Définition du langage R

Version 4.3.1 (2023-06-16) PROJET

Équipe de base R

Ce manuel concerne R, version 4.3.1 (2023-06-16).

Table des matières

1. Introduction

R est un système de calcul statistique et graphique. Il fournit, entre autres choses, un langage de programmation, des graphiques de haut niveau, des interfaces avec d'autres langages et des fonctionnalités de débogage.
Ce manuel détaille et définit le langage R.

Le langage R est un dialecte du S qui a été conçu dans les années 1980 et est depuis largement utilisé dans la communauté statistique. Son concepteur principal, John M. Chambers, a reçu le prix ACM Software Systems Award 1998 pour S.

La syntaxe du langage présente une similitude superficielle avec le C, mais la sémantique est de type FPL (langage de programmation fonctionnel) avec des affinités plus fortes avec Lisp et APL. Il permet notamment de « calculer sur le langage », ce qui permet à son tour d'écrire des fonctions qui prennent des expressions en entrée, ce qui est souvent utile pour la modélisation statistique et graphique.

Il est possible d'aller assez loin en utilisant R de manière interactive, en exécutant des expressions simples à partir de la ligne de commande. Certains utilisateurs n'auront peut-être jamais besoin d'aller au-delà de ce niveau, d'autres voudront écrire leurs propres fonctions, soit de manière ad hoc pour systématiser le travail répétitif, soit dans la perspective d'écrire des packages complémentaires pour de nouvelles fonctionnalités.

Le but de ce manuel est de documenter la langue en soi. C'est-à-dire les objets sur lesquels il travaille et les détails du processus d'évaluation de l'expression, qu'il est utile de connaître lors de la programmation des fonctions R. Les principaux sous-systèmes destinés à des tâches spécifiques, telles que les graphiques, ne sont que brièvement décrits dans ce manuel et seront documentés séparément.

Bien qu'une grande partie du texte s'applique également à S, il existe également quelques différences substantielles, et afin de ne pas confondre les choses, nous nous concentrerons sur la description de R.

La conception du langage contient un certain nombre de subtilités et d'embûches courantes qui peuvent surprendre l'utilisateur. La plupart d'entre eux sont dus à des considérations de cohérence à un niveau plus profond, comme nous l'expliquerons. Il existe également un certain nombre de raccourcis et d'expressions idiomatiques utiles, qui permettent à l'utilisateur d'exprimer succinctement des opérations assez compliquées. Beaucoup d'entre eux deviennent naturels une fois que l'on est familiarisé avec les concepts sous-jacents. Dans certains cas, il existe plusieurs façons d'effectuer une tâche, mais certaines techniques s'appuieront sur l'implémentation du langage, tandis que d'autres fonctionneront à un niveau d'abstraction plus élevé. Dans de tels cas, nous indiquerons l'utilisation préférée.

Une certaine familiarité avec R est supposée. Ceci n'est pas une introduction à R mais plutôt un manuel de référence pour les programmeurs. D'autres manuels fournissent des informations complémentaires : en particulier la section « Préface » dans An Introduction to R fournit une introduction à R et la section « Interfaces système et langues étrangères » dans Writing R Extensions détaille comment étendre R à l'aide de code compilé.

2 objets

Dans tout langage informatique, les variables permettent d'accéder aux données stockées en mémoire.
R ne fournit pas d'accès direct à la mémoire de l'ordinateur mais fournit plutôt un numéro
de structures de données spécialisées que nous appellerons objets. Ces objets sont désignés par
symboles ou variables. En R, cependant, les symboles sont eux-mêmes des objets et peuvent être manipulés
au même titre que tout autre objet. Ceci est différent de beaucoup d'autres langues et a une large portée
effets variés.

Dans ce chapitre, nous fournissons des descriptions préliminaires des différentes structures de données fournies dans R. Des discussions plus détaillées sur bon nombre d'entre eux seront trouvées dans les chapitres suivants. Le La fonction spécifique R typeof renvoie le type d'un objet R. Notez que dans le code C sous-jacent
R, tous les objets sont des pointeurs vers une structure avec typedef SEXPREC ; les différents types de données R sont représenté en C par SEXPTYPE, qui détermine comment les informations contenues dans les différentes parties de la structure est utilisée.

Le tableau suivant décrit les valeurs possibles renvoyées par typeof et ce qu'elles sont.

"NUL"	NUL
"symbole"	un nom de variable
"liste de paires"	un objet pairlist (principalement interne)
"fermeture"	une fonction
"environnement"	un environnement
"promesse"	un objet utilisé pour implémenter une évaluation paresseuse
"langue"	une construction en langage R
"spécial"	une fonction interne qui n'évalue pas ses arguments
"intégré"	une fonction interne qui évalue ses arguments
"carboniser"	un objet chaîne 'scalaire' (interne uniquement) ***
"logique"	un vecteur contenant des valeurs logiques
"entier"	un vecteur contenant des valeurs entières
"double"	un vecteur contenant des valeurs réelles
"caractère"	un vecteur contenant des valeurs complexes
"complexe"	un vecteur contenant des valeurs de caractères
"..."	l'argument spécial de longueur variable un type ***
"n'importe lequel"	spécial qui correspond à tous les types : il n'y a pas d'objets de ce genre
"expression" "liste"	un objet d'expression une liste
"bytecode"	code d'octet (interne uniquement) ***
"externalptr"	un objet pointeur externe
"faibleref"	un objet de référence faible
"brut"	un vecteur contenant des octets
"S4"	un objet S4 qui n'est pas un objet simple

Les utilisateurs ne peuvent pas facilement mettre la main sur des objets de types marqués d'un « *** ».

Le mode fonction donne des informations sur le mode d'un objet au sens de Becker, Chambers & Wilks (1988), et est plus compatible avec d'autres implémentations du langage S.
Enfin, la fonction storage.mode renvoie le mode de stockage de son argument au sens de
Becker et coll. (1988). Il est généralement utilisé lors de l'appel de fonctions écrites dans un autre langage,
comme C ou FORTRAN, pour garantir que les objets R ont le type de données attendu par la routine
être appelé. (En langage S, les vecteurs à valeurs entières ou réelles sont tous deux de mode "numérique",
leurs modes de stockage doivent donc être distingués.)

```
> x <- 1:3
> type de (x)
```

```
[1] "entier"
> mode(x)
[1] "numérique"
> stockage.mode(x)
[1] "entier"
```

Les objets R sont souvent contraints à différents types lors des calculs. Il y a aussi beaucoup fonctions disponibles pour effectuer une coercition explicite. Lors de la programmation en langage R, le Le type d'un objet n'affecte généralement pas les calculs, cependant, lorsqu'il s'agit d'objets étrangers langages ou du système d'exploitation, il est souvent nécessaire de s'assurer qu'un objet est du bon taper.

2.1 Types de base

2.1.1 Vecteurs

Les vecteurs peuvent être considérés comme des cellules contiguës contenant des données. Les cellules sont accessibles via opérations d'indexation telles que x[5]. Plus de détails sont donnés à la Section 3.4 [Indexation], page 15.

R a six types de vecteurs de base (« atomiques » : logique, entier, réel, complexe, caractère (en C aka 'string') et brut. Les modes et modes de stockage des différents types de vecteurs sont répertoriés dans le tableau suivant.

type	mode	mode de stockage
d'entier	numérique	logique
logique	logique	entier
double	numérique	double
personnage	complexe	complexe
de caractère complexe		
brut	brut	brut

Les nombres simples, tels que 4,2, et les chaînes, telles que « quatre virgule deux » sont toujours des vecteurs de longueur 1; il n'y a plus de types de base. Des vecteurs de longueur nulle sont possibles (et utiles).

Un élément unique d'un vecteur de caractères est souvent appelé chaîne de caractères ou abrégé. chaîne.[1]

2.1.2 Listes

Les listes (« vecteurs génériques ») sont un autre type de stockage de données. Les listes contiennent des éléments, dont chacun peut contenir n'importe quel type d'objet R, c'est-à-dire que les éléments d'une liste ne doivent pas nécessairement être du même type. Les éléments de liste sont accessibles via trois opérations d'indexation différentes. Ceux-ci sont expliqués dans détail dans la Section 3.4 [Indexation], page 15.

Les listes sont des vecteurs et les types de vecteurs de base sont appelés vecteurs atomiques. nécessaire d'exclure des listes.

2.1.3 Objets langage

Il existe trois types d'objets qui constituent le langage R. Ce sont des appels, des expressions, et des noms. Puisque R a des objets de type « expression », nous essaierons d'éviter l'utilisation du mot expression dans d'autres contextes. En particulier, les expressions syntaxiquement correctes seront mentionnées comme déclarations.

Ces objets ont respectivement les modes « appel », « expression » et « nom ».

[1] Notez que les vecteurs de caractères en interne sont de type STRSXP et que chaque élément (nous appelons chaîne ci-dessus) est de type C. CHARSXP (avec typeof(.) == "char" interne uniquement, voir [typeof-char], page 2).

Ils peuvent être créés directement à partir d'expressions à l'aide du mécanisme de guillemets et convertis vers et depuis des listes par les fonctions as.list et as.call. Les composants de l'arbre d'analyse peuvent être extraits à l'aide des opérations d'indexation standard.

2.1.3.1 Objets symboles Les symboles

font référence aux objets R. Le nom de tout objet R est généralement un symbole. Les symboles peuvent être créés via les fonctions as.name et quote.

Les symboles ont le mode « nom », le mode de stockage « symbole » et le type « symbole ». Ils peuvent être forcés vers et depuis des chaînes de caractères en utilisant as.character et as.name. Ils apparaissent naturellement sous forme d'atomes d'expressions analysées, essayez par exemple as.list(quote(x + y)).

2.1.4 Objets expression Dans R on peut avoir

des objets de type "expression". Une expression contient une ou plusieurs instructions. Une instruction est une collection de jetons syntaxiquement correcte. Les objets d'expression sont des objets de langage spéciaux qui contiennent des instructions R analysées mais non évaluées. La principale différence est qu'un objet expression peut contenir plusieurs de ces expressions. Une autre différence plus subtile est que les objets de type « expression » ne sont évalués que lorsqu'ils sont explicitement passés à eval, alors que d'autres objets de langage peuvent être évalués dans certains cas inattendus.

Un objet d'expression se comporte un peu comme une liste et ses composants doivent être accessibles dans le de la même manière que les composants d'une liste.

2.1.5 Objets fonction Dans R, les fonctions

sont des objets et peuvent être manipulées de la même manière que n'importe quel autre objet.
Les fonctions (ou plus précisément les fermetures de fonctions) ont trois composants de base : une liste d'arguments formelle, un corps et un environnement. La liste d'arguments est une liste d'arguments séparés par des virgules.
Un argument peut être un symbole, ou une construction « symbole = défaut », ou l'argument spécial....
La deuxième forme d'argument est utilisée pour spécifier une valeur par défaut pour un argument. Cette valeur sera utilisée si la fonction est appelée sans aucune valeur spécifiée pour cet argument. L'argument ... est spécial et peut contenir n'importe quel nombre d'arguments. Il est généralement utilisé si le nombre d'arguments est inconnu ou dans les cas où les arguments seront transmis à une autre fonction.

