, cette valeur signifie que toute séquence d'espaces et de tabulations constitue un seul séparateur. Elle a également pour effet d'ignorer les espaces et les tabulations en début ou en fin de ligne.

Vous pouvez définir la valeur de FS sur la ligne de commande à l'aide de l'option '-F' :

```
awk -F, 'programme' fichiers-entrée
```

OFMT
Cette chaîne est utilisée par awk pour contrôler la conversion des nombres en chaînes (voir la section 8.9 [Conversion des chaînes et des nombres], page 67) pour l'impression avec l'instruction print.
Elle fonctionne en étant passée, en pratique, comme premier argument à la fonction sprintf. Sa valeur par défaut est « %.6g ».
Les versions précédentes d'awk utilisaient également OFMT pour spécifier le format de conversion des nombres en chaînes de caractères dans les expressions générales ; cette fonctionnalité a été remplacée par CONVFMT.

OFS
Il s'agit du séparateur de champs de sortie (voir section 4.3 [Séparateurs de sortie], page 37). Il est inséré entre les champs affichés par une instruction d'impression. Sa valeur par défaut est « », une chaîne de caractères composée d'un seul espace.

ORS
Il s'agit du séparateur d'enregistrements de sortie. Il est affiché à la fin de chaque instruction d'impression.
Sa valeur par défaut est une chaîne contenant un seul caractère de nouvelle ligne, qui peut s'écrire « \n ». (Voir la section 4.3 [Séparateurs de sortie], page 37.)

RS
Il s'agit du séparateur d'enregistrements d'entrée d'awk. Sa valeur par défaut est une chaîne contenant un seul caractère de nouvelle ligne, ce qui signifie qu'un enregistrement d'entrée est constitué d'une seule ligne de texte.
(Voir la section 3.1 [Comment les données d'entrée sont divisées en enregistrements], page 21.)

SUBSEP
SUBSEP est le séparateur d'indices. Sa valeur par défaut est « \034 » et il sert à séparer les parties du nom d'un tableau multidimensionnel. Ainsi, si vous accédez à

foo[12,3], il accède en réalité à foo["12\0343"] (voir la section 10.8 [Tableaux multidimensionnels], page 86).

13.2 Variables intégrées qui transmettent des informations

Voici la liste des variables définies automatiquement par awk dans certaines situations afin de fournir des informations à votre programme.

ARGC
ARGV

Les arguments de ligne de commande disponibles pour les programmes awk sont stockés dans un tableau appelé ARGV. ARGC représente le nombre d'arguments de ligne de commande présents. Voir le chapitre 14 [Invocation d'awk], page 105. ARGV est indexé de zéro à ARGC - 1. Par exemple :

```
awk 'DÉBUT {
        for (i = 0; i < ARGC; i++) print ARGV[i] }'
            inventory-shipped
    BBS-list Dans cet exemple, ARGV[0] contient
```

"awk", ARGV[1] contient "inventory-shipped" et ARGV[2] contient "BBS-list". La valeur de ARGC est 3, soit 1 de plus que l'indice du dernier élément de ARGV puisque les éléments sont numérotés à partir de zéro.

Les noms ARGC et ARGV, ainsi que la convention d'indexation du tableau de 0 à ARGC - 1, sont dérivés de la méthode du langage C d'accès aux arguments de la ligne de commande. ments.

Notez que le programme awk n'est pas inclus dans ARGV. Les autres options spéciales de la ligne de commande, ainsi que leurs arguments, ne le sont pas non plus. En revanche, les affectations de variables sur la ligne de commande sont traitées comme des arguments et apparaissent donc dans le tableau ARGV.

Votre programme peut modifier ARGC et les éléments de ARGV. À chaque fois qu'awk atteint la fin d'un fichier d'entrée, il utilise l'élément suivant de ARGV comme nom du fichier d'entrée suivant. En y stockant une chaîne différente, votre programme peut modifier les fichiers lus.
Vous pouvez utiliser « - » pour représenter l'entrée standard. En stockant des éléments supplémentaires et en incrémentant ARGC, vous pouvez forcer la lecture de fichiers supplémentaires.

Si vous diminuez la valeur de ARGC, cela élimine les fichiers d'entrée de la fin de la liste.
En enregistrant ailleurs l'ancienne valeur de ARGC, votre programme peut traiter les arguments supprimés autrement que comme des noms de fichiers.

Pour supprimer un fichier du milieu de la liste, stockez la chaîne vide (« ») dans ARGV à la place du nom du fichier. Particularité d'awk : les noms de fichiers remplacés par la chaîne vide sont ignorés.

ENVIRON est un tableau contenant les valeurs de l'environnement. Les indices du tableau correspondent aux noms des variables d'environnement ; les valeurs sont les valeurs de ces variables. Par exemple, ENVIRON["HOME"] peut être égal à '/u/close'. Modifier ce tableau n'affecte pas l'environnement transmis aux programmes qu'awk peut lancer par redirection ou via la fonction système.

Certains systèmes d'exploitation peuvent ne pas disposer de variables d'environnement. Sur ces systèmes, le tableau ENVIRON est vide.

FILENAME : Il s'agit du nom du fichier que awk est en train de lire. Si awk lit depuis l'entrée standard (c'est-à-dire si aucun fichier n'est spécifié sur la ligne de commande), FILENAME est défini sur « - ». FILENAME est modifié à chaque lecture d'un nouveau fichier (voir le chapitre 3 [Lecture des fichiers d'entrée], page 21).

FNR FNR est le numéro d'enregistrement actuel dans le fichier actuel. FNR est incrémenté à chaque fois.
Un nouvel enregistrement est lu (voir section 3.7 [Entrée explicite avec getline], page 30).
réinitialisé à 0 à chaque fois qu'un nouveau fichier d'entrée est créé.

NF NF représente le nombre de champs dans l'enregistrement d'entrée actuel. NF est défini à chaque nouvel enregistrement.
est lu lors de la création d'un nouveau champ ou lorsque $0 change (voir section 3.2 [Examen de
Champs], page 22).

Non. Il s'agit du nombre d'enregistrements d'entrée traités par awk depuis le début de l'opération.
Exécution du programme. (Voir la section 3.1 [Comment l'entrée est divisée en enregistrements], page 21). NR
est défini à chaque fois qu'un nouvel enregistrement est lu.

RLENGTH RLENGTH est la longueur de la sous-chaîne correspondant à la fonction de correspondance (voir section 11.3) .
[Fonctions intégrées pour la manipulation de chaînes de caractères], page 90). RLENGTH est défini en appelant la
Fonction de correspondance. Sa valeur est la longueur de la chaîne correspondante, ou −1 si aucune correspondance n'a été trouvée.
trouvé.

RSTART RSTART est l'indice de début, en caractères, de la sous-chaîne correspondant à la fonction de correspondance.
(voir la section 11.3 [Fonctions intégrées pour la manipulation de chaînes], page 90). RSTART est
définie en appelant la fonction match. Sa valeur est la position de la chaîne où le
commence la sous-chaîne correspondante, ou 0 si aucune correspondance n'a été trouvée.

14. Invocation d'awk

Il existe deux façons d'exécuter awk : soit avec un programme explicite, soit avec un ou plusieurs fichiers de programme. Voici des modèles pour les deux ; les éléments entre crochets « […] » dans ces modèles sont facultatifs.

14.1 Options de ligne de commande

Les options commencent par un signe moins et ne comportent qu'un seul caractère. Si l'option prend un argument, le mot-clé est immédiatement suivi du signe égal (« = ») et de la valeur de l'argument. Par souci de concision, la discussion ci-dessous ne porte que sur les options courtes classiques ; cependant, les options longues et courtes sont interchangeables dans tous les contextes.

-F fs
: Définit la variable FS sur fs (voir la section 3.5 [Spécification de la façon dont les champs sont séparés], page 25).

-f fichier source
: Indique que le programme awk se trouve dans le fichier source au lieu du premier argument non optionnel.

-v var=val
: La variable var prend la valeur val avant le début de l'exécution du programme. Ces valeurs de variables sont accessibles à l'intérieur de la règle BEGIN (voir ci-dessous pour une explication plus détaillée).
L'option '-v' ne peut définir qu'une seule variable, mais vous pouvez l'utiliser plusieurs fois, en définissant une variable différente à chaque fois, comme ceci : '-v foo=1 -v bar=2'.

Toute autre option est signalée comme invalide avec un message d'avertissement, mais est par ailleurs ignorée.

Si l'option « -f » n'est pas utilisée, le premier argument de ligne de commande qui n'est pas une option est attendu.
être le texte du programme.