Le corps est une instruction R analysée. Il s'agit généralement d'une collection d'instructions entre accolades, mais il peut s'agir d'une instruction unique, d'un symbole ou même d'une constante.

L'environnement d'une fonction est l'environnement qui était actif au moment de la création de la fonction. Tous les symboles liés dans cet environnement sont capturés et disponibles pour la fonction. Cette combinaison du code de la fonction et des liaisons dans son environnement est appelée « fermeture de fonction », un terme issu de la théorie de la programmation fonctionnelle. Dans ce document, nous utilisons généralement le terme « fonction », mais nous utilisons « clôture » pour souligner l'importance de l'environnement attaché.

Il est possible d'extraire et de manipuler les trois parties d'un objet de fermeture à l'aide de constructions formelles, de corps et d'environnement (les trois peuvent également être utilisées sur le côté gauche des affectations). Le dernier d'entre eux peut être utilisé pour supprimer la capture d'environnement indésirable.

Lorsqu'une fonction est appelée, un nouvel environnement (appelé environnement d'évaluation) est créé, dont l'enceinte (voir Section 2.1.10 [Objets d'environnement], page 6) est l'environnement de fermeture de la fonction. Ce nouvel environnement est initialement rempli avec les arguments non évalués de la fonction ; au fur et à mesure que l'évaluation progresse, des variables locales sont créées en son sein.

Il existe également une fonction permettant de convertir des fonctions vers et depuis des structures de liste en utilisant as.list et as.function. Ceux-ci ont été inclus pour assurer la compatibilité avec S et leur utilisation est déconseillée.

2.1.6 NULLE

Il existe un objet spécial appelé NULL. Il est utilisé chaque fois qu'il est nécessaire d'indiquer ou de préciser qu'un objet est absent. Il ne faut pas le confondre avec un vecteur ou une liste de longueur nulle.

L'objet NULL n'a ni type ni propriétés modifiables. Il n'y a qu'un seul objet NULL dans R, auquel toutes les instances font référence. Pour tester NULL, utilisez is.null. Vous ne pouvez pas définir d'attributs sur NULL.

2.1.7 Objets intégrés et formes spéciales Ces deux types

d'objets contiennent les fonctions intégrées de R, c'est-à-dire celles qui sont affichées en tant que .Primitive dans les listes de codes (ainsi que celles accessibles via la fonction .Internal et donc non visibles par l'utilisateur comme objets). La différence entre les deux réside dans la gestion des arguments. Les fonctions intégrées voient tous leurs arguments évalués et transmis à la fonction interne, conformément à l'appel par valeur, tandis que les fonctions spéciales transmettent les arguments non évalués à la fonction interne.

Depuis le langage R, ces objets ne sont qu'un autre type de fonction. Le est.primitif
la fonction peut les distinguer des fonctions interprétées.

2.1.8 Objets Promise Les objets

Promise font partie du mécanisme d'évaluation paresseuse de R. Ils contiennent trois emplacements : une valeur, une expression et un environnement. Lorsqu'une fonction est appelée, les arguments correspondent, puis chacun des arguments formels est lié à une promesse. L'expression donnée pour cet argument formel et un pointeur vers l'environnement à partir duquel la fonction a été appelée sont stockés dans la promesse.

Jusqu'à ce que cet argument soit accédé, aucune valeur n'est associée à la promesse. Lors de l'accès à l'argument, l'expression stockée est évaluée dans l'environnement stocké et le résultat est renvoyé. Le résultat est également sauvegardé par la promesse. La fonction de substitution extraira le contenu du slot d'expression. Cela permet au programmeur d'accéder soit à la valeur, soit à l'expression associée à la promesse.

Dans le langage R, les objets de promesse ne sont presque visibles qu'implicitement : les arguments de fonction réels sont de ce type. Il existe également une fonction delayAssign qui fera une promesse à partir d'une expression. Il n'existe généralement aucun moyen dans le code R de vérifier si un objet est une promesse ou non, ni aucun moyen d'utiliser le code R pour déterminer l'environnement d'une promesse.

2.1.9 Point-point-point

Le type d'objet ... est stocké sous forme de liste de paires. Les composants de ... sont accessibles de la manière habituelle par liste de paires à partir du code C, mais ... n'est pas facilement accessible en tant qu'objet dans le code interprété, et même l'existence d'un tel objet ne doit généralement pas être supposée, car cela peut changement dans le futur.

L'objet peut être capturé (avec des promesses forcées !) sous forme de liste, par exemple dans un tableau
On voit

```
        arguments <- liste(...)
##....
        pour (a dans les arguments) {
##....
```

Notez que l'implémentation de ... en tant qu'objet pairlist ne doit pas être considérée comme faisant partie de l'API R, et le code en dehors de la base R ne doit pas s'appuyer sur cette description actuelle de .... D'un autre côté, la liste ci-dessus (. ..) access, et les autres fonctions « accès aux points » ...length(), ...elt(), ...names() et « mots réservés » ..1, ..2, etc, voir aussi la page d'aide ?dots, font partie de l'API R stable.

Si une fonction a ... comme argument formel, alors tous les arguments réels qui ne correspondent pas à un les arguments formels correspondent à ....

2.1.10 Environnements

Les environnements peuvent être considérés comme composés de deux choses. Un cadre, composé d'un ensemble de paires symbole-valeur, et d'une enceinte, un pointeur vers un environnement englobant. Lorsque R recherche la valeur d'un symbole, le cadre est examiné et si un symbole correspondant est trouvé, sa valeur sera renvoyée. Dans le cas contraire, l'environnement englobant est alors accédé et le processus est répété.
Les environnements forment une arborescence dans laquelle les enclos jouent le rôle de parents. L'arborescence des environnements est enracinée dans un environnement vide, disponible via emptyenv(), qui n'a pas de parent. C'est le parent direct de l'environnement du package de base (disponible via la fonction baseenv()).

Les environnements sont créés implicitement par des appels de fonction, comme décrit dans la Section 2.1.5 [Objets fonction], page 4, et la Section 3.5.2 [Environnement lexical], page 20. Dans ce cas, l'environnement contient les variables locales à la fonction (y compris les arguments), et son enceinte est l'environnement de la fonction actuellement appelée. Les environnements peuvent également être créés directement par new.env. Le contenu du cadre d'un environnement est accessible à l'aide de ls, noms, $, [, [[, get et get0, et manipulé par $<-, [[<-, et assign ainsi que eval et evalq.

La fonction parent.env peut être utilisée pour accéder à l'enceinte d'un environnement.

Contrairement à la plupart des autres objets R, les environnements ne sont pas copiés lorsqu'ils sont transmis aux fonctions ou utilisés dans des affectations. Ainsi, si vous attribuez le même environnement à plusieurs symboles et en modifiez un, les autres changeront également. En particulier, attribuer des attributs à un environnement peut entraîner des surprises.

2.1.11 Objets Pairlist Les objets Pairlist

sont similaires aux listes de paires pointées de Lisp. Ils sont largement utilisés dans les composants internes de R, mais sont rarement visibles dans le code interprété, bien qu'ils soient renvoyés par des formulaires formels et puissent être créés par (par exemple) la fonction pairlist. Une liste de paires de longueur nulle est NULL, comme on pouvait s'y attendre en Lisp mais contrairement à une liste de longueur nulle. Chacun de ces objets possède trois emplacements, une valeur CAR, une valeur CDR et une valeur TAG. La valeur TAG est une chaîne de texte et CAR et CDR représentent généralement respectivement un élément de liste (tête) et le reste (queue) de la liste avec un objet NULL comme terminateur (la terminologie CAR/CDR est traditionnelle Lisp et initialement appelée l'adresse et le décrément s'enregistrent sur un ordinateur IBM du début des années 60).

Les listes de paires sont gérées dans le langage R exactement de la même manière que les vecteurs génériques (« listes »). En particulier, les éléments sont accessibles en utilisant la même syntaxe [[]]. L'utilisation de listes de paires est déconseillée car les vecteurs génériques sont généralement plus efficaces à utiliser. Lorsqu'une liste de paires interne est accessible à partir de R, elle est généralement (y compris lorsqu'elle est sous-ensemble) convertie en un vecteur générique.

Dans de très rares cas, les listes de paires sont visibles par l'utilisateur : l'une d'entre elles est .Options.

2.1.12 Le type « Any » Il n'est pas vraiment

possible qu'un objet soit de type « Any », mais c'est néanmoins une valeur de type valide. Il est utilisé dans certaines circonstances (plutôt rares), par exemple as.vector(x, "any"), indiquant que la coercition de type ne doit pas être effectuée.

2.2 Attributs

Tous les objets, à l'exception de NULL, peuvent avoir un ou plusieurs attributs qui leur sont attachés. Les attributs sont stockés sous forme de liste de paires dans laquelle tous les éléments sont nommés, mais doivent être considérés comme un ensemble de paires nom=valeur. Une liste des attributs peut être obtenue à l'aide d'attributs et définie par attributs<-, les composants individuels sont accessibles en utilisant attr et attr<-.

Certains attributs ont des fonctions d'accès spéciales (par exemple niveaux <- pour les facteurs) et celles-ci doivent être utilisées lorsqu'elles sont disponibles. En plus de masquer les détails de mise en œuvre, ils peuvent effectuer des opérations supplémentaires. R tente d'intercepter les appels à attr<- et à attributs<- qui impliquent les attributs spéciaux et applique les contrôles de cohérence.

Les matrices et les tableaux sont simplement des vecteurs avec l'attribut dim et éventuellement des dimnames attaché au vecteur.

Les attributs sont utilisés pour implémenter la structure de classe utilisée dans R. Si un objet possède un attribut de classe, cet attribut sera examiné lors de l'évaluation. La structure des classes dans R est décrite en détail au Chapitre 5 [Programmation orientée objet], page 26.

2.2.1 Noms

Un attribut de noms, lorsqu'il est présent, étiquette les éléments individuels d'un vecteur ou d'une liste. Lorsqu'un objet est imprimé, l'attribut noms, lorsqu'il est présent, est utilisé pour étiqueter les éléments. L'attribut noms peut également être utilisé à des fins d'indexation, par exemple quantile(x)["25%"].

On peut obtenir et définir les noms en utilisant des constructions noms et noms<-. Ce dernier effectuera les contrôles de cohérence nécessaires pour s'assurer que l'attribut names a le bon type et la bonne longueur.

Les listes de paires et les tableaux unidimensionnels sont traités spécialement. Pour les objets de liste de paires, un attribut de noms virtuels est utilisé ; l'attribut noms est en réalité construit à partir des balises des composants de la liste. Pour les tableaux unidimensionnels, l'attribut names accède réellement à dimnames[[1]].

2.2.2 Dimensions

L'attribut dim est utilisé pour implémenter des tableaux. Le contenu du tableau est stocké dans un vecteur dans l'ordre des colonnes principales et l'attribut dim est un vecteur d'entiers spécifiant les étendues respectives du tableau. R garantit que la longueur du vecteur est le produit des longueurs des dimensions. La longueur d'une ou plusieurs dimensions peut être nulle.

Un vecteur n'est pas la même chose qu'un tableau unidimensionnel puisque ce dernier a un attribut dim de longueur un, alors que le premier n'a pas d'attribut dim.