L'option `-f` peut être utilisée plusieurs fois sur la ligne de commande. Dans ce cas, `awk` lit le code source de son programme depuis tous les fichiers spécifiés, comme s'ils étaient concaténés en un seul fichier. Ceci est utile pour créer des bibliothèques de fonctions `awk`. Les fonctions utiles peuvent ainsi être écrites une seule fois, puis récupérées depuis un emplacement standard, au lieu d'être incluses dans chaque programme. Vous pouvez toujours saisir un programme dans le terminal et utiliser les fonctions de la bibliothèque en spécifiant `-f /dev/tty`. `awk` lira alors un fichier depuis le terminal pour l'utiliser dans son programme. Après avoir saisi votre programme, appuyez sur Ctrl+D (le caractère de fin de fichier) pour le terminer. (Vous pouvez également utiliser `-f -` pour lire le code source du programme depuis l'entrée standard, mais vous ne pourrez alors pas utiliser l'entrée standard comme source de données.)

14.2 Autres arguments de ligne de commande

Tout argument supplémentaire sur la ligne de commande est généralement considéré comme un fichier d'entrée à traiter dans l'ordre spécifié. Cependant, un argument de la forme `var=valeur` signifie affecter la valeur `valeur` à la variable `var` ; il ne spécifie aucun fichier.

Tous ces arguments sont mis à la disposition de votre programme awk dans le tableau ARGV (voir le chapitre 13 [Variables intégrées], page 101). Les options de ligne de commande et le texte du programme (le cas échéant) sont exclus du tableau ARGV. Tous les autres arguments, y compris les affectations de variables, y sont inclus.

La distinction entre les arguments de nom de fichier et les arguments d'affectation de variable est effectuée lorsqu'awk s'apprête à ouvrir le fichier d'entrée suivant. À ce moment de l'exécution, awk vérifie si le « nom de fichier » correspond bien à une affectation de variable ; si c'est le cas, awk affecte la variable au lieu de lire le fichier.

Par conséquent, les variables reçoivent effectivement les valeurs spécifiées une fois que tous les fichiers précédemment spécifiés ont été lus. En particulier, les valeurs des variables assignées de cette manière ne sont pas disponibles à l'intérieur d'une règle BEGIN (voir la section 6.7 [Modèles spéciaux BEGIN et END], page 53), car ces règles sont exécutées avant qu'awk ne commence l'analyse de la liste des arguments. Les valeurs fournies sur la ligne de commande sont traitées pour détecter les séquences d'échappement (voir la section 8.1 [Expressions constantes], page 57).

Dans certaines versions antérieures d'awk, lorsqu'une affectation de variable survenait avant tout nom de fichier, cette affectation avait lieu avant l'exécution de la règle BEGIN. Certaines applications dépendaient de cette fonctionnalité. Lors de la modification d'awk pour plus de cohérence, l'option `-v` a été ajoutée afin de prendre en charge les applications qui dépendaient de ce comportement antérieur.

La fonction d'affectation de variables est particulièrement utile pour définir des variables telles que RS, OFS et ORS, qui contrôlent les formats d'entrée et de sortie, avant l'analyse des fichiers de données. Elle permet également de gérer l'état du système en cas de plusieurs passages sur un fichier de données. Par exemple :

```
awk 'pass == 1 { passer 1 truc }
     pass == 2 { passer 2 éléments }' pass=1 fichier de données pass=2 fichier de données
```

Compte tenu de la fonctionnalité d'affectation de variables, l'option « -F » n'est pas strictement nécessaire. Elle reste disponible pour Compatibilité historique.

Annexe A Résumé awk

Cette annexe fournit un bref résumé de la ligne de commande awk et du langage awk.
Conçu pour servir de « guide de référence rapide », il est donc concis, mais complet.

A.1 Résumé des options de ligne de commande

La ligne de commande comprend les options de la commande awk elle-même, le texte du programme awk (s'il n'est pas fourni via l'option '-f'), et les valeurs à rendre disponibles dans les variables awk prédéfinies ARGC et ARGV :

```
awk [options] -f fichier-source [--] fichier . . .
awk [options] [--] 'programme' fichier . . .
```

Les options acceptées par awk sont :

-F fs
: Utilisez fs comme séparateur de champ de saisie (la valeur de la variable prédéfinie FS).

-f fichier programme
: Lisez le code source du programme awk à partir du fichier program-file, au lieu du premier argument de la ligne de commande.

-v var=val
: Attribuez à la variable var la valeur val avant le début de l'exécution du programme.

--
: Signalez la fin des options. Ceci est utile pour permettre la transmission d'arguments supplémentaires au programme awk. Il faut commencer par un tiret (-). Ceci est principalement dû à un souci de cohérence avec l'analyse syntaxique des arguments. conventions de posix.

Toute autre option est signalée comme invalide, mais est par ailleurs ignorée. Voir le chapitre 14 [Invocation] awk], page 105, pour plus de détails.

A.2 Résumé linguistique

Un programme awk est constitué d'une séquence d'instructions de type motif-action et de définitions de fonctions optionnelles.

```
modèle { déclarations d'action }

nom de la fonction (liste des paramètres) { déclarations d'action }
```

La commande awk commence par lire le code source du programme à partir du ou des fichiers programmes spécifiés, ou à partir du premier argument non optionnel de la ligne de commande. L'option « -f » peut être utilisée plusieurs fois.
ligne. awk lit le texte du programme à partir de tous les fichiers de programme, les concaténant en conséquence.
l'ordre dans lequel elles sont spécifiées. Ceci est utile pour créer des bibliothèques de fonctions awk, sans avoir
pour les inclure dans chaque nouveau programme awk qui les utilise. Pour utiliser une fonction de bibliothèque dans un fichier provenant de
Pour exécuter un programme saisi en ligne de commande, spécifiez « -f /dev/tty » ; puis saisissez votre programme et terminez.
Utilisez Ctrl+D. Voir le chapitre 14 [Invocation d'awk], page 105.

awk compile le programme dans un format interne, puis procède à la lecture de chaque fichier nommé dans le tableau ARGV. Si aucun fichier n'est spécifié sur la ligne de commande, awk lit l'entrée standard.

Si un fichier spécifié sur la ligne de commande a la forme « var=val », il est interprété comme une affectation de variable : la variable var se voit attribuer la valeur val. Si l'un des fichiers contient une valeur nulle, cet élément de la liste est ignoré.

Pour chaque ligne de l'entrée, awk vérifie si elle correspond à un motif défini dans le programme awk. Si la ligne correspond à un motif, l'action associée est exécutée.

A.3 Variables et champs

Les variables d'awk sont dynamiques ; elles sont initialisées lors de leur première utilisation. Leurs valeurs sont soit des nombres à virgule flottante, soit des chaînes de caractères. awk gère également les tableaux unidimensionnels ; il est possible de simuler des tableaux multidimensionnels. Plusieurs variables prédéfinies sont initialisées par awk pendant l'exécution du programme ; elles sont résumées ci-dessous.

A.3.1 Champs

Lors de la lecture de chaque ligne d'entrée, awk la divise en champs, en utilisant la valeur de la variable FS comme séparateur. Si FS est un caractère unique, les champs sont séparés par ce caractère. Sinon, FS doit être une expression régulière complète. Dans le cas particulier où FS est un espace unique, les champs sont séparés par des suites d'espaces et/ou de tabulations.

Chaque champ de la ligne de saisie peut être référencé par sa position : $1, $2, etc. $0 représente l'ensemble. ligne. La valeur d'un champ peut également lui être attribuée. Les numéros de champ ne sont pas nécessairement des constantes :

```
n = 5
afficher $n
```

Affiche le cinquième champ de la ligne d'entrée. La variable NF est initialisée avec le nombre total de champs de la ligne d'entrée.

Les références à des champs inexistants (c'est-à-dire les champs après $NF) renvoient la chaîne nulle. Cependant, l'affectation d'une valeur à un champ inexistant (par exemple, $(NF+2) = 5) incrémente la valeur de NF, crée les champs intermédiaires avec la chaîne nulle comme valeur et entraîne le recalcul de la valeur de $0, les champs étant séparés par la valeur de OFS.

Voir le chapitre 3 [Lecture des fichiers d'entrée], page 21, pour une description complète de la façon dont awk définit et utilise les champs.

A.3.2 Variables intégrées

Les variables intégrées d'awk sont :

ARGC Le nombre d'arguments de ligne de commande (sans compter les options ni le programme awk lui-même).

ARGV Tableau des arguments de la ligne de commande. Le tableau est indexé de 0 à ARGC − 1.
La modification dynamique du contenu d'ARGV permet de contrôler les fichiers utilisés pour les données.

CONVFMT Le format de conversion à utiliser lors de la conversion de nombres en chaînes de caractères.

ENVIRON Un tableau contenant les valeurs des variables d'environnement. Le tableau est indexé par
nom de variable, chaque élément étant la valeur de cette variable. Ainsi, l'environnement
La variable HOME se trouverait dans ENVIRON["HOME"]. Sa valeur pourrait être '/u/close'.

Modifier ce tableau n'affecte pas l'environnement vu par les programmes qui utilisent awk.
génère via une redirection ou la fonction système.

Certains systèmes d'exploitation ne possèdent pas de variables d'environnement. Le tableau ENVIRON est
vide lors de l'exécution sur ces systèmes.

NOM_DU_FICHIER Le nom du fichier d'entrée actuel. Si aucun fichier n'est spécifié sur la ligne de commande, le
La valeur de FILENAME est '-'.