2.2.3 Noms de couleurs

Les tableaux peuvent nommer chaque dimension séparément en utilisant l'attribut dimnames qui est une liste de vecteurs de caractères. La liste dimnames peut elle-même avoir des noms qui sont ensuite utilisés pour les en-têtes d'étendue lors de l'impression des tableaux.

2.2.4 Cours

R possède un système de classes[2] élaboré , principalement contrôlé via l'attribut class. Cet attribut est un vecteur de caractères contenant la liste des classes dont hérite un objet. Ceci constitue la base de la fonctionnalité « méthodes génériques » dans R.

Cet attribut peut être consulté et manipulé virtuellement sans restriction par les utilisateurs. Il n'est pas possible de vérifier qu'un objet contient réellement les composants attendus par les méthodes de classe. Ainsi, la modification de l'attribut class doit être effectuée avec prudence, et lorsqu'elles sont disponibles, des fonctions spécifiques de création et de coercition doivent être préférées.

2.2.5 Attributs des séries chronologiques

L'attribut tsp est utilisé pour contenir les paramètres de série temporelle, de début, de fin et de fréquence. Cette construction est principalement utilisée pour traiter des séries à sous-structure périodique telles que des données mensuelles ou trimestrielles.

---

[2] en fait deux, mais ce projet de manuel est antérieur au package de méthodes.

2.2.6 Copie d'attributs

La question de savoir si les attributs doivent être copiés lorsqu'un objet est modifié est un domaine complexe, mais il existe quelques règles générales (Becker, Chambers & Wilks, 1988, pp. 144-6).

Les fonctions scalaires (celles qui opèrent élément par élément sur un vecteur et dont la sortie est similaire à l'entrée) doivent conserver les attributs (sauf peut-être la classe).

Les opérations binaires copient normalement la plupart des attributs de l'argument le plus long (et s'ils sont de même longueur, des deux, préférant les valeurs du premier). Ici, « la plupart » signifie tout sauf les noms, dim et dimnames qui sont définis de manière appropriée par le code de l'opérateur.

Le sous-ensemble (autre que par un index vide) supprime généralement tous les attributs à l'exception des noms, des dim et des dimnames qui sont réinitialisés le cas échéant. D'un autre côté, la sous-affectation préserve généralement les attributs même si la longueur est modifiée. La coercition supprime tous les attributs.

La méthode de tri par défaut supprime tous les attributs à l'exception des noms, qui sont triés avec l'objet.

2.3 Objets composés spéciaux

2.3.1 Facteurs

Les facteurs sont utilisés pour décrire des éléments pouvant avoir un nombre fini de valeurs (sexe, classe sociale, etc.). Un facteur a un attribut de niveaux et une classe « facteur ». Facultativement, il peut également contenir un attribut contrasts qui contrôle la paramétrisation utilisée lorsque le facteur est utilisé dans une fonction de modélisation.

Un facteur peut être purement nominal ou avoir des catégories ordonnées. Dans ce dernier cas, il doit être défini comme tel et avoir un vecteur de classe c("ordonné", "facteur").

Les facteurs sont actuellement implémentés à l'aide d'un tableau d'entiers pour spécifier les niveaux réels et d'un deuxième tableau de noms mappés sur les entiers. Malheureusement, les utilisateurs utilisent souvent cette implémentation pour faciliter certains calculs. Il s'agit cependant d'un problème d'implémentation et il n'est pas garanti qu'il soit valable dans toutes les implémentations de R.

2.3.2 Objets de trame de données Les trames

de données sont les structures R qui imitent le plus fidèlement l'ensemble de données SAS ou SPSS, c'est-à-dire une matrice de données « cas par variables ».

Une trame de données est une liste de vecteurs, de facteurs et/ou de matrices ayant tous la même longueur (nombre de lignes dans le cas des matrices). De plus, une trame de données possède généralement un attribut noms pour étiqueter les variables et un attribut row.names pour étiqueter les cas.

Un bloc de données peut contenir une liste de même longueur que les autres composants. La liste peut contenir des éléments de différentes longueurs, fournissant ainsi une structure de données pour les tableaux irréguliers.
Cependant, au moment d'écrire ces lignes, ces tableaux ne sont généralement pas gérés correctement.

3 Évaluation des expressions

Lorsqu'un utilisateur tape une commande à l'invite (ou lorsqu'une expression est lue à partir d'un fichier), la première chose qui lui arrive est que la commande est transformée par l'analyseur en une représentation interne. L'évaluateur exécute les expressions R analysées et renvoie la valeur de l'expression. Toutes les expressions ont une valeur. C'est le cœur du langage.

Ce chapitre décrit les mécanismes de base de l'évaluateur, mais évite la discussion de fonctions ou de groupes de fonctions spécifiques qui sont décrits plus tard dans des chapitres séparés ou pour lesquels les pages d'aide devraient constituer une documentation suffisante.

Les utilisateurs peuvent construire des expressions et appeler l'évaluateur sur celles-ci.

3.1 Évaluation simple

3.1.1 Constantes

Tout nombre saisi directement à l'invite est une constante et est évalué.

```
> 1
[1] 1
```

De manière peut-être inattendue, le nombre renvoyé par l'expression 1 est un nombre. Dans la plupart des cas, la différence entre un entier et une valeur numérique n'aura pas d'importance car R fera la bonne chose en utilisant les nombres. Il y a cependant des moments où nous aimerions créer explicitement une valeur entière pour une constante. Nous pouvons le faire en appelant la fonction as.integer ou en utilisant diverses autres techniques. Mais l'approche la plus simple consiste peut-être à qualifier notre constante avec le suffixe « L ». Par exemple, pour créer la valeur entière 1, nous pourrions utiliser

```
> 1L
[1]
```

Nous pouvons utiliser le suffixe « L » pour qualifier n'importe quel nombre dans le but d'en faire un entier explicite. Ainsi, « 0x10L » crée la valeur entière 16 à partir de la représentation hexadécimale. La constante 1e3L donne 1000 sous forme de nombre entier plutôt que de valeur numérique et équivaut à 1000L. (Notez que le « L » est traité comme qualifiant le terme 1e3 et non le 3.) Si nous qualifions une valeur avec « L » qui n'est pas une valeur entière, par exemple 1e-3L, nous recevons un avertissement et la valeur numérique est créé. Un avertissement est également généré s'il y a un point décimal inutile dans le nombre, par exemple 1.L.

Nous obtenons une erreur de syntaxe lorsque nous utilisons 'L' avec des nombres complexes, par exemple 12iL donne une erreur.

Les constantes sont assez ennuyeuses et pour faire plus, nous avons besoin de symboles.

3.1.2 Recherche de symboles

Lorsqu'une nouvelle variable est créée, elle doit avoir un nom pour pouvoir être référencée et elle a généralement une valeur. Le nom lui-même est un symbole. Lorsqu'un symbole est évalué, sa valeur est renvoyée. Nous expliquerons plus tard en détail comment déterminer la valeur associée à un symbole.

Dans ce petit exemple, y est un symbole et sa valeur est 4. Un symbole est aussi un objet R, mais on a rarement besoin de traiter directement les symboles, sauf lorsqu'on fait de la « programmation sur le langage ».
(Chapitre 6 [L'informatique sur le langage], page 32). > y <- 4 > y

```
[1] 4
```

3.1.3 Appels de fonctions

La plupart des calculs effectués dans R impliquent l'évaluation de fonctions. Nous appellerons également cela l'invocation de fonction. Les fonctions sont invoquées par leur nom avec une liste d'arguments séparés par des virgules. > moyenne (1:10)

```
[1] 5,5
```

Dans cet exemple, la fonction moyenne a été appelée avec un seul argument, le vecteur d'entiers de 1 à 10.

R contient un grand nombre de fonctions ayant des objectifs différents. La plupart sont utilisés pour produire un résultat qui est un objet R, mais d'autres sont utilisés pour leurs effets secondaires, par exemple l'impression et le traçage les fonctions.

Les appels de fonction peuvent avoir des arguments balisés (ou nommés), comme dans plot(x, y, pch = 3). Arguments sans balises sont dites positionnelles puisque la fonction doit distinguer leur signification de leurs positions séquentielles parmi les arguments de l'appel, par exemple, que x désigne l'abscisse variable et y l'ordonnée. L'utilisation de balises/noms est une commodité évidente pour les fonctions avec un grand nombre d'arguments facultatifs.

Un type spécial d'appels de fonction peut apparaître sur le côté gauche de l'opérateur d'affectation un péché

```
> classe(x) <- "foo"
```

Ce que fait réellement cette construction, c'est d'appeler la fonction class<- avec l'objet d'origine et le côté droit. Cette fonction effectue la modification de l'objet et renvoie le résultat qui est ensuite stocké dans la variable d'origine. (Au moins conceptuellement, c'est ce qui se passe. Des efforts supplémentaires sont déployés pour éviter la duplication inutile des données.)

3.1.4 Opérateurs

R permet l'utilisation d'expressions arithmétiques utilisant des opérateurs similaires à ceux de la programmation C la langue, par exemple

```
> 1 + 2
[1] 3
```

Les expressions peuvent être regroupées à l'aide de parenthèses, mélangées à des appels de fonction et affectées à variables de manière simple

```
> y <- 2 * (a + log(x))
```

R contient un certain nombre d'opérateurs. Ils sont répertoriés dans le tableau ci-dessous.

-	Moins, peut être unaire ou binaire
+	De plus, peut être unaire ou binaire
!	Unaire non
~	Tilde, utilisé pour les formules modèles, peut être unaire ou binaire
?	Aide
:	Séquence binaire (dans les formules modèles : interaction)
*	Multiplication, binaire
/	Division, binaire
^	Exponentiation, binaire
%X%	Opérateurs binaires spéciaux, x peut être remplacé par n'importe quel nom
%	Module, binaire
% %/	Division entière, binaire
%	Produit matriciel, binaire
%*%	Produit externe, binaire
%o%	Produit Kronecker, binaire
%x% %in%	Opérateur de correspondance, binaire (dans les formules modèles : imbrication)
<	Inférieur à, binaire
>	Supérieur à, binaire
==	Égal à, binaire
>=	Supérieur ou égal à, binaire

<=	Inférieur ou égal à, binaire
&	Et, binaire, vectorisé
&&	Et, binaire, non vectorisé
|	Ou, binaire, vectorisé
||	Ou, binaire, non vectorisé
<-	Affectation de gauche, binaire
->	Affectation correcte, binaire
$	Sous-ensemble de liste, binaire

Hormis la syntaxe, il n'y a aucune différence entre appliquer un opérateur et appeler un fonction. En fait, x + y peut s'écrire de manière équivalente « + » (x, y). Notez que puisque « + » est un nom de fonction non standard, il doit être mis entre guillemets.

R traite des vecteurs entiers de données à la fois, et la plupart des opérateurs élémentaires et de base les fonctions mathématiques comme log sont vectorisées (comme indiqué dans le tableau ci-dessus). Cela signifie que, par exemple, l'ajout de deux vecteurs de même longueur créera un vecteur contenant l'élément par élément sommes, en boucle implicitement sur l'index vectoriel. Cela s'applique également à d'autres opérateurs comme -, *, et/ainsi qu'à des structures de dimensions supérieures. Notez en particulier que multiplier deux matrices ne produit pas le produit matriciel habituel (l'opérateur %*% existe à cet effet).
Certains points plus fins relatifs aux opérations vectorisées seront abordés dans la section 3.3 [Elementary opérations arithmétiques], page 14.

Pour accéder aux éléments individuels d'un vecteur atomique, on utilise généralement la construction x[i].

```
> x <- rnorme(5)
>x
 [1] -0,12526937 -0,27961154 -1,03718717 -0,08156527 1,37167090
>x[2]
 [1] -0,2796115
```

Les composants de liste sont plus couramment accessibles en utilisant x$a ou x[[i]].

```
> x <-options()
> x$invite
 [1] "> "
```

Les constructions d'indexation peuvent également apparaître sur le côté droit d'une affectation.

Comme les autres opérateurs, l'indexation se fait en réalité par fonctions, et on aurait pu utiliser '['(x, 2) au lieu de x[2].

Les opérations d'indexation de R contiennent de nombreuses fonctionnalités avancées qui sont décrites plus en détail dans Section 3.4 [Indexation], page 15.

3.2 Structures de contrôle

Le calcul dans R consiste à évaluer séquentiellement des instructions. Déclarations, telles que x<-1:10 ou moyenne(y), peut être séparé par un point-virgule ou une nouvelle ligne. Chaque fois que l'évaluateur est présenté avec une instruction syntaxiquement complète indiquant que l'instruction est évaluée et la valeur revenu. Le résultat de l'évaluation d'une déclaration peut être appelé la valeur de la déclaration[1]
La valeur peut toujours être attribuée à un symbole.

Les points-virgules et les nouvelles lignes peuvent être utilisés pour séparer les instructions. Un point-virgule toujours indique la fin d'une instruction tandis qu'une nouvelle ligne peut indiquer la fin d'une instruction. Si la l'instruction actuelle n'est pas syntaxiquement complète, les nouvelles lignes sont simplement ignorées par l'évaluateur. Si la session est interactive, l'invite passe de « > » à « + ».

```
> x <- 0 ; x + 5
```

[1] L'évaluation se déroule toujours dans un environnement. Voir Section 3.5 [Portée des variables], page 20, pour plus d'informations. détails.

```
 [1] 5 >
et <- 1:10 > 1 ;
2 [1] 1
 [1] 2
```

Les instructions peuvent être regroupées à l'aide des accolades '{' et '}'. Un groupe d'instructions est parfois appelé un bloc. Les instructions simples sont évaluées lorsqu'une nouvelle ligne est tapée à la fin de l'instruction syntaxiquement complète. Les blocs ne sont pas évalués tant qu'une nouvelle ligne n'est pas saisie après l'accolade fermante. Dans le reste de cette section, instruction fait référence soit à une instruction unique, soit à un bloc.

```
> { x <- 0
+ x + 5
+ }
 [1] 5
```

3.2.1 si

L'instruction if/else évalue conditionnellement deux instructions. Il existe une condition qui est évaluée et si la valeur est VRAIE alors la première instruction est évaluée ; sinon, la deuxième instruction sera évaluée. L'instruction if/else renvoie, comme valeur, la valeur de l'instruction sélectionnée. La syntaxe formelle est

```
si (instruction1)
        déclaration2
autre
        déclaration3
```

Tout d'abord, l'instruction1 est évaluée pour produire la valeur1. Si value1 est un vecteur logique avec le premier élément TRUE alors l'instruction2 est évaluée. Si le premier élément de value1 est FALSE, alors l'instruction3 est évaluée. Si value1 est un vecteur numérique, alors l'instruction3 est évaluée lorsque le premier élément de value1 est zéro et sinon, l'instruction2 est évaluée. Seul le premier élément de value1 est utilisé.
Tous les autres éléments sont ignorés. Si value1 a un type autre qu'un vecteur logique ou numérique, une erreur est signalée.

Les instructions if/else peuvent être utilisées pour éviter des problèmes numériques tels que la prise du logarithme d'un nombre négatif. Parce que les instructions if/else sont les mêmes que les autres instructions, vous pouvez leur attribuer une valeur. Les deux exemples ci-dessous sont équivalents.

```
> if( any(x <= 0) ) y <- log(1+x) else y <- log(x) > y <- if( any(x <= 0) )
log(1+x) else log (X)
```

La clause else est facultative. L'instruction if(any(x <= 0)) x <- x[x <= 0] est valide. Lorsque l'instruction if n'est pas dans un bloc, l'instruction else, si elle est présente, doit apparaître sur la même ligne que la fin de l'instruction2. Sinon, la nouvelle ligne à la fin de l'instruction2 complète le if et produit une instruction syntaxiquement complète qui est évaluée. Une solution simple consiste à utiliser une instruction composée entourée d'accolades, en plaçant le else sur la même ligne que l'accolade fermante qui marque la fin de l'instruction.

```
Les instructions if/else peuvent être
    imbriquées. si (instruction1)
            {instruction2
    } sinon si ( instruction3 ) {
            instruction4 }
    else if ( instruction5 ) { instruction6 }
            else instruction8
```

L'une des instructions paires sera évaluée et la valeur résultante sera renvoyée. Si la clause else facultative est omise et que toutes les instructions impaires sont évaluées à FALSE, aucune instruction ne sera évaluée et NULL est renvoyé.

Les instructions impaires sont évaluées, dans l'ordre, jusqu'à ce que l'on obtienne la valeur VRAI, puis l'instruction paire associée est évaluée. Dans cet exemple, l'instruction 6 ne sera évaluée que si l'instruction 1 est FAUX, si l'instruction 3 est FAUX et si l'instruction 5 est VRAI. Il n'y a pas de limite au nombre de clauses else if autorisées.

3.2.2 Bouclage R

comporte trois instructions qui fournissent un bouclage explicite2. Il s'agit de for, while et repeat. Les deux constructions intégrées, next et break, offrent un contrôle supplémentaire sur l'évaluation. R fournit d'autres fonctions pour le bouclage implicite telles que tapply, apply et lapply. De plus, de nombreuses opérations, notamment arithmétiques, sont vectorisées, vous n'aurez donc peut-être pas besoin d'utiliser une boucle.

Deux instructions peuvent être utilisées pour contrôler explicitement le bouclage. Ils sont en pause et ensuite. L'instruction break provoque une sortie de la boucle la plus interne en cours d'exécution. L'instruction suivante ramène immédiatement le contrôle au début de la boucle.
L'itération suivante de la boucle (s'il y en a une) est alors exécutée. Aucune instruction ci-dessous dans la boucle actuelle n'est évaluée.

La valeur renvoyée par une instruction de boucle est toujours NULL et est renvoyée de manière invisible.

3.2.3 répéter

L'instruction répéter provoque une évaluation répétée du corps jusqu'à ce qu'une pause soit spécifiquement demandée. Cela signifie que vous devez être prudent lorsque vous utilisez la répétition en raison du danger d'une boucle infinie. La syntaxe de la boucle de répétition est

```
répéter la déclaration
```

Lors de l'utilisation de la répétition, l'instruction doit être une instruction de bloc. Vous devez à la fois effectuer des calculs et tester s'il faut ou non rompre la boucle, ce qui nécessite généralement deux instructions.

3.2.4 pendant que

L'instruction while est très similaire à l'instruction repeat. La syntaxe de la boucle while est

```
while ( instruction1 ) instruction2
```

où instruction1 est évaluée et si sa valeur est VRAIE alors instruction2 est évaluée. Ce processus se poursuit jusqu'à ce que instruction1 soit évaluée à FALSE.

3.2.5 pour

La syntaxe de la boucle for est

```
pour (nom dans le vecteur)
    instruction1
```

où vector peut être soit un vecteur, soit une liste. Pour chaque élément du vecteur, le nom de la variable est défini sur la valeur de cet élément et l'instruction 1 est évaluée. Un effet secondaire est que le nom de la variable existe toujours une fois la boucle terminée et qu'il a la valeur du dernier élément du vecteur pour lequel la boucle a été évaluée.

3.2.6 commutateur

Techniquement parlant, switch n'est qu'une fonction parmi d'autres, mais sa sémantique est proche de celle des structures de contrôle d'autres langages de programmation.

---

[2] Le bouclage est l'évaluation répétée d'une instruction ou d'un bloc d'instructions.

La syntaxe est

switch (instruction, liste)

où les éléments de la liste peuvent être nommés. Tout d'abord, l'énoncé est évalué et le résultat, la valeur, est obtenu. Si la valeur est un nombre compris entre 1 et la longueur de la liste, alors l'élément correspondant de la liste est évalué et le résultat est renvoyé. Si la valeur est trop grande ou trop petite, NULL est renvoyé.

```
> x <- 3 >
switch(x, 2+2, moyenne(1:10), rnorm(5)) [1] 2,2903605
 2,3271663 -0,7060073 1,3622045 -0,2892720 > switch(2, 2+2, moyenne(1:10 ), rnorm(5))
[1] 5.5 > switch(6, 2+2, moyenne(1:10), rnorm(5))


NUL
```

Si value est un vecteur de caractères, alors l'élément de ... avec un nom qui correspond exactement à value est évalué. S'il n'y a pas de correspondance, un seul argument sans nom sera utilisé par défaut. Si aucune valeur par défaut n'est spécifiée, NULL est renvoyé.

```
> y <- "fruit" >
switch(y, fruit = "banane", légumes = "brocoli", "Ni l'un ni l'autre") [1] "banane" > y <- "viande" >
 switch(y, fruit =
"banane ", légume
= "brocoli", "Ni l'un ni l'autre")
 [1] "Ni l'un ni l'autre"
```

Une utilisation courante de switch consiste à effectuer un branchement en fonction de la valeur de caractère de l'un des arguments d'une fonction.

```
> centre <- function(x, type) { + switch(type,
moyenne =
+              moyenne(x), médiane =
+                médiane(x), rogné =
+                moyenne(x, trim = .1))
+ }
> x <- rcauchy(10) >
centre(x, "moyenne") [1]
 0,8760325 >
centre(x, "médiane") [1]
 0,5360891 >
centre(x, "tronqué") [1] 0,6086504
```

switch renvoie soit la valeur de l'instruction qui a été évaluée, soit NULL si aucune instruction a été évalué.

Choisir parmi une liste d'alternatives qui existe déjà, switch n'est peut-être pas la meilleure façon d'en sélectionner une pour l'évaluation. Il est souvent préférable d'utiliser eval et l'opérateur de sous-ensemble, [[, directement via eval(x[[condition]]).

3.3 Opérations arithmétiques élémentaires

Dans cette section, nous discutons des subtilités des règles qui s'appliquent aux opérations de base comme l'addition ou la multiplication de deux vecteurs ou matrices.

3.3.1 Règles de recyclage

Si l'on tente d'ajouter deux structures avec un nombre d'éléments différent, alors la plus courte est recyclée à la longueur la plus longue. Autrement dit, si par exemple vous ajoutez c(1, 2, 3) à un vecteur à six éléments

alors vous ajouterez vraiment c(1, 2, 3, 1, 2, 3). Si la longueur du vecteur le plus long n'est pas un multiple de celle du plus court, un avertissement est émis.