FNR Le numéro d'enregistrement d'entrée dans le fichier d'entrée actuel.

FS Le séparateur de champs de saisie, vide par défaut.

NF Le nombre de champs dans l'enregistrement d'entrée actuel.

Non. Nombre total d'enregistrements d'entrée consultés jusqu'à présent.

OFMT Le format de sortie des nombres pour l'instruction d'impression est « %.6g » par défaut.

OFS Le séparateur de champs de sortie, vide par défaut.

ORS Le séparateur d'enregistrements de sortie, par défaut une nouvelle ligne.

RS Le séparateur d'enregistrements d'entrée est par défaut un saut de ligne. RS est exceptionnel en ce sens que seul le
Le premier caractère de sa valeur de chaîne est utilisé pour séparer les enregistrements. Si RS est défini sur null
Si la chaîne est vide, les enregistrements sont séparés par des lignes vides. Lorsque RS est défini sur la chaîne nulle,
Le caractère de nouvelle ligne sert alors toujours de séparateur de champs, en plus de tout autre caractère.
valeur que FS peut avoir.

RSTART L'index du premier caractère correspondant ; 0 si aucune correspondance.

RLENGTH La longueur de la chaîne correspondant à la correspondance ; −1 si aucune correspondance.

SUBSEP La chaîne utilisée pour séparer plusieurs indices dans les éléments du tableau, par défaut "\034".

Voir le chapitre 13 [Variables intégrées], page 101, pour plus d'informations.

A.3.3 Tableaux

Les indices des tableaux sont définis par une expression entre crochets (« [» et «] »). Indices des tableaux
Les nombres sont toujours des chaînes de caractères ; ils sont convertis en chaînes de caractères si nécessaire, selon la conversion standard.
règles (voir section 8.9 [Conversion des chaînes et des nombres], page 67).

Si vous utilisez plusieurs expressions séparées par des virgules à l'intérieur des crochets, alors le tableau
L'indice est une chaîne de caractères constituée de la concaténation des valeurs individuelles de l'indice, converties en
chaînes de caractères, séparées par le séparateur d'indice (la valeur de SUBSEP).

L'opérateur spécial `in` peut être utilisé dans une instruction `if` ou `while` pour vérifier si un tableau possède un index.
composée d'une valeur particulière.

```
si (val dans le tableau)
        afficher tableau[val]
```

Si le tableau comporte plusieurs indices, utilisez (i, j, . . .) dans le tableau pour tester l'existence d'un élément.

La construction « in » peut également être utilisée dans une boucle « for » pour parcourir tous les éléments d'un tableau. Voir la section 10.5 [Analyse de tous les éléments d'un tableau], page 84.

Un élément peut être supprimé d'un tableau à l'aide de l'instruction delete.

Voir le chapitre 10 [Tableaux dans awk], page 81, pour des informations plus détaillées.

A.3.4 Types de données

La valeur d'une expression awk est toujours soit un nombre, soit une chaîne de caractères.

Dans certains contextes (comme les opérateurs arithmétiques), des valeurs numériques sont requises. Ces opérateurs convertissent les chaînes de caractères en nombres en interprétant le texte de la chaîne comme un chiffre. Si la chaîne ne ressemble pas à un chiffre, elle est remplacée par 0.

Dans certains contextes (comme la concaténation), les valeurs de type chaîne de caractères sont requises. La conversion des nombres en chaînes de caractères s'effectue en les affichant avec la fonction `sprintf`. Pour plus de détails, consultez la section 8.9 [Conversion de chaînes de caractères et de nombres], page 67.

Pour forcer la conversion d'une chaîne de caractères en nombre, il suffit d'y ajouter 0. Si la valeur de départ est est déjà un nombre, cela ne le change pas.

Pour forcer la conversion d'une valeur numérique en chaîne de caractères, concaténez-la avec la chaîne nulle.

Le langage awk définit les comparaisons comme étant numériques si les deux opérandes sont numériques, ou si l'un est numérique et l'autre une chaîne de caractères numérique. Sinon, un ou les deux opérandes sont convertis en chaînes de caractères et une comparaison de chaînes est effectuée.

Les variables non initialisées ont la valeur de chaîne "" (la chaîne nulle ou vide). Dans les contextes où Un nombre est requis, ceci équivaut à 0.

Voir la section 8.2 [Variables], page 59, pour plus d'informations sur la dénomination et l'initialisation des variables ; voir la section 8.9 [Conversion des chaînes et des nombres], page 67, pour plus d'informations sur la façon dont les valeurs des variables sont interprétées.

A.4 Modèles et actions

Un programme awk est principalement composé de règles, chacune consistant en un modèle suivi d'une action. L'action est encadrée par des accolades « { » et « } ». Soit le motif est manquant, soit l'action est manquante, mais pas les deux. Si le motif est manquant, l'action est exécutée pour chaque ligne d'entrée. Une action manquante est équivalente à cette action :

```
{ imprimer }
```

qui imprime la ligne entière.

Les commentaires commencent par le caractère « # » et se poursuivent jusqu'à la fin de la ligne. Des lignes vides peuvent séparer les instructions. Normalement, une instruction se termine par un saut de ligne, sauf pour les lignes se terminant par une virgule, une accolade ouvrante (« , », « { », « ? », « : », « && » ou « || »). Les instructions des lignes se terminant par « do » ou « else » sont automatiquement reportées à la ligne suivante. Dans les autres cas, une ligne peut être prolongée en la terminant par une barre oblique inverse (« \ »), auquel cas le saut de ligne est ignoré.

Plusieurs instructions peuvent figurer sur une même ligne en les séparant par un point-virgule (;). Ceci s'applique aux deux les instructions figurant dans la partie action d'une règle (cas habituel), et les instructions de la règle.

Voir la section 2.5 [Commentaires dans les programmes awk], page 18, pour plus d'informations sur les conventions de commentaires d'awk ; voir la section 2.6 [Instructions awk versus lignes], page 19, pour une description de la ligne mécanisme de continuation dans awk.

A.4.1 Motifs

Les modèles awk peuvent être l'un des suivants :

```
/expression régulière/ motif
d'expression relationnelle && motif
motif || motif motif ? motif : motif
(motif) ! motif motif1, motif2 DÉBUT
```

```
FIN
```

Les règles BEGIN et END sont deux types particuliers de modèles qui ne sont pas testés sur les données d'entrée. Les parties « action » de toutes les règles BEGIN sont fusionnées comme si toutes les instructions avaient été écrites dans une seule règle BEGIN. Elles sont exécutées avant toute lecture des données d'entrée. De même, toutes les règles END sont fusionnées et exécutées lorsque toutes les données d'entrée sont traitées (ou lorsqu'une instruction de sortie est exécutée).
Les modèles BEGIN et END ne peuvent pas être combinés avec d'autres modèles dans les expressions de modèles. Les règles BEGIN et END ne peuvent pas comporter d'éléments d'action manquants.

Pour les modèles '/regular-expression/', l'instruction associée est exécutée pour chaque ligne d'entrée correspondant à l'expression régulière. Les expressions régulières sont des extensions de celles d'egrep et sont résumées ci-dessous.

Une expression relationnelle peut utiliser n'importe lequel des opérateurs définis ci-dessous dans la section sur les actions. Ces tests vérifient généralement si certains champs correspondent à certaines expressions régulières.

Les opérateurs '&&', '||' et '!' correspondent respectivement aux opérateurs logiques « et », « ou » et « non », comme en C. Ils effectuent une évaluation en court-circuit, également en C, et servent à combiner des expressions de motifs plus simples. Comme dans la plupart des langages, les parenthèses permettent de modifier l'ordre d'évaluation.

L'opérateur '?:' est similaire à celui du langage C. Si le premier motif correspond, alors le second correspond.
Le modèle est comparé à l'enregistrement d'entrée ; sinon, le troisième est comparé. Un seul des
Les deuxième et troisième motifs correspondent.

La forme « motif1, motif2 » d'un motif est appelée motif de plage. Elle correspond à toutes les lignes d'entrée.
en commençant par une ligne correspondant au motif 1, et en continuant jusqu'à une ligne correspondant au motif 2,
inclusif. Un modèle de plage ne peut pas être utilisé comme opérande pour aucun des opérateurs de modèle.

Voir le chapitre 6 [Modèles], page 47, pour une description complète de la partie modèle des règles awk.

A.4.2 Expressions régulières

Les expressions régulières sont le type étendu que l'on trouve dans egrep. Elles sont composées de caractères comme
suit :

c	correspond au caractère c (en supposant que c soit un caractère sans signification particulière dans les expressions régulières).
\c	correspond au caractère littéral c.
.	correspond à n'importe quel caractère sauf le saut de ligne.
^	correspond au début d'une ligne ou d'une chaîne de caractères.
$	correspond à la fin d'une ligne ou d'une chaîne de caractères.

[abc. . .] correspond à n'importe lequel des caractères abc. . . (classe de caractères).