Depuis R 1.4.0, toute opération arithmétique impliquant un vecteur de longueur nulle a un résultat de longueur nulle.

3.3.2 Propagation des noms propagation des noms

(le premier gagne, je pense - même s'il n'a pas de nom ?? —- le premier *avec des noms* gagne, le recyclage fait perdre des noms au plus court)

3.3.3 Attributs dimensionnels

(matrice+matrice, les dimensions doivent correspondre. vecteur+matrice : recyclez d'abord, puis vérifiez si les dimensions conviennent, erreur sinon)

3.3.4 Gestion de NA Les valeurs

manquantes au sens statistique, c'est-à-dire les variables dont la valeur n'est pas connue, ont la valeur NA. Cela ne doit pas être confondu avec la propriété manquante pour un argument de fonction qui n'a pas été fourni (voir Section 4.1.2 [Arguments], page 22).

Comme les éléments d'un vecteur atomique doivent être du même type, il existe plusieurs types de valeurs NA. Il existe un cas où cela est particulièrement important pour l'utilisateur. Le type par défaut de NA est logique, à moins d'être contraint à un autre type, donc l'apparition d'une valeur manquante peut déclencher une indexation logique plutôt que numérique (voir Section 3.4 [Indexation], page 15, pour plus de détails).

Les calculs numériques et logiques avec NA renvoient généralement NA. Dans les cas où le résultat de l'opération serait le même pour toutes les valeurs possibles que le NA pourrait prendre, l'opération peut renvoyer cette valeur. En particulier, « FALSE & NA » est FALSE, « TRUE | NA' est VRAI. NA n'est égal à aucune autre valeur ni à lui-même ; les tests pour NA sont effectués à l'aide de is.na. Cependant, une valeur NA correspondra à une autre valeur NA en correspondance.

Les calculs numériques dont le résultat n'est pas défini, comme « 0/0 », produisent la valeur NaN. Ceci n'existe que dans le type double et pour des composants réels ou imaginaires de type complexe.
La fonction is.nan est fournie pour vérifier spécifiquement NaN, is.na renvoie également TRUE pour NaN.
La contrainte de NaN en type logique ou entier donne un NA du type approprié, mais la contrainte en caractère donne la chaîne "NaN". Les valeurs NaN sont incomparables, donc les tests d'égalité ou de collation impliquant NaN entraîneront NA. Ils sont considérés comme correspondant à n'importe quelle valeur NaN (et à aucune autre valeur, pas même NA) par correspondance.

Le NA du type caractère est à partir de R 1.5.0 distinct de la chaîne "NA". Les programmeurs qui doivent spécifier une chaîne explicite NA doivent utiliser « NA_character_ » plutôt que « NA », ou définir les éléments sur NA en utilisant is.na<-.

Il existe des constantes NA_integer_, NA_real_, NA_complex_ et NA_character_ qui généreront (dans l'analyseur) une valeur NA du type approprié et seront utilisées lors de l'analyse lorsqu'il n'est pas possible autrement d'identifier le type d'un NA (et les options de contrôle demandez que cela soit fait).

Il n'y a pas de valeur NA pour les vecteurs bruts.

3.4 Indexation

R contient plusieurs constructions qui permettent d'accéder à des éléments ou sous-ensembles individuels via des opérations d'indexation. Dans le cas des types vectoriels de base, on peut accéder au i-ème élément en utilisant x[i], mais il existe également une indexation des listes, des matrices et des tableaux multidimensionnels. Il existe plusieurs formes d'indexation en plus de l'indexation avec un seul entier. L'indexation peut être utilisée à la fois pour extraire une partie d'un objet et pour remplacer des parties d'un objet (ou pour ajouter des parties).

R dispose de trois opérateurs d'indexation de base, avec une syntaxe affichée par les exemples suivants

x[i]

x[i, j] x[[i]]
x[[i, j]]

xax"a"

Pour les vecteurs et les matrices, les formes [[sont rarement utilisées, bien qu'elles présentent quelques légères différences sémantiques par rapport à la forme [(par exemple, elle supprime tout attribut noms ou dimnames, et cette correspondance partielle est utilisée pour les indices de caractères). Lors de l'indexation de structures multidimensionnelles avec un seul index, x[[i]] ou x[i] renverra le ième élément séquentiel de x.

Pour les listes, on utilise généralement [[pour sélectionner n'importe quel élément, alors que [renvoie une liste des éléments sélectionnés.

Le formulaire [[permet de sélectionner un seul élément à l'aide d'indices entiers ou de caractères, tandis que [permet l'indexation par vecteurs. Notez cependant que pour une liste ou un autre objet récursif, l'index peut être un vecteur et chaque élément du vecteur est appliqué tour à tour à la liste, au composant sélectionné, au composant sélectionné de ce composant, etc. Le résultat est toujours un seul élément.

Le formulaire utilisant $ s'applique aux objets récursifs tels que les listes et les listes de paires. Il autorise uniquement une chaîne de caractères littérale ou un symbole comme index. Autrement dit, l'index n'est pas calculable : dans les cas où vous devez évaluer une expression pour trouver l'index, utilisez x[[expr]]. Appliquer $ à un objet non récursif est une erreur.

3.4.1 Indexation par vecteurs R permet des

constructions puissantes utilisant des vecteurs comme indices. Nous discuterons d'abord de l'indexation des vecteurs simples. Pour plus de simplicité, supposons que l'expression soit x[i]. Alors les possibilités suivantes existent selon le type de i.

- Entier. Tous les éléments de i doivent avoir le même signe. S'ils sont positifs, les éléments de x avec ces numéros d'index sont sélectionnés. Si i contient des éléments négatifs, tous les éléments sauf ceux indiqués sont sélectionnés.

  Si i est positif et dépasse length(x) alors la sélection correspondante est NA. Les valeurs négatives hors limites pour i sont silencieusement ignorées depuis la version R 2.6.0, de manière compatible S, car elles signifient supprimer des éléments inexistants et il s'agit d'une opération vide (« no-op »).

  Un cas particulier est l'indice zéro, qui a des effets nuls : x[0] est un vecteur vide et, par ailleurs, inclure des zéros parmi les indices positifs ou négatifs a le même effet que s'ils étaient omis.

- Autre numérique. Les valeurs non entières sont converties en entier (par troncature vers zéro)
  Avant utilisation.

- Logique. L'indexation i devrait généralement avoir la même longueur que x. S'il est plus court, alors ses éléments seront recyclés comme indiqué dans la section 3.3 [Opérations arithmétiques élémentaires], page 14. S'il est plus long, alors x est conceptuellement étendu avec les NA. Les valeurs sélectionnées de x sont celles pour lesquelles i est VRAI.

- Personnage. Les chaînes de i sont comparées à l'attribut noms de x et les entiers résultants sont utilisés. Pour [[et $, la correspondance partielle est utilisée si la correspondance exacte échoue, donc x$aa correspondra à x$aabb si x ne contient pas de composant nommé "aa" et "aabb" est le seul nom qui a le préfixe "aa". Pour [[, la correspondance partielle peut être contrôlée via l'argument exact qui est par défaut NA, indiquant que la correspondance partielle est autorisée, mais devrait entraîner un avertissement lorsqu'elle se produit. La définition de exact sur TRUE empêche la correspondance partielle, une valeur FALSE le permet et n'émet aucun avertissement. Notez que [nécessite toujours un exact

correspondre. La chaîne "" est traitée spécialement : elle indique 'aucun nom' et ne correspond à aucun élément (pas même ceux sans nom). Notez que la correspondance partielle n'est utilisée que lors de l'extraction et non lors du remplacement.

- Facteur. Le résultat est identique à x[as.integer(i)]. Les niveaux de facteurs ne sont jamais utilisés. Si si vous le souhaitez, utilisez x[as.character(i)] ou une construction similaire.
- Vide. L'expression x[] renvoie x, mais supprime les attributs « non pertinents » du résultat. Seuls les noms et les attributs dim et dimnames dans les tableaux multidimensionnels sont conservés. • NUL.

Ceci est traité comme s'il s'agissait d'un entier (0).

L'indexation avec une valeur manquante (c'est-à-dire NA) donne un résultat NA. Cette règle s'applique également au cas d'indexation logique, c'est à dire que les éléments de x qui ont un sélecteur NA dans i sont inclus dans le résultat, mais leur valeur sera NA.

Notez cependant qu'il existe différents modes de NA : la constante littérale est de mode « logique », mais elle est fréquemment automatiquement contrainte à d'autres types. Un des effets de ceci est que x[NA] a la longueur de x, mais x[c(1, NA)] a une longueur de 2. En effet, les règles pour les indices logiques s'appliquent dans le premier cas, mais celles pour les indices entiers dans le dernier.

L'indexation avec [effectuera également le sous-ensemble pertinent de tous les attributs de noms.

3.4.2 Matrices et tableaux d'indexation Le sous-ensemble

des structures multidimensionnelles suit généralement les mêmes règles que l'indexation unidimensionnelle pour chaque variable d'index, le composant pertinent des dimnames prenant la place des noms. Quelques règles particulières s'appliquent cependant :

Normalement, on accède à une structure en utilisant le nombre d'indices correspondant à sa dimension. Il est cependant également possible d'utiliser un seul index auquel cas les attributs dim et dimnames sont ignorés et le résultat est effectivement celui de c(m)[i]. Notez que m[1] est généralement très différent de m[1,] ou m[, 1].

Il est possible d'utiliser une matrice d'entiers comme index. Dans ce cas, le nombre de colonnes de la matrice doit correspondre au nombre de dimensions de la structure, et le résultat sera un vecteur dont la longueur est égale au nombre de lignes de la matrice. L'exemple suivant montre comment extraire les éléments m[1, 1] et m[2, 2] en une seule opération.

```
> m <- matrice (1:4, 2)
> m
     [,1] [,2] 3
[1,]    1
[2,]    2    4
> i <- matrice(c(1, 1, 2, 2), 2, byrow = VRAI) > i

     [,1] [,2] 1
[1,]    1
[2,]    2    2
> m[je]
[1] 1 4
```

Les matrices d'indexation ne peuvent pas contenir d'indices négatifs. Les valeurs NA et zéro sont autorisées : les lignes d'une matrice d'index contenant un zéro sont ignorées, tandis que les lignes contenant un NA produisent un NA dans le résultat.

Tant dans le cas de l'utilisation d'un index unique que dans l'indexation matricielle, un attribut de noms est utilisé s'il est présent, comme si la structure avait été unidimensionnelle.

Si une opération d'indexation fait que le résultat a une de ses étendues de longueur un, comme lors de la sélection d'une seule tranche d'une matrice tridimensionnelle avec (disons) m[2, ,], la dimension correspondante est généralement supprimée du résultat. Si une structure unidimensionnelle en résulte, un

le vecteur est obtenu. Ceci est parfois indésirable et peut être désactivé en ajoutant « drop = FALSE » à l'opération d'indexation. Notez qu'il s'agit d'un argument supplémentaire à la fonction [et ne s'ajoute pas au nombre d'index. Par conséquent, la manière correcte de sélectionner la première ligne d'une matrice en tant que matrice 1 par n est m[1, , drop = FALSE]. Oublier de désactiver la fonction de suppression est une cause fréquente d'échec dans les sous-programmes généraux où un index a occasionnellement, mais pas généralement, une longueur de un. Cette règle s'applique toujours à un tableau unidimensionnel, où tout sous-ensemble donnera un résultat vectoriel à moins que « drop = FALSE » ne soit utilisé.

Notez que les vecteurs sont distincts des tableaux unidimensionnels dans la mesure où ces derniers ont des attributs dim et dimnames (tous deux de longueur un). Les tableaux unidimensionnels ne sont pas facilement obtenus à partir d'opérations de sous-ensemble, mais ils peuvent être construits explicitement et sont renvoyés par table. Ceci est parfois utile car les éléments de la liste dimnames peuvent eux-mêmes être nommés, ce qui n'est pas le cas de l'attribut names.

Certaines opérations telles que m[FALSE,] aboutissent à des structures dans lesquelles une dimension a une étendue nulle. R essaie généralement de gérer ces structures de manière judicieuse.

3.4.3 Indexation d'autres structures L'opérateur [est une

fonction générique qui permet d'ajouter des méthodes de classe, ainsi que les opérateurs $ et [[. Ainsi, il est possible d'avoir des opérations d'indexation définies par l'utilisateur pour n'importe quelle structure. Une telle fonction, disons [.foo, est appelée avec un ensemble d'arguments dont le premier est la structure en cours d'indexation et le reste sont les indices. Dans le cas de $, l'argument d'index est en mode "symbole" même en utilisant la forme x$"abc". Il est important d'être conscient que les méthodes de classe ne se comportent pas forcément de la même manière que les méthodes de base, par exemple en matière de correspondance partielle.

L'exemple le plus important de méthode de classe pour [est celui utilisé pour les trames de données. Il n'est pas décrit en détail ici (voir la page d'aide pour [.data.frame), mais en termes généraux, si deux indices sont fournis (même si l'un d'entre eux est vide), cela crée une indexation de type matricielle pour une structure qui est fondamentalement un liste de vecteurs de même longueur. Si un seul index est fourni, il est interprété comme indexant la liste des colonnes ; dans ce cas, l'argument drop est ignoré, avec un avertissement.

Les opérateurs de base $ et [[peuvent être appliqués aux environnements. Seuls les indices de caractères sont autorisé et aucune correspondance partielle n'est effectuée.

3.4.4 Affectation de sous-ensembles

L'affectation aux sous-ensembles d'une structure est un cas particulier de mécanisme général d'affectation complexe :

```
x[3:5] <- 13:15
```

Le résultat de cette commande est comme si ce qui suit avait été exécuté '*tmp*' <- xx <-

```
"[<-"('*tmp*', 3:5,
value=13:15) rm('*tmp *')
```

Notez que l'index est d'abord converti en index numérique, puis les éléments sont remplacés séquentiellement le long de l'index numérique, comme si une boucle for avait été utilisée. Toute variable existante appelée '*tmp*' sera écrasée et supprimée, et ce nom de variable ne doit pas être utilisé dans le code.

Le même mécanisme peut être appliqué à des fonctions autres que [. La fonction de remplacement porte le même nom avec <- collé. Son dernier argument, qu'il faut appeler valeur, est la nouvelle valeur à attribuer. Par exemple, noms(x) <- c("a","b")

est équivalent à

```
'*tmp*' <- x
```

x <- "noms<-"('*tmp*', value=c("a","b")) rm('*tmp*')

L'imbrication d'affectations complexes est évaluée de manière récursive

noms(x)[3] <- "Trois"

est équivalent à

'*tmp*' <- xx <-
"names<-"('*tmp*', value="[<-"(names('*tmp*'), 3, value="Trois")) rm('*tmp*')

Les affectations complexes dans l'environnement englobant (en utilisant <<-) sont également autorisées :

noms(x)[3] <<- "Trois"

est équivalent à

'*tmp*' <<- get(x, envir=parent.env(), Heheres=TRUE) names('*tmp*')[3] <-
"Trois" x <<- '*tmp *' rm('*tmp*') et aussi
à

'*tmp*' <- get(x,envir=parent.env(), Heheres=TRUE) x <<- "names<-"('*tmp*',
value="[<-"(names(' *tmp*'), 3, valeur="Trois")) rm('*tmp*')

Seule la variable cible est évaluée dans l'environnement englobant, donc

e<-c(a=1,b=2)
je<-1
local({ e
<- c(A=10,B=11) je <-2

e[i] <<- e[i]+1
})

utilise la valeur locale de i sur les deux LHS et RHS, et la valeur locale de e sur le RHS de
la déclaration de suraffectation. Il définit e dans l'environnement extérieur pour

un B
1 12

Autrement dit, la superaffectation est équivalente aux quatre lignes

'*tmp*' <- get(e, envir=parent.env(), Heheres=TRUE) '*tmp*'[i] <- e[i]+1 e
<<- '*tmp*' rm('*tmp*')

De la même manière

x[est.na(x)] <<- 0

est équivalent à

'*tmp*' <- get(x,envir=parent.env(), Heheres=TRUE) '*tmp*'[is.na(x)] <- 0
x <<- '*tmp* ' rm('*tmp*') et non

'*tmp*' <- get(x,envir=parent.env(), Heheres=TRUE) '*tmp*'[is.na('*tmp*')]
<- 0 x <<- '*tmp *'

```
rm('*tmp*')
```

Ces deux interprétations candidates ne diffèrent que s'il existe également une variable locale x. C'est une bonne idée d'éviter d'avoir une variable locale portant le même nom que la variable cible d'une superaffectation. Comme ce cas n'a pas été traité correctement dans les versions 1.9.1 et antérieures, un tel code ne doit pas être sérieusement nécessaire.

3.5 Portée des variables

Presque tous les langages de programmation disposent d'un ensemble de règles de portée, permettant d'utiliser le même nom pour différents objets. Cela permet, par exemple, à une variable locale d'une fonction d'avoir le même nom qu'un objet global.

R utilise un modèle de portée lexicale, similaire aux langages comme Pascal. Cependant, R est un langage de programmation fonctionnel qui permet la création et la manipulation dynamiques de fonctions et d'objets langage, et possède des fonctionnalités supplémentaires reflétant ce fait.

3.5.1 Environnement mondial

L'environnement global est la racine de l'espace de travail utilisateur. Une opération d'affectation à partir de la ligne de commande fera appartenir l'objet concerné à l'environnement global. Son environnement englobant est l'environnement suivant sur le chemin de recherche, et ainsi de suite jusqu'à l'environnement vide qui est l'enceinte de l'environnement de base.

3.5.2 Environnement lexical

Chaque appel à une fonction crée un cadre qui contient les variables locales créées dans la fonction et est évalué dans un environnement qui, en combinaison, crée un nouvel environnement.

Notez la terminologie : un cadre est un ensemble de variables, un environnement est une imbrication de cadres (ou de manière équivalente : le cadre le plus interne plus l'environnement englobant).

Les environnements peuvent être affectés à des variables ou être contenus dans d'autres objets. Cependant, remarquez qu'il ne s'agit pas d'objets standards - en particulier, ils ne sont pas copiés lors de l'affectation.

Un objet de fermeture (mode "fonction") contiendra l'environnement dans lequel il est créé dans le cadre de sa définition (par défaut. L'environnement peut être manipulé en utilisant l'environnement <-).
Lorsque la fonction est ensuite appelée, son environnement d'évaluation est créé avec l'environnement de fermeture comme enceinte. Notez qu'il ne s'agit pas nécessairement de l'environnement de l'appelant !

Ainsi, lorsqu'une variable est demandée à l'intérieur d'une fonction, elle est d'abord recherchée dans l'environnement d'évaluation, puis dans l'enceinte, l'enceinte de l'enceinte, etc. ; une fois l'environnement global ou l'environnement d'un package atteint, la recherche continue sur le chemin de recherche jusqu'à l'environnement du package de base. Si la variable n'y est pas trouvée, la recherche se poursuivra à côté de l'environnement vide et échouera.

3.5.3 La pile d'appels

Chaque fois qu'une fonction est invoquée, un nouveau cadre d'évaluation est créé. À tout moment du calcul, les environnements actuellement actifs sont accessibles via la pile d'appels. Chaque fois qu'une fonction est invoquée, une construction spéciale appelée contexte est créée en interne et placée sur une liste de contextes. Lorsqu'une fonction a fini d'évaluer, son contexte est supprimé de la pile d'appels.

Rendre disponibles les variables définies plus haut dans la pile d'appels est appelé portée dynamique. La liaison d'une variable est ensuite déterminée par la définition la plus récente (dans le temps) de la variable.
Cela contredit les règles de portée par défaut dans R, qui utilisent les liaisons dans l'environnement dans lequel la fonction a été définie (portée lexicale). Certaines fonctions, en particulier celles qui utilisent et manipulent des formules de modèle, doivent simuler une portée dynamique en accédant directement à la pile d'appels.

L'accès à la pile d'appels est fourni via une famille de fonctions dont les noms commencent par « sys ». Ils sont brièvement énumérés ci-dessous.

sys.call Récupère l'appel pour le contexte spécifié.

sys.frame
Obtenez le cadre d'évaluation pour le contexte spécifié.

sys.nframe
Obtenez le cadre d'environnement pour tous les contextes actifs.

fonction système
Récupère la fonction invoquée dans le contexte spécifié.

sys.parent
Récupère le parent de l'invocation de fonction actuelle.

appels système
Obtenez les appels pour tous les contextes actifs.

sys.frames
Obtenez les cadres d'évaluation pour tous les contextes actifs.

sys.parents
Obtenez les étiquettes numériques pour tous les contextes actifs.

sys.on.exit Définit
une fonction à exécuter lorsque le contexte spécifié est quitté.

sys.status
Appelle sys.frames, sys.parents et sys.calls.

parent.frame
Obtenez le cadre d'évaluation pour le contexte parent spécifié.

3.5.4 Chemin de recherche En

plus de la structure de l'environnement d'évaluation, R dispose d'un chemin de recherche d'environnements dans lesquels sont recherchées des variables introuvables ailleurs. Ceci est utilisé pour deux choses : les packages de fonctions et les données utilisateur attachées.

Le premier élément du chemin de recherche est l'environnement global et le dernier est le package de base.
Un environnement de chargement automatique est utilisé pour contenir des objets proxy qui peuvent être chargés à la demande.
D'autres environnements sont insérés dans le chemin à l'aide d'une pièce jointe ou d'une bibliothèque.

Les packages qui ont un espace de noms ont un chemin de recherche différent. Lorsqu'une recherche d'un objet R est lancée à partir d'un objet dans un tel package, le package lui-même est recherché en premier, puis ses importations, puis l'espace de noms de base et enfin l'environnement global et le reste du chemin de recherche normal. L'effet est que les références à d'autres objets dans le même package seront résolues dans le package et que les objets ne pourront pas être masqués par des objets du même nom dans l'environnement global ou dans d'autres packages.

4 fonctions

4.1 Fonctions d'écriture

Bien que R puisse être très utile en tant qu'outil d'analyse de données, la plupart des utilisateurs se retrouvent très vite à vouloir écrire leurs propres fonctions. C'est l'un des réels avantages de R. Les utilisateurs peuvent le programmer et, s'ils le souhaitent, modifier les fonctions au niveau du système par des fonctions qu'ils jugent plus appropriées.