[^abc. . .] correspond à n'importe quel caractère sauf abc. . . et la nouvelle ligne (classe de caractères négative).

r1|r2	correspond soit à r1 soit à r2 (alternance).
r1r2	correspond à r1, puis à r2 (concaténation).
r+	correspond à un ou plusieurs r.
r*	Correspond à zéro ou plusieurs r.
r?	Correspond à zéro ou un r.
(r)	correspond à r (groupement).

Voir la section 6.2 [Expressions régulières en tant que modèles], page 47, pour une explication plus détaillée de
expressions régulières.

Les séquences d'échappement autorisées dans les constantes de chaîne sont également valides dans les expressions régulières (voir Section 8.1 [Expressions constantes], page 57).

A.4.3 Actions

Les instructions d'action sont encadrées par des accolades, '{' et '}'. Elles se composent des éléments habituels
Les instructions d'affectation, conditionnelles et de boucle sont présentes dans la plupart des langages. Les opérateurs et les contrôles sont également utilisés.
Les instructions et les instructions d'entrée/sortie disponibles sont calquées sur celles du langage C.

A.4.3.1 Opérateurs

Les opérateurs de la bibliothèque awk, par ordre de priorité croissante, sont :

= += -= *= /= %= ^=
: Affectation. L'affectation absolue (var=valeur) et l'affectation par opérateur (l'autre formulaires) sont pris en charge.

?: : Une expression conditionnelle, comme en C. Elle a la forme expr1 ? expr2 : expr3. Si expr1 est vraie, la valeur de l'expression est expr2 ; sinon, c'est expr3. Seule l'une des valeurs expr2 et expr3 est évaluée.

|| : « ou » logique.

&& : « et » logique.

~ !~ : Correspondance d'expression régulière, correspondance négative.

< <= > >= != ==
: Les opérateurs relationnels habituels.

vide : Concaténation de chaînes.

+ - : Addition et soustraction.

* / % : Multiplication, division et modulo.

+ - ! : Plus unaire, moins unaire et négation logique.

^ : L'exponentiation ('**' peut également être utilisé, et '**=' pour l'opérateur d'affectation, mais ils ne sont pas spécifiés dans la norme POSIX).

++ -- : Incrémentation et décrémentation, préfixes et postfixes.

$: Référence de terrain.

Voir le chapitre 8 [Expressions comme énoncés d'action], page 57, pour une description complète de tous les opérateurs énumérés ci-dessus. Voir la section 3.2 [Examen des champs], page 22, pour une description du champ opérateur de référence.

A.4.3.2 Déclarations de contrôle

Les instructions de contrôle sont les suivantes :

```
instruction if (condition) [ instruction else ]
instruction while (condition)
instruction do tant que (condition)
instruction pour (expr1; expr2; expr3)
instruction for (var dans le tableau)
casser
continuer
supprimer le tableau[index]
sortie [ expression ]
{ déclarations }
```

Voir le chapitre 9 [Instructions de contrôle dans les actions], page 73, pour une description complète de toutes les instructions de contrôle. déclarations énumérées ci-dessus.

A.4.3.3 Instructions d'E/S

Les instructions d'entrée/sortie sont les suivantes :

getline Définir $0 à partir du prochain enregistrement d'entrée ; définir NF, NR, FNR.

getline <file Définir
$0 à partir du prochain enregistrement du fichier ; définir NF.

getline var Définir
var à partir de l'enregistrement d'entrée suivant ; définir NF, FNR.

getline var <file Définir
var à partir de l'enregistrement suivant du fichier.

suivant Le traitement de l'enregistrement d'entrée actuel est interrompu. L'enregistrement suivant est lu et le traitement reprend avec le premier motif du programme awk. Si la fin des données d'entrée est atteinte, la ou les règles END, le cas échéant, sont exécutées.

imprimer Affiche l'enregistrement actuel.

Afficher la liste des
expressions. Afficher la

liste des expressions > fichier
Imprime les expressions dans un fichier.

printf fmt, expr-list Format et
impression.

printf fmt, liste d'expressions > fichier
Format et impression dans le fichier.

D'autres redirections d'entrée/sortie sont également autorisées. Pour `print` et `printf`, `>> fichier` ajoute la sortie au fichier, et `| commande` écrit dans un tube. De même, `commande | getline` redirige l'entrée vers `getline`. `getline` renvoie 0 à la fin du fichier, et -1 en cas d'erreur.

Voir la section 3.7 [Saisie explicite avec getline], page 30, pour une description complète de l'instruction getline. Voir le chapitre 4 [Affichage de la sortie], page 35, pour une description complète des instructions print et printf.
Enfin, voir la section 9.7 [La déclaration suivante], page 78, pour une description de la manière dont la déclaration suivante travaux.

Résumé A.4.3.4 printf

Les instructions printf et sprintf de la commande awk acceptent les formats de spécification de conversion suivants :

%c Un caractère ASCII. Si l'argument utilisé pour '%c' est numérique, il est traité comme un caractère et affiché. Sinon, l'argument est considéré comme une chaîne de caractères, et seul le premier caractère de cette chaîne est affiché.

%d
%i Un nombre décimal (la partie entière).

%et Un nombre à virgule flottante de la forme '[-]d.dddddE[+-]dd'.

%f Un nombre à virgule flottante de la forme [-]ddd.dddddd.

%g	Utilisez la conversion '%e' ou '%f', celle qui produit la chaîne la plus courte, avec les caractères non significatifs Zéros supprimés.
%le	Un nombre octal non signé (encore une fois, un entier).
%s	Une chaîne de caractères.
%x	Un nombre hexadécimal non signé (un entier).
%X	Comme '%x', mais en utilisant 'A' à 'F' au lieu de 'a' à 'f' pour les nombres décimaux de 10 à 10. 15.
%%	Un seul caractère '%' ; aucun argument n'est converti.

Il existe des paramètres supplémentaires facultatifs qui peuvent se trouver entre le symbole « % » et la lettre de contrôle :

-	L'expression doit être alignée à gauche dans son champ.
largeur	Le champ doit être complété à cette largeur. Si la largeur commence par un zéro, alors le champ est Complété par des zéros. Sinon, il est complété par des espaces.
.préc	Un nombre indiquant la largeur maximale des chaînes ou des chiffres à droite de la virgule décimale indiquer.

L'une ou l'autre des valeurs de largeur et/ou de précision peuvent être spécifiées comme '*'. Dans ce cas, le La valeur est extraite de la liste des arguments.

Voir la section 4.5 [Utilisation des instructions printf pour une impression plus sophistiquée], page 38, pour des exemples et pour plus de détails. une description plus détaillée.

A.4.3.5 Fonctions numériques

awk possède les fonctions arithmétiques prédéfinies suivantes :

atan2(y, x)	renvoie l'arctangente de y/x en radians.
cos(expr)	renvoie le cosinus en radians.
exp(expr)	la fonction exponentielle.
int(expr)	tronque en entier.
log(expr)	la fonction logarithme naturel.
rand()	renvoie un nombre aléatoire compris entre 0 et 1.
sin(expr)	renvoie le sinus en radians.
racine carrée(expr)	la fonction racine carrée.
srand(expr)	Utilisez expr comme nouvelle graine pour le générateur de nombres aléatoires. Si aucune expression n'est fournie, L'heure de la journée est utilisée. La valeur de retour est la graine précédente du nombre aléatoire. générateur.

A.4.3.6 Fonctions de chaînes de caractères

awk possède les fonctions de chaînes prédéfinies suivantes :

La fonction `gsub(r, s, t)` remplace chaque sous-chaîne correspondant à l'expression régulière `r` dans la chaîne `t` par la chaîne `s` et renvoie le nombre de substitutions. Si `t` n'est pas fourni, la fonction renvoie 0. La fonction `index(s, t)` renvoie l'indice de la chaîne `t` dans la chaîne `s`, ou 0 si `t` est absent.

longueur(s)
: Renvoie la longueur de la chaîne s. La longueur de $0 est renvoyée si aucun argument n'est fourni.

match(s, r)
: renvoie la position dans s où l'expression régulière r apparaît, ou 0 si r n'est pas présent, et définit les valeurs de RSTART et RLENGTH.

La fonction `split(s, a, r)` divise la chaîne `s` en éléments du tableau `a` selon l'expression régulière `r`, et renvoie le nombre de champs. Si `r` est omis, `FS` est utilisé à sa place. La fonction `sprintf(fmt, expr-list)` affiche `expr-list` selon `fmt`, et renvoie la chaîne résultante.

La fonction `sub(r, s, t)` est similaire à `gsub`, mais ne remplace que la première sous-chaîne correspondante. La fonction `substr(s, i, n)` renvoie la sous-chaîne de n caractères de `s` commençant à l'indice `i`. Si `n` est omis, le reste de `s` est conservé.

La fonction `tolower(str)` renvoie une copie de la chaîne `str`, où toutes les majuscules sont converties en minuscules. Les caractères non alphabétiques restent inchangés.