R fournit également des fonctionnalités qui facilitent la documentation de toutes les fonctions que vous avez créées. Voir la section « Rédaction de la documentation R » dans Rédaction d'extensions R.

4.1.1 Syntaxe et exemples La syntaxe pour

écrire une fonction est le corps de la fonction

(arglist)

Le premier composant de la déclaration de fonction est le mot-clé function qui indique à R que vous souhaitez créer une fonction.

Une liste d'arguments est une liste d'arguments formels séparés par des virgules. Un argument formel peut être un symbole, une déclaration de la forme « symbole = expression », ou l'argument formel spécial....

Le corps peut être n'importe quelle expression R valide. Généralement, le corps est un ensemble d'expressions contenu entre accolades ("{" et "}").

Généralement, les fonctions sont attribuées aux symboles, mais ce n'est pas obligatoire. La valeur renvoyée par l'appel à la fonction est une fonction. Si aucun nom n'est attribué à cette fonction, on parle de fonction anonyme. Les fonctions anonymes sont le plus souvent utilisées comme arguments pour d'autres fonctions telles que la famille apply ou external.

Voici une fonction simple : echo <- function(x) print(x). Donc echo est une fonction qui prend un seul argument et lorsque echo est invoqué, elle imprime son argument.

4.1.2 Arguments Les

arguments formels de la fonction définissent les variables dont les valeurs seront fournies au moment où la fonction est invoquée. Les noms de ces arguments peuvent être utilisés dans le corps de la fonction où ils obtiennent la valeur fournie au moment de l'invocation de la fonction.

Les valeurs par défaut des arguments peuvent être spécifiées en utilisant la forme spéciale « nom = expression ». Dans ce cas, si l'utilisateur ne spécifie pas de valeur pour l'argument lorsque la fonction est invoquée, l'expression sera associée au symbole correspondant. Lorsqu'une valeur est nécessaire, l'expression est évaluée dans le cadre d'évaluation de la fonction.

Les comportements par défaut peuvent également être spécifiés en utilisant la fonction manquante. Lorsque manquant est appelé avec le nom d'un argument formel, il renvoie TRUE si l'argument formel ne correspond à aucun argument réel et n'a pas été modifié ultérieurement dans le corps de la fonction. Un argument manquant aura ainsi sa valeur par défaut, le cas échéant. La fonction manquante ne force pas l'évaluation de l'argument.

Le type spécial d'argument ... peut contenir n'importe quel nombre d'arguments fournis. Il est utilisé à diverses fins. Il vous permet d'écrire une fonction qui prend un nombre arbitraire d'arguments. Il peut être utilisé pour absorber certains arguments dans une fonction intermédiaire qui pourront ensuite être extraits par des fonctions appelées ultérieurement.

4.2 Fonctions en tant qu'objets Les fonctions

sont des objets de première classe dans R. Elles peuvent être utilisées partout où un objet R est requis. En particulier, ils peuvent être passés en arguments aux fonctions et renvoyés en tant que valeurs par les fonctions.
Voir Section 2.1.5 [Objets fonction], page 4, pour les détails.

4.3 Évaluation

4.3.1 Environnement d'évaluation

Lorsqu'une fonction est appelée ou invoquée, un nouveau cadre d'évaluation est créé. Dans ce cadre, les arguments formels sont mis en correspondance avec les arguments fournis selon les règles données dans la Section 4.3.2 [Correspondance d'arguments], page 23. Les instructions dans le corps de la fonction sont évaluées séquentiellement dans ce cadre d'environnement.

Le cadre englobant du cadre d'évaluation est le cadre d'environnement associé à la fonction invoquée. Cela peut être différent de S. Bien que de nombreuses fonctions aient .GlobalEnv comme environnement, cela ne doit pas nécessairement être vrai et les fonctions définies dans les packages avec des espaces de noms (normalement) ont l'espace de noms du package comme environnement.

4.3.2 Correspondance d'arguments Cette

sous-section s'applique aux fermetures mais pas aux fonctions primitives. Ces derniers ignorent généralement les balises et effectuent une correspondance de position, mais leurs pages d'aide doivent être consultées pour les exceptions, notamment log, round, signif, rep et seq.int.

La première chose qui se produit dans une évaluation de fonction est la correspondance entre le formel et le réel. ou des arguments fournis. Cela se fait par un processus en trois étapes :

1. Correspondance exacte sur les balises. Pour chaque argument nommé fourni, la liste des arguments formels est recherchée pour un élément dont le nom correspond exactement. C'est une erreur de faire correspondre le même argument formel à plusieurs réels ou vice versa.
2. Correspondance partielle sur les balises. Chaque argument fourni nommé restant est comparé aux arguments formels restants en utilisant une correspondance partielle. Si le nom de l'argument fourni correspond exactement à la première partie d'un argument formel, alors les deux arguments sont considérés comme correspondant. C'est une erreur d'avoir plusieurs correspondances partielles. Notez que si f <- function(fumble, fooey) fbody, alors f(f = 1, fo = 2) est illégal, même si le 2ème argument réel ne correspond qu'à fooey. f(f = 1, fooey = 2) est cependant légal puisque le deuxième argument correspond exactement et est retiré de la considération pour la correspondance partielle. Si les arguments formels contiennent ... alors la correspondance partielle n'est appliquée qu'aux arguments qui la précèdent.
3. Correspondance de position. Tous les arguments formels sans correspondance sont liés aux arguments fournis sans nom, dans l'ordre. S'il y a un argument ..., il reprendra les arguments restants, étiquetés ou non.

Si des arguments ne correspondent pas, une erreur est déclarée.

La correspondance des arguments est complétée par les fonctions match.arg, match.call et match.fun. L'accès à l'algorithme de correspondance partielle utilisé par R se fait via pmatch.

4.3.3 Évaluation des arguments L'une des

choses les plus importantes à savoir sur l'évaluation des arguments d'une fonction est que les arguments fournis et les arguments par défaut sont traités différemment. Les arguments fournis à une fonction sont évalués dans le cadre d'évaluation de la fonction appelante. Les arguments par défaut d'une fonction sont évalués dans le cadre d'évaluation de la fonction.

La sémantique de l'appel d'une fonction dans l'argument R est un appel par valeur. En général, les arguments fournis se comportent comme s'il s'agissait de variables locales initialisées avec la valeur fournie et le nom de l'argument formel correspondant. Changer la valeur d'un argument fourni dans une fonction n'affectera pas la valeur de la variable dans le cadre appelant.

R a une forme d'évaluation paresseuse des arguments de fonction. Les arguments ne sont évalués que lorsque cela est nécessaire. Il est important de comprendre que dans certains cas, l'argument ne sera jamais évalué. Ainsi, il est de mauvais style d'utiliser des arguments dans des fonctions pour provoquer des effets secondaires. Alors qu'en C, il est courant de

utilisez le formulaire foo(x = y) pour invoquer foo avec la valeur de y et simultanément pour attribuer la valeur de y à x, ce même style ne doit pas être utilisé dans R. Il n'y a aucune garantie que l'argument sera un jour évalué et la cession ne peut donc pas avoir lieu.

Il convient également de noter que l'effet de foo(x <- y) si l'argument est évalué est de changer la valeur de x dans l'environnement appelant et non dans l'environnement d'évaluation de foo.

Il est possible d'accéder aux expressions réelles (et non par défaut) utilisées comme arguments à l'intérieur de la fonction. Le mécanisme est mis en œuvre via des promesses. Lorsqu'une fonction est évaluée, l'expression réelle utilisée comme argument est stockée dans la promesse avec un pointeur vers l'environnement à partir duquel la fonction a été appelée. Lorsque (si) l'argument est évalué, l'expression stockée est évaluée dans l'environnement à partir duquel la fonction a été appelée. Étant donné que seul un pointeur vers l'environnement est utilisé, toutes les modifications apportées à cet environnement seront appliquées lors de cette évaluation. La valeur résultante est alors également stockée à un endroit distinct dans la promesse.
Les évaluations suivantes récupèrent cette valeur stockée (une deuxième évaluation n'est pas effectuée). L'accès à l'expression non évaluée est également disponible en utilisant substitut.

Lorsqu'une fonction est appelée, chaque argument formel se voit attribuer une promesse dans l'environnement local de l'appel avec le slot d'expression contenant l'argument réel (s'il existe) et le slot d'environnement contenant l'environnement de l'appelant. Si aucun argument réel pour un argument formel n'est donné dans l'appel et qu'il existe une expression par défaut, elle est affectée de la même manière à l'emplacement d'expression de l'argument formel, mais avec l'environnement défini sur l'environnement local.

Le processus consistant à remplir l'espace de valeur d'une promesse en évaluant le contenu de l'espace d'expression dans l'environnement de la promesse est appelé forcer la promesse. Une promesse ne sera forcée qu'une seule fois, le contenu du slot de valeur étant utilisé directement ultérieurement.

Une promesse est forcée lorsque sa valeur est nécessaire. Cela se produit généralement au sein des fonctions internes, mais une promesse peut également être forcée par une évaluation directe de la promesse elle-même. Ceci est parfois utile lorsqu'une expression par défaut dépend de la valeur d'un autre argument formel ou d'une autre variable dans l'environnement local. Cela se voit dans l'exemple suivant où l'étiquette isolée garantit que l'étiquette est basée sur la valeur de x avant qu'elle ne soit modifiée dans la ligne suivante.

```
fonction (x, étiquette = deparse (x)) { étiquette

    x <- x + 1
    Imprimer l'étiquette)
}
```

Le créneau d'expression d'une promesse peut lui-même impliquer d'autres promesses. Cela se produit chaque fois qu'un argument non évalué est transmis comme argument à une autre fonction. Lors du forçage d'une promesse, d'autres promesses dans son expression seront également forcées de manière récursive au fur et à mesure de leur évaluation.

4.3.4 Portée La

portée ou les règles de portée sont simplement l'ensemble des règles utilisées par l'évaluateur pour trouver une valeur pour un symbole. Chaque langage informatique possède un ensemble de telles règles. Dans R, les règles sont assez simples mais il existe des mécanismes pour renverser les règles habituelles ou par défaut.

R adhère à un ensemble de règles appelées portée lexicale. Cela signifie que les liaisons de variables en vigueur au moment de la création de l'expression sont utilisées pour fournir des valeurs à tous les symboles non liés dans l'expression.

La plupart des propriétés intéressantes de scope sont impliquées dans les fonctions d'évaluation et nous nous concentrons sur cette question. Un symbole peut être lié ou non lié. Tous les arguments formels d'une fonction fournissent des symboles liés dans le corps de la fonction. Tous les autres symboles présents dans le corps de la fonction sont soit des variables locales, soit des variables non liées. Une variable locale est une variable définie dans la fonction. Étant donné que R n'a pas de définition formelle des variables, elles sont simplement utilisées selon les besoins. Il peut être difficile de déterminer si une variable est locale ou non. Locale

les variables doivent d'abord être définies, cela se fait généralement en les plaçant sur le côté gauche d'une affectation.