La fonction toupper(str) renvoie une copie de la chaîne str, où toutes les lettres minuscules sont converties en majuscules. Les caractères non alphabétiques restent inchangés.

système(ligne de commande)
: Exécutez la commande cmd-line et renvoyez le code de sortie.

A.4.3.7 Constantes de chaîne

Dans awk, les constantes de chaîne sont des séquences de caractères encadrées par des guillemets doubles ("). À l'intérieur des chaînes, certaines séquences d'échappement sont reconnues, comme en C. Ce sont :

\\ Une barre oblique inverse, au sens propre.

\un Le caractère « alerte » ; généralement le caractère ASCII BEL.

b Retour arrière.

\f Alimentation du formulaire.

\n Nouvelle ligne.

\r Retour du wagon.

\t Onglet horizontal.

\v Onglet vertical.

\x chiffres hexadécimaux : Le caractère représenté par la chaîne de chiffres hexadécimaux suivant le '\x'. Comme en C ANSI, tous les chiffres hexadécimaux suivants sont considérés comme faisant partie de la séquence d'échappement. (Cette caractéristique devrait nous renseigner sur la conception du langage par un comité.) Par exemple, « \x1B » est une chaîne contenant le caractère d'échappement ASCII ESC. (La séquence d'échappement « \x » n'est pas présente dans awk POSIX.)

\ddd Le caractère représenté par une séquence de 1, 2 ou 3 chiffres octaux. Ainsi, « \033 » est également une chaîne contenant le caractère d'échappement ASCII ESC.

\c Le caractère littéral c.

Les séquences d'échappement peuvent également être utilisées à l'intérieur d'expressions régulières constantes (par exemple, l'expression régulière /[\t\f\n\r\v]/ correspond aux caractères d'espacement).

Voir la section 8.1 [Expressions constantes], page 57.

A.5 Fonctions

Les fonctions d'awk sont définies comme suit :

```
nom_de_fonction(liste_de_paramètres) { instructions }
```

Les paramètres effectifs fournis lors de l'appel de fonction sont utilisés pour instancier les paramètres formels. déclaré dans la fonction. Les tableaux sont passés par référence, les autres variables par valeur.

S'il y a moins d'arguments passés que de noms dans la liste des paramètres, les noms supplémentaires sont La valeur est une chaîne vide. Les noms supplémentaires ont le rôle de variables locales.

Dans l'appel d'une fonction définie par l'utilisateur, la parenthèse ouvrante doit suivre immédiatement le nom de la fonction, sans espace. Ceci afin d'éviter toute ambiguïté syntaxique avec l'opérateur de concaténation.

Le mot func peut être utilisé à la place de function (mais pas dans posix awk).

Utilisez l'instruction return pour renvoyer une valeur depuis une fonction.

Voir le chapitre 12 [Fonctions définies par l'utilisateur], page 95, pour une description plus complète.

Annexe B Exemple de programme

L'exemple suivant est un programme awk complet qui affiche le nombre d'occurrences de chaque mot dans son entrée. Il illustre la nature associative des tableaux awk en utilisant des chaînes de caractères comme indices. Il présente également la construction « for x in array ». Enfin, il montre comment awk peut être utilisé conjointement avec d'autres utilitaires pour accomplir une tâche utile, même complexe, avec un minimum d'effort. Des explications suivent le listing du programme.

```
maladroit '
# Afficher la liste des fréquences des mots {
        pour (i = 1 ; i <= NF ; i++) freq[$i]++
}
FIN
    { pour (mot dans freq) printf
            "%s\t%d\n", mot, freq[mot]
}'
```

La première chose à remarquer concernant ce programme est qu'il comporte deux règles. La première, ayant un motif vide, est exécutée sur chaque ligne de l'entrée. Elle utilise le mécanisme d'accès aux champs d'awk (voir la section 3.2 [Examen des champs], page 22) pour extraire les mots individuels de la ligne, et la variable intégrée NF (voir le chapitre 13 [Variables intégrées], page 101) pour déterminer le nombre de champs disponibles.

Pour chaque mot saisi, un élément du tableau freq est incrémenté pour indiquer que le mot a été saisi.
a été vu une fois de plus.

La deuxième règle, puisqu'elle contient le motif END, n'est exécutée que lorsque l'entrée est épuisée. Elle affiche le contenu du tableau de fréquences construit lors de la première action.

Notez que ce programme présente plusieurs problèmes qui l'empêcheraient d'être utile seul sur de vrais fichiers texte :

Les mots sont détectés selon la convention awk, qui considère que les champs sont séparés par des espaces et que les autres caractères de l'entrée (à l'exception des sauts de ligne) n'ont aucune signification particulière pour awk. Cela signifie que les signes de ponctuation sont comptabilisés comme faisant partie des mots.

Le langage awk fait la distinction entre majuscules et minuscules. Par conséquent, « foo » et « Foo » ne sont pas considérés comme un seul et même mot. Ceci est problématique car, dans un texte courant, les mots en début de phrase commencent par une majuscule, et un analyseur de fréquence ne devrait pas être sensible à cette différence.

Les résultats ne sont pas présentés dans un ordre pertinent. Vous serez probablement plus intéressé par les mots les plus fréquents, ou par un tableau alphabétique de la fréquence d'apparition de chaque mot.

La solution à ces problèmes consiste à utiliser certaines des fonctionnalités les plus avancées du langage awk.
Tout d'abord, nous utilisons la fonction `tolower` pour supprimer la casse. Ensuite, nous utilisons la fonction `gsub` pour supprimer la ponctuation.

caractères. Enfin, nous utilisons l'utilitaire de tri du système pour traiter la sortie du script awk. Voici d'abord la nouvelle version du programme :

```
maladroit '
# Afficher la liste des fréquences des mots {

        $0 = tolower($0) gsub(/          # supprimer les distinctions de casse
        [^a-z0-9_ \t]/, "", $0) # supprimer la ponctuation for (i = 1; i <= NF; i++) freq[$i]+
        +

}

FIN
      { pour (mot dans freq) printf
              "%s\t%d\n", mot, freq[mot]
}'
```

En supposant que nous ayons enregistré ce programme dans un fichier nommé « frequency.awk », et que les données se trouvent dans 'file1', le pipeline suivant

```
awk -f frequency.awk fichier1 | sort +1 -nr
```

produit un tableau des mots apparaissant dans 'file1' par ordre de fréquence décroissante.

Le programme awk traite les données de manière appropriée et produit un tableau de fréquence des mots, qui n'est pas ordonné.

La sortie du script awk est ensuite triée par la commande `sort` et affichée dans le terminal. Les options de `sort` dans cet exemple spécifient que le tri s'effectue en utilisant le deuxième champ de chaque ligne d'entrée (en ignorant un champ), que les clés de tri doivent être traitées comme des nombres (sinon « 15 » précéderait « 5 ») et que le tri doit être effectué par ordre décroissant (inverse).

Nous aurions même pu effectuer le tri directement dans le programme, en modifiant l'action FIN comme suit :

```
FIN {
        trier = "tri +1 -nr"
        pour (mot dans freq) printf
              "%s\t%d\n", mot, freq[mot] | trier
        fermer(trier)
}'
```

Consultez la documentation générale du système d'exploitation pour plus d'informations sur l'utilisation de la commande de tri.

Annexe C Glossaire

Une action est une série d'instructions awk associées à une règle. Si le modèle de la règle correspond à un enregistrement d'entrée, le langage awk exécute l'action de la règle. Les actions sont toujours placées entre accolades. Voir le chapitre 7 [Présentation des actions], page 55.

Henry Spencer, de l'Université de Toronto, a écrit un assembleur awk remarquable, entièrement composé de scripts awk et entièrement adaptable. Ce programme, long de plusieurs milliers de lignes, inclut des descriptions de machines pour plusieurs micro-ordinateurs 8 bits. Il illustre parfaitement le type de programme qui aurait sans doute été mieux écrit dans un autre langage.

ansi L'American National Standards Institute (ANSI). Cet organisme élabore de nombreuses normes, parmi lesquelles celle du langage de programmation C.

Une affectation est une expression awk qui modifie la valeur d'une variable ou d'un objet de données awk. Un objet auquel on peut affecter une valeur est appelé une lvalue. Voir la section 8.7 [Expressions d'affectation], page 64.

Langage awk Le langage dans lequel les programmes awk sont écrits.

Programme awk Un programme awk est constitué d'une série de modèles et d'actions, collectivement appelés règles. Pour chaque enregistrement d'entrée fourni au programme, les règles de ce dernier sont toutes traitées successivement. Les programmes awk peuvent également contenir des définitions de fonctions. Un script awk est un autre nom pour un programme awk.

Fonctions intégrées : Le langage awk propose des fonctions intégrées permettant d'effectuer divers calculs numériques, temporels et de chaînes de caractères. On peut citer par exemple sqrt (pour la racine carrée d'un nombre) et substr (pour extraire une sous-chaîne d'une chaîne). Voir le chapitre 11 [Fonctions intégrées], page 89.

Les variables intégrées ARGC, ARGV, CONVFMT, ENVIRON, FILENAME, FNR, FS, NF, NR, OFMT, OFS, ORS, RLENGTH, RSTART, RS et SUBSEP ont une signification particulière pour awk. Leur modification affecte l'environnement d'exécution d'awk. Voir le chapitre 13 [Variables intégrées], page 101.

Croisillons Voir « Appareils dentaires bouclés ».

C Le langage de programmation système utilisé par la plupart des logiciels GNU. Le langage de programmation awk possède une syntaxe similaire à celle du C, et ce manuel souligne les similitudes entre awk et C le cas échéant.

CHEM est un préprocesseur pour pic qui lit les descriptions de molécules et produit des données d'entrée pic pour leur représentation. Il a été écrit par Brian Kernighan et est disponible à l'adresse netlib@research.att.com.

Instruction composée : Une série d'instructions awk, encadrées par des accolades. Les instructions composées peuvent être imbriquées. Voir le chapitre 9 [Instructions de contrôle dans les actions], page 73.

Enchaînement Concaténer deux chaînes de caractères consiste à les juxtaposer pour former une nouvelle chaîne. Par exemple, la chaîne « foo » concaténée avec la chaîne « bar » donne la chaîne « foobar ». Voir la section 8.4 [Concaténation de chaînes], page 61.

Expression conditionnelle : Une
expression utilisant l'opérateur ternaire « ?: », telle que expr1 ? expr2 : expr3. L'expression expr1 est évaluée ; si le résultat est vrai, la valeur de l'expression est celle de expr2 ; sinon, elle est celle de expr3. Dans tous les cas, seule l'une des expressions expr2 ou expr3 est évaluée. Voir la section 8.11 [Expressions conditionnelles], page 69.

Expression régulière constante
Une expression régulière constante est une expression régulière écrite entre barres obliques, comme '/foo/'. Cette expression régulière est choisie lors de l'écriture du programme awk et ne peut être modifiée pendant son exécution. Voir la section 6.2.1 [Utilisation des expressions régulières], page 47.

Expression de comparaison :
Une relation qui vaut soit vrai, soit faux, comme (a < b). Les expressions de comparaison sont utilisées dans les instructions if, while et for, ainsi que dans les modèles pour sélectionner les enregistrements d'entrée à traiter. Voir la section 8.5 [Expressions de comparaison], page 62.

Les accolades :
les caractères « { » et « } ». Les accolades sont utilisées dans awk pour délimiter les actions, les instructions composées et les corps de fonctions.

Objets de
données : il s'agit de nombres et de chaînes de caractères. Les nombres sont convertis en chaînes et inversement, selon les besoins. Voir la section 8.9 [Conversion des chaînes et des nombres], page 67.

Expression régulière dynamique
Une expression régulière dynamique est une expression régulière écrite sous forme d'expression ordinaire.
Il peut s'agir d'une constante de type chaîne de caractères, comme « foo », mais aussi d'une expression dont la valeur peut varier. Voir la section 6.2.1 [Utilisation des expressions régulières], page 47.

Séquences
d'échappement : séquence spéciale de caractères utilisée pour décrire les caractères non imprimables, comme « \n » pour le saut de ligne ou « \033 » pour le caractère d'échappement ASCII ESC. Voir la section 8.1 [Expressions constantes], page 57.

Champ
Lorsque awk lit un enregistrement d'entrée, il divise l'enregistrement en morceaux séparés par des espaces (ou par une expression régulière de séparation que vous pouvez modifier en définissant la variable intégrée FS).
Ces éléments sont appelés champs. Voir la section 3.1 [Comment les données d'entrée sont divisées en enregistrements], page 21.

Format
Les chaînes de format sont utilisées pour contrôler l'apparence de la sortie dans l'instruction printf.
De plus, la conversion des données numériques en chaînes de caractères est contrôlée par la chaîne de format contenue dans la variable intégrée CONVFMT. Voir la section 4.5.2 [Lettres de contrôle de format], page 38.

Une fonction est un ensemble d'instructions spécialisé, souvent utilisé pour encapsuler des tâches générales ou spécifiques à un programme. awk possède plusieurs fonctions intégrées et permet également de définir les siennes. Voir le chapitre 11 [Fonctions intégrées], page 89, et le chapitre 12 [Fonctions définies par l'utilisateur], page 95.

rester bouche bée
L'implémentation GNU de awk.

GNOU
« GNU n'est pas Unix ». Un projet en cours de la Free Software Foundation visant à créer un environnement informatique complet, librement distribuable et conforme à la norme POSIX.

Enregistrement
d'entrée : Un bloc de données unique lu par awk. Généralement, un enregistrement d'entrée awk se compose d'une seule ligne de texte. Voir la section 3.1 [Comment l'entrée est divisée en enregistrements], page 21.

Dans le langage awk, un mot-clé est un terme ayant une signification particulière. Les mots-clés sont réservés et ne peuvent pas être utilisés comme noms de variables. Les mots-clés d'awk sont : if,
else, while, do...while, for, for...in, break, continue, delete, next, function, func et exit.

Lvalue : Expression pouvant figurer à gauche d'un opérateur d'affectation. Dans la plupart des langages, les lvalues peuvent être des variables ou des éléments de tableau. En awk, un désignateur de champ peut également servir de lvalue.

Nombre : Un objet de données à valeur numérique. L'implémentation awk utilise des nombres à virgule flottante double précision pour représenter les nombres.

Modèle Les modèles indiquent à awk quels enregistrements d'entrée sont pertinents pour quelles règles.

Un motif est une expression conditionnelle arbitraire par rapport à laquelle une entrée est testée. Si la condition est satisfaite, on dit que le motif correspond à l'enregistrement d'entrée. Un motif typique peut comparer l'enregistrement d'entrée à une expression régulière. Voir le chapitre 6 [Motifs], page 47.

posix Il s'agit du nom d'une série de normes développées par l'IEEE qui spécifient une interface pour système d'exploitation portable. « IX » fait référence à l'héritage Unix de ces normes.
La principale norme qui intéresse les utilisateurs d'awk est P1003.2, la norme relative au langage de commande et aux utilitaires.

Plage (de lignes d'entrée)
Une séquence de lignes consécutives extraites du fichier d'entrée. Un motif peut spécifier des plages de lignes d'entrée à traiter par awk, ou des lignes individuelles. Voir le chapitre 6 [Modèles], page 47.

Récursivité : lorsqu'une fonction s'appelle elle-même, directement ou indirectement. Si cela n'est pas clair, reportez-vous à la documentation. entrée pour « récursivité ».

La redirection
consiste à effectuer une entrée à partir d'un flux autre que le flux d'entrée standard, ou une sortie vers un flux autre que le flux de sortie standard.

Vous pouvez rediriger la sortie des instructions `print` et `printf` vers un fichier ou une commande système à l'aide des opérateurs `>`, `>>` et `|`. Vous pouvez rediriger l'entrée vers l'instruction `getline` à l'aide des opérateurs `<` et `|`. Voir la section 4.6 [Redirection de la sortie de `print` et `printf`], page 42.

Expression régulière Voir
« regexp ».

Expression régulière Abréviation d'expression régulière. Une expression régulière est un motif qui désigne un ensemble de chaînes de caractères, potentiellement infini. Par exemple, l'expression régulière 'R.*xp' correspond à toute chaîne commençant par la lettre 'R' et se terminant par les lettres 'xp'. En awk, les expressions régulières sont utilisées dans les motifs et les expressions conditionnelles. Elles peuvent contenir des séquences d'échappement. Voir la section 6.2 [Expressions régulières comme motifs], page 47.

Règle Une règle est un segment d'un programme awk qui spécifie comment traiter des enregistrements d'entrée individuels. Elle se compose d'un motif et d'une action. awk lit un enregistrement d'entrée ; ensuite, pour chaque règle, si l'enregistrement correspond au motif de la règle, awk exécute l'action correspondante. Sinon, la règle ne fait rien pour cet enregistrement.

Effet secondaire

Un effet de bord se produit lorsqu'une expression a un effet autre que celui de simplement produire une valeur. Les expressions d'affectation, les expressions d'incrémentation et les appels de fonction ont des effets de bord. Voir la section 8.7 [Expressions d'affectation], page 64.

Nom de fichier
spécial : nom de fichier interprété en interne par awk, au lieu d'être transmis directement au système d'exploitation sous-jacent. Par exemple : « /dev/stdin ». Voir la section 4.7 [Flux d'E/S standard], page 44.

Éditeur de flux

Un programme qui lit des enregistrements à partir d'un flux d'entrée et les traite un ou plusieurs à la fois. Ceci contraste avec les programmes par lots, qui peuvent s'attendre à lire leur entrée

les fichiers dans leur intégralité avant de commencer quoi que ce soit, et avec les programmes interactifs, qui nécessitent l'intervention de l'utilisateur.

Une chaîne de caractères est une donnée constituée d'une séquence de caractères, comme « Je suis une chaîne de caractères ». Les chaînes constantes sont écrites entre guillemets doubles dans le langage awk et peuvent contenir des séquences d'échappement. Voir la section 8.1 [Expressions constantes], page 57.

Espace blanc :

une séquence de caractères vides ou de tabulations apparaissant à l'intérieur d'un enregistrement d'entrée ou d'une chaîne de caractères.

Indice

#

'#' 18
'#!' 18

$

$ (opérateur de terrain) 22

-

Option « -f » . . . . . 17 Option « -f » . . . . . 105 Option « -F » . . . . . Option 25 -F . . . . . 105 Option '-v' 105

UN

accès aux champs 22
acronyme 1
action, accolades . . . . . 55 action, valeur par défaut . . . . . 13 action, définition de . . . . . 55 action, séparation des énoncés 55
addition 60
et opérateur 64
applications de awk 20
ARGC 102
Arguments de l'appel de fonction. . . . . 70 arguments, ligne de commande 105
ARGV 102, 105
opérateurs arithmétiques 60
Affectation de tableau 83
Référence de tableau 82
Tableaux 81
tableaux, définition de . . . . . 81 tableaux, suppression d'un élément . . . . . 85 tableaux, indices multidimensionnels . . . . . 86 tableaux, présence d'éléments . . . . . 82 tableaux, instruction for spéciale . . . . . 84 opérateurs d'affectation . . . . . 64 affectations à des champs . . . . . 24 tableaux associatifs . . . . . 81 langage maladroit. . . . . 11 programmes maladroits. 11

B

Suite de la barre oblique inverse 19
Fonction de base d'awk . . . . . 13 Fichier « liste BBS » . . . . . 11 Motif spécial BEGIN . . . . . 53 Corps d'une boucle 74
Expressions booléennes 64
opérateurs booléens . . . . . 64 motifs booléens 51
instruction break 76
Mise en mémoire tampon de la sortie . . . . . 94 tampons, vidage 94
fonctions intégrées. . . . . 89 variables intégrées . . . . . 101 variables intégrées, modifiables par l'utilisateur 101

Appel par référence en C. 97
Appel par valeur . . . . . 97 Appel d'une fonction . . . . . 70 Sensibilité à la casse . . . . . 17 Modification du contenu d'un champ. . . . . 24 Modification du séparateur d'enregistrements 21
Fermeture . . . . . 33, 43 Fermeture des fichiers et tubes d'entrée . . . . . 33 Fermeture des fichiers et tubes de sortie 43
Ligne de commande 105
Formats de ligne de commande. 16
ligne de commande, activation du système de fichiers. . . . . 26 commentaires 18
Expressions de comparaison 62
Expressions de comparaison sous forme de motifs 51
Expressions régulières calculées 48
Concaténation 61
expression conditionnelle . . . . . 69 constantes, types de . . . . . 57 continuation de lignes. 19
suite de la déclaration 77
Déclaration de contrôle 73
Conversion de chaînes et de nombres . . . . . 67, 68 conversions, lors de l'indexation 86
CONVFMT . . . . . 62, 67, 86, 101 accolades 55

D

action par défaut 13
Modèle par défaut. 13
Définition des fonctions . ...

Suppression d'éléments de tableaux 85
différences entre gawk et awk . . . . . 57, 61 différences : gawk et awk. 44
division . . . . . 60 Documentation des programmes awk 18
Expressions régulières dynamiques 48

ET

Affectation d'élément . . . . . 83 Élément du
tableau . . . . . 82 Motif
vide . . . . . 54 FIN du motif
spécial. 53
ENVIRONNEMENT 102
Notation des séquences d'échappement 57
Examen des champs 22
Scripts exécutables 18
instruction de sortie 79
Entrée explicite 30
Exponentiation 60
Expression 57
expressions conditionnelles . . . . . 69 expressions
d'affectation . . . . . 64 expressions
booléennes . . . . . 64 expressions de
comparaison 62

F

Séparateur de champs, choix de... 26
Séparateur de champs, FS 25
Séparateur de champs : sur la ligne de commande 26
Champ, modification du contenu de 24
Champs 22
champs, séparant . . . . . 25 descripteurs
de fichiers . . . . . 44 fichier,
programme awk . . . . . 17 NOM DE
FICHIER . . . . . 21, 102 vidage
des tampons. . . . . .
94 FNR. . . . . 22, 103 pour
(x dans . . .) 84
pour l'énoncé 75
Spécificateur de format 38
chaîne de format. 38
sortie formatée 38
FS . . . . . 25, 101 appel
de fonction 70
Définition de fonction . . . . . 95 fonctions
définies par l'utilisateur 95

G

getline . . . . . 30 gsub . ...
93

H

Historique d'awk 1
Fonctionnement d'awk 14

je

instruction if 73
Opérateurs d'incrémentation 66
entrée 21
fichier d'entrée, échantillon 11
Redirection d'entrée . . . . . 31 entrée,
explicite . . . . . 30 entrée,
commande getline. . . . . 30 entrée, enregistrements
sur plusieurs lignes . . . . . 29 Entrée,
standard . . . . . 16 Interaction, awk et
autres programmes . . . . . 93 Fichier « inventory-
shipped » . . . . . 12 Appel de
awk 105

L

Langage, awk 11
longueur. 90
opérations logiques 64
options longues 105
boucles . . . . . .
74 boucles, sortie 76
lvalue 65

M

manuel, en utilisant ceci 11
correspondance 90, 91
métacaractères 48
Modificateurs (dans les spécificateurs de format) 39
indices multidimensionnels . . . . . 86 enregistrements
sur plusieurs lignes . . . . . 29 passages
multiples sur les données. . . . . 106 instructions
multiples sur une seule ligne. 20
multiplication 60

N

Déclaration suivante 78
NF . . . . . 22, 103 non
opérateur 64
NR . . . . . 22, 103 nombre
de champs, NF . . . . . 22 nombre
d'enregistrements, NR ou FNR . . . . . 22 nombres,
utilisés comme indices . . . . . 86 constante
numérique 57
valeur numérique 57

LE

OFMT 37, 68, 101
OFS. 37, 101
blagues courtes 45
Priorité des opérateurs . . . . . 71 opérateurs,
$ . . . . . 22 opérateurs
arithmétiques . . . . . 60 opérateurs
d'affectation . . . . . 64 opérateurs
booléens . . . . . 64 opérateurs
d'incrémentation . . . . . 66 opérateurs
de correspondance d'expressions régulières. . . . . .
47 opérateurs relationnels . . . . . 51, 62 opérateurs,
chaîne 61

opérateurs, correspondance de chaînes 47
options, ligne de commande . . . . . 105 options,
long . . . . . 105 opérateur
ou 64
ORS. 37, 101
sortie 35
séparateur de champ de sortie, OFS 37
Séparateur d'enregistrements de sortie, ORS . . . . . .
37 Redirection de sortie. . . . . 42 Mise
en mémoire tampon de la sortie . . . . . .
94 Sortie formatée . . . . . 38 sortie,
tuyauterie 42

P

passes, multiples . . . . . 106 motif,
sensible à la casse . . . . . 17 motif,
expressions de comparaison . . . . . 51 motif, par
défaut . . . . . 13 motif, définition
de . . . . . 47 motif,
vide . . . . . 54 motif, expressions
régulières . . . . . 47 motifs,
DÉBUT . . . . . 53 motifs,
booléen . . . . . 51 motifs,
FIN. . . . . 53 motifs,
plage. . . . . 53 modèles, types
de . . . . . 47 tuyaux de
sortie 42
priorité . . . . . 71 'print
$0' . . . . . 13 instruction
print . . . . . 35 instruction printf,
syntaxe de. . . . . 38 printf, caractères de contrôle de
format . . . . . 38 printf,
modificateurs 39
impression 35
fichier programme. 17
programme, awk 11
programme, définition de . . . . . 13 programme,
autonome . . . . . 18 programmes,
documentant 18

Q

quotient 60

R

Modèle de plage 53
lectures de fichiers 21
lectures de fichiers, commande getline . . . . . 30 lectures
de fichiers, enregistrements sur plusieurs lignes 29
Séparateur d'enregistrements . . . . . .
21 enregistrements, plusieurs lignes . . . . . .
29 Redirection de l'entrée 31
Redirection de la sortie . . . . . 42 référence
au tableau 82
regexp . . . . . 47 regexp
comme expression . . . . . 63 opérateurs
regexp . . . . . 62 opérateurs
de recherche regexp . . . . . 47 opérateurs de
correspondance d'expressions régulières . . . . . 47 métacaractères
d'expressions régulières . . . . . 48 expressions régulières
comme séparateurs de champs . . . . . 26 expressions régulières
comme motifs . . . . . 47 expressions régulières
calculées . . . . . 48 opérateurs
relationnels 51, 62
reste . . . . . 60 suppression
d'éléments de tableaux . . . . . 85 instruction
return 98
RLENGTH 91, 103
RS 21,
101 RSTART . . . . . 91, 103 règle,
définition de. . . . . 13 exécution de
programmes awk . . . . . 16 exécution de
programmes longs 17

S

Exemple de fichier d'entrée 11
Analyse d'un tableau 84
Script, définition de. . . . . 13 Scripts,
exécutable . . . . . 18 scripts
shell . . . . . 18 programmes
autonomes . . . . . 18 scripts
shell . . . . . 18 effet
secondaire 65
guillemets simples, pourquoi sont-ils nécessaires . . . . . .
16 division . . . . . .
91 sprintf 92
sortie d'erreur standard . . . . . 44 entrée
standard . . . . . 16, 21, 44 sortie
standard . . . . . 44 constantes de
chaînes . . . . . 57 opérateurs de
chaînes . . . . . 61 opérateurs
de correspondance de chaînes 47
sous 92
Indices dans les tableaux 86
SUBSEP 86, 101
substr. 93
soustraction 60
système. 93

T

. 93
toupper 93

DANS

Utilisation des commentaires : 18 fonctions définies par l'utilisateur :
95 variables définies par l'utilisateur : 59

Utilisation de awk 1
en utilisant ce manuel 11

Variables V , définies par l'utilisateur 59

DANS

Qu'est-ce que l'awk ? 1
Quand utiliser awk ? 20
instruction while 74

Sommaire

Préface. 1
Licence publique générale GNU 3
1 Utilisation de ce manuel . . . . . 11 2 Premiers pas avec awk . . . . . 13 3 Lecture des fichiers d'entrée . . . . . 21 4 Impression de la sortie . . . . . 35 5 répliques percutantes 45
6 Motifs. 47
7 Aperçu des actions. 55
8 Expressions en tant qu'instructions d'action . . . . . 57 9 Instructions de contrôle dans les actions. 73
10 Tableaux dans awk. . . . . 81 11 Fonctions intégrées 89
12 Fonctions définies par l'utilisateur 95
13 Variables intégrées. 101
14. Utilisation d' awk . . . . . 105. Annexe A : Résumé d'awk . . . . . 107. Annexe B : Exemple de programme . . . . . 119. Annexe C : Glossaire 121
Index 125

Table des matières

Préface . . . . . 1 Historique
d'awk. 1

Licence publique générale GNU 3
Préambule . ...

MODIFICATION 4
Comment appliquer ces termes à vos nouveaux programmes 8

1 Utilisation de ce manuel . . . . . 11 1.1 Fichiers de
données pour les exemples 11

2 Premiers pas avec awk . . . . . 13 2.1 Un exemple très
simple . . . . . 13 2.2 Un exemple avec deux
règles . . . . . 14 2.3 Un exemple plus
complexe . . . . . 15 2.4 Comment exécuter des
programmes awk . . . . . 16 2.4.1 Programmes awk
éphémères . . . . . 16 2.4.2 Exécution d'awk sans fichier
d'entrée. . . . . 16 2.4.3 Exécution de programmes
longs. . . . . 17 2.4.4 Programmes awk
exécutables. . . . . 18 2.5 Commentaires dans les
programmes awk . . . . . 18 2.6 Instructions awk versus
lignes . . . . . 19 2.7 Quand utiliser
awk 20

3 Lecture des fichiers d'entrée . . . . . 21 3.1 Comment les données d'entrée
sont divisées en enregistrements. . . . . 21 3.2 Examen des
champs. ... 22 3.3 Numéros de champs non constants . . . . . 23 3.4 Modification du contenu d'un champ 24 3.5
Spécification de la séparation des champs 25 3.6
Enregistrements multilignes . . . . . 29 3.7 Entrée
explicite avec getline. . . . . 30 3.8 Fermeture des
fichiers d'entrée et des tubes 33

4. Sortie d'impression . . . . . 35 4.1 L'instruction
d'impression. 35 4.2
Exemples d'instructions d'impression . . . . . 35 4.3 Séparateurs de sortie. 37
4.4 Contrôle de l'affichage numérique avec print . . . . . 37 4.5 Utilisation
des instructions printf pour un affichage plus sophistiqué 38 4.5.1
Introduction à l'instruction printf . . . . . 38 4.5.2 Lettres de contrôle de format. . . . . 38 4.5.3 Modificateurs pour les formats
printf. . . . . 39 4.5.4 Exemples d'utilisation de
printf . . . . . 40 4.6 Redirection de la sortie de print et
printf. . . . . 42 4.6.1 Redirection de la sortie vers des fichiers et des
tubes. . . . . 42 4.6.2 Fermeture des fichiers et tubes de
sortie . . . . . 43 4.7 Flux d'E/S
standard 44

5 répliques percutantes 45

6 Motifs 47

6.1 Types de motifs. 47 6.2
Expressions régulières en tant que motifs 47 6.2.1
Comment utiliser les expressions régulières 47 6.2.2
Opérateurs d'expressions régulières. . . . . 48 6.2.3 Sensibilité
à la casse dans la correspondance . . . . . 50 6.3 Expressions
de comparaison en tant que modèles . . . . . 51 6.4 Opérateurs
booléens et modèles. . . . . 51 6.5 Expressions en tant que
modèles. . . . . 52 6.6 Spécification des plages
d'enregistrements avec des modèles. . . . . 53 6.7 Modèles spéciaux
BEGIN et END . . . . . 53 6.8 Le motif
vide. 54

7 Aperçu des actions 55

8 Expressions comme instructions d'action . . . . . 57 8.1 Expressions
constantes 57 8.2
Variables . . . . .
59 8.2.1 Affectation de variables sur la ligne de commande 60 8.3
Opérateurs arithmétiques 60 8.4
Concaténation de chaînes . . . . .
61 8.5 Expressions de comparaison 62 8.6
Expressions booléennes. 64 8.7
Expressions d'affectation. 64 8.8
Opérateurs d'incrémentation. 66 8.9
Conversion de chaînes de caractères et de nombres 67
8.10 Valeurs numériques et chaînes de caractères. . . . . .
68 8.11 Expressions conditionnelles 69 8.12
Appels de fonctions . . . . .
70 8.13 Priorité des opérateurs (imbrication des opérateurs) 71

9 Instructions de contrôle dans les actions 73

9.1 L'instruction if . . . . . 73 9.2 L'instruction
while. . . . . 74 9.3 L'instruction do-
while. . . . . 74 9.4 L'instruction
« for » . . . . . 75 9.5 L'instruction
« break » 76 9.6
L'instruction « continue » . . . . . 77 9.7 L'énoncé
suivant . . . . . 78 9.8 L'énoncé de
sortie . ...

10 Tableaux avec awk . . . . . 81 10.1 Introduction aux
tableaux . . . . . 81 10.2 Référence à un
élément de tableau . . . . . 82 10.3 Affectation des éléments
d'un tableau . . . . . 83 10.4 Exemple simple de
tableau . . . . . 83 10.5 Parcours de tous les éléments
d'un tableau . . . . . 84 10.6 L'instruction
DELETE. . . . . 85 10.7 Utilisation des nombres
pour indexer les tableaux. . . . . 86 10.8 Tableaux
multidimensionnels . . . . . 86 10.9 Analyse de
tableaux multidimensionnels 88

dans

11 Fonctions intégrées . . . . . 89 11.1 Appel des fonctions intégrées . . . . . 89 11.2 Fonctions numériques intégrées . . . . . 89 11.3 Fonctions intégrées pour la manipulation de chaînes de caractères . . . . . 90 11.4 Fonctions intégrées pour les entrées/sorties 93

12 Fonctions définies par l'utilisateur . . . . . 95 12.1 Syntaxe des définitions de fonctions . . . . . 95 12.2 Exemple de définition de fonction . . . . . 96 12.3 Appel des fonctions définies par l'utilisateur . . . . . 97 12.4 Déclaration de retour. 98

13 Variables intégrées 101

13.1 Variables intégrées contrôlant awk . . . . . 101 13.2 Variables intégrées transmettant des informations 102

14. Appel de awk. . . . . 105 14.1 Options de la ligne de commande. . . . . 105 14.2 Autres arguments de la ligne de commande. 105

Annexe A : Résumé d'awk . . . . . 107 A.1 Résumé des options de ligne de commande. . . . . 107 A.2 Résumé du langage . . . . . 107 A.3 Variables et champs. . . . . 108 A.3.1 Champs . . . . . 108 A.3.2 Variables intégrées . . . . . 108 A.3.3 Tableaux. . . . . 109 A.3.4 Types de données. . . . . 110 A.4 Modèles et actions. . . . . 110 A.4.1 Modèles. . . . . 111 A.4.2 Expressions régulières . . . . . 112 A.4.3 Actions. . . . . 112 A.4.3.1 Opérateurs. . . . . 113 A.4.3.2 Instructions de contrôle . . . . . 113 A.4.3.3 Instructions d'E/S. . . . . 114 A.4.3.4 Résumé de printf . . . . . 114 A.4.3.5 Fonctions numériques. . . . . 115 A.4.3.6 Fonctions de chaînes de caractères. . . . . 116 A.4.3.7 Constantes de chaînes de caractères. . . . . 116 A.5 Fonctions 117

Annexe B Exemple de programme 119

Annexe C Glossaire 121

Index 125