Pendant le processus d'évaluation, si un symbole non lié est détecté, R tente de lui trouver une valeur. Les règles de cadrage déterminent le déroulement de ce processus. Dans R, l'environnement de la fonction est recherché en premier, puis son enceinte et ainsi de suite jusqu'à atteindre l'environnement global.

L'environnement global est en tête d'une liste de recherche d'environnements qui sont recherchés séquentiellement pour un symbole correspondant. La valeur de la première correspondance est ensuite utilisée.

Lorsque cet ensemble de règles est combiné avec le fait que les fonctions peuvent être renvoyées sous forme de valeurs à partir d'autres fonctions, on obtient des propriétés plutôt agréables, mais à première vue particulières.

Un exemple simple : f

```
<- function() { y <- 10 g <-
        function(x)
        x + y return(g)

}
h <- f() h(3)
```

Une question plutôt intéressante est de savoir ce qui se passe lorsque h est évalué. Lorsqu'un corps de fonction est évalué, il n'y a aucun problème pour déterminer les valeurs des variables locales ou des variables liées.
Les règles de portée déterminent comment le langage trouvera les valeurs des variables non liées.

Lorsque h(3) est évalué on voit que son corps est celui de g. Dans ce corps, x est lié à l'argument formel et y n'est pas lié. Dans un langage à portée lexicale, x sera associé à la valeur 3 et y à la valeur 10 locale à f donc h(3) devrait renvoyer la valeur 13. Dans R, c'est effectivement ce qui se passe.

Dans S, à cause des différentes règles de portée, on obtiendra une erreur indiquant que y n'est pas trouvé, sauf s'il y a une variable y dans votre espace de travail, auquel cas sa valeur sera utilisée.

5 Programmation orientée objet

La programmation orientée objet est un style de programmation devenu populaire ces dernières années. Une grande partie de sa popularité vient du fait qu'il facilite l'écriture et la maintenance de systèmes complexes. Cela se fait par le biais de plusieurs mécanismes différents.

Les concepts de classe et de méthodes sont au cœur de tout langage orienté objet. Une classe est une définition d'un objet. En règle générale, une classe contient plusieurs emplacements utilisés pour contenir des informations spécifiques à la classe. Un objet dans le langage doit être une instance d'une classe. La programmation est basée sur des objets ou des instances de classes.

Les calculs sont effectués via des méthodes. Les méthodes sont essentiellement des fonctions spécialisées pour effectuer des calculs spécifiques sur des objets, généralement d'une classe spécifique. C'est ce qui rend le langage orienté objet. En R, des fonctions génériques sont utilisées pour déterminer la méthode appropriée. La fonction générique est chargée de déterminer la classe de ses arguments et utilise ces informations pour sélectionner la méthode appropriée.

Une autre caractéristique de la plupart des langages orientés objet est le concept d'héritage. Dans la plupart des problèmes de programmation, de nombreux objets sont généralement liés les uns aux autres. La programmation est considérablement simplifiée si certains composants peuvent être réutilisés.

Si une classe hérite d'une autre classe, elle obtient généralement tous les emplacements de la classe parent et peut l'étendre en ajoutant de nouveaux emplacements. Lors de la répartition des méthodes (via les fonctions génériques) si une méthode pour la classe n'existe pas alors une méthode pour le parent est recherchée.

Dans ce chapitre, nous discutons de la manière dont cette stratégie générale a été implémentée dans R et discutons de certaines des limites de la conception actuelle. L'un des avantages apportés par la plupart des systèmes d'objets est une plus grande cohérence. Ceci est réalisé via les règles qui sont vérifiées par le compilateur ou l'interpréteur. Malheureusement, en raison de la manière dont le système d'objets est incorporé dans R, cet avantage n'est pas obtenu. Les utilisateurs sont priés d'utiliser le système d'objets de manière simple. Bien qu'il soit possible de réaliser des exploits plutôt intéressants, ceux-ci ont tendance à conduire à un code obscurci et peuvent dépendre de détails d'implémentation qui ne seront pas reportés.

La plus grande utilisation de la programmation orientée objet dans R se fait via les méthodes d'impression, les méthodes récapitulatives et les méthodes de tracé. Ces méthodes nous permettent d'avoir un appel de fonction générique, par exemple plot, qui répartit le type de son argument et appelle une fonction de traçage spécifique aux données fournies.

Afin de clarifier les concepts, nous envisagerons la mise en œuvre d'un petit système conçu pour enseigner aux étudiants les probabilités. Dans ce système, les objets sont des fonctions de probabilité et les méthodes que nous considérerons sont des méthodes de recherche de moments et de tracé. Les probabilités peuvent toujours être représentées en termes de fonction de distribution cumulative, mais peuvent souvent être représentées d'autres manières. Par exemple comme densité, lorsqu'elle existe ou comme fonction génératrice de moment lorsqu'elle existe.

5.1 Définition

Plutôt que d'avoir un système orienté objet à part entière, R dispose d'un système de classes et d'un mécanisme de répartition basé sur la classe d'un objet. Le mécanisme de répartition du code interprété repose sur quatre objets spéciaux stockés dans le cadre d'évaluation. Ces objets spéciaux sont .Generic, .Class, .Method et .Group. Il existe un mécanisme de répartition distinct utilisé pour les fonctions et types internes qui seront abordés ailleurs.

Le système de classes est facilité par l'attribut class. Cet attribut est un vecteur de caractères des noms de classe. Donc, pour créer un objet de classe "foo", il suffit d'attacher un attribut de classe contenant la chaîne ""foo"'. Ainsi, pratiquement tout peut être transformé en objet de classe « foo ».

Le système objet utilise des fonctions génériques via deux fonctions de répartition, UseMethod et NextMethod. L'utilisation typique du système objet est de commencer par appeler une fonction générique.
Il s'agit généralement d'une fonction très simple et constituée d'une seule ligne de code. La moyenne de la fonction système est précisément une telle fonction,

```
> signifier
fonction (x, ...)
UseMethod("moyenne")
```

Lorsque la moyenne est appelée, elle peut avoir n'importe quel nombre d'arguments mais son premier argument est spécial et la classe de ce premier argument est utilisée pour déterminer quelle méthode doit être appelée. La variable .Class est définie sur l'attribut de classe de x, .Generic est définie sur la chaîne "mean" et une recherche est effectuée pour la méthode correcte à invoquer. Les attributs de classe de tout autre argument à signifier sont ignorés.

Supposons que x ait un attribut de classe contenant "foo" et "bar", dans cet ordre. Ensuite, R rechercherait d'abord une fonction appelée Mean.foo et s'il n'en trouvait pas, il rechercherait alors une fonction Mean.bar et si cette recherche échouait également, une recherche finale pour Mean.default serait effectuée. Si la dernière recherche échoue, R signale une erreur. C'est une bonne idée de toujours écrire une méthode par défaut. Notez que les fonctions Mean.foo etc. sont appelées, dans ce contexte, des méthodes.

NextMethod fournit un autre mécanisme de répartition. Une fonction peut contenir un appel à NextMethod n'importe où. La détermination de la méthode qui doit ensuite être invoquée repose principalement sur les valeurs actuelles de .Class et .Generic. Ceci est quelque peu problématique puisque la méthode est en réalité une fonction ordinaire et que les utilisateurs peuvent l'appeler directement. S'ils le font, il n'y aura aucune valeur pour .Generic ou .Class.

Si une méthode est invoquée directement et qu'elle contient un appel à NextMethod, alors le premier argument de NextMethod est utilisé pour déterminer la fonction générique. Une erreur est signalée si cet argument n'a pas été fourni ; c'est donc une bonne idée de toujours fournir cet argument.

Dans le cas où une méthode est invoquée directement, l'attribut de classe du premier argument du La méthode est utilisée comme valeur de .Class.

Les méthodes elles-mêmes utilisent NextMethod pour fournir une forme d'héritage. Généralement, une méthode spécifique effectue quelques opérations pour configurer les données, puis appelle la méthode appropriée suivante via un appel à NextMethod.

Considérez l'exemple simple suivant. Un point dans l'espace euclidien bidimensionnel peut être spécifié par ses coordonnées cartésiennes (xy) ou polaires (r-thêta). Par conséquent, pour stocker des informations sur l'emplacement du point, nous pourrions définir deux classes, "xypoint" et "rthetapoint". Toutes les structures de données « xypoint » sont des listes avec un composant x et un composant y. Tous les objets 'rthetapoint' sont des listes avec un composant r et un composant thêta.

Supposons maintenant que nous souhaitions obtenir la position x de l'un ou l'autre type d'objet. Cela peut facilement être obtenu grâce à des fonctions génériques. Nous définissons la fonction générique xpos comme suit. xpos <-

```
fonction(x, ...)
    UtiliserMéthode("xpos")
```

Nous pouvons maintenant définir des méthodes :

```
xpos.xypoint <- function(x) x$x xpos.rthetapoint
<- function(x) x$r * cos(x$theta)
```

L'utilisateur appelle simplement la fonction xpos avec l'une ou l'autre représentation comme argument. Le La méthode de répartition interne trouve la classe de l'objet et appelle les méthodes appropriées.

Il est assez simple d'ajouter d'autres représentations. Il n'est pas nécessaire d'écrire une nouvelle fonction générique uniquement les méthodes. Cela facilite l'ajout aux systèmes existants puisque l'utilisateur est uniquement responsable du traitement de la nouvelle représentation et non d'aucune des représentations existantes.

L'essentiel des utilisations de cette méthodologie consiste à fournir une impression spécialisée pour des objets de différents types ; il existe environ 40 méthodes d'impression.

5.2 Héritage

L'attribut de classe d'un objet peut avoir plusieurs éléments. Lorsqu'une fonction générique est appelée, le premier héritage est principalement géré via NextMethod. NextMethod détermine la méthode en cours d'évaluation, trouve la classe suivante à partir de la classe suivante.

FIXME : il manque quelque chose ici

5.3 Répartition des méthodes

Les fonctions génériques doivent consister en une seule instruction. Ils doivent généralement être de la forme foo <- function(x, ...) UseMethod("foo", x). Lorsque UseMethod est appelé, il détermine la méthode appropriée, puis cette méthode est invoquée avec les mêmes arguments, dans le même ordre que l'appel au générique, comme si l'appel avait été effectué directement à la méthode.

Afin de déterminer la méthode correcte, l'attribut de classe du premier argument du générique est obtenu et utilisé pour trouver la méthode correcte. Le nom de la fonction générique est combiné avec le premier élément de l'attribut class dans le formulaire generic.class et une fonction portant ce nom est recherchée. Si la fonction est trouvée, elle est utilisée. Si aucune fonction de ce type n'est trouvée, alors le deuxième élément de l'attribut class est utilisé, et ainsi de suite jusqu'à ce que tous les éléments de l'attribut class soient épuisés. Si aucune méthode n'a été trouvée à ce stade, la méthode generic.default est utilisée. Si le premier argument de la fonction générique n'a pas d'attribut de classe, alors generic.default est utilisé. Depuis l'introduction des espaces de noms, les méthodes peuvent ne pas être accessibles par leurs noms (c'est-à-dire que get("generic.class") peut échouer), mais elles seront accessibles par getS3method("generic","class").

Tout objet peut avoir un attribut de classe. Cet attribut peut avoir n'importe quel nombre d'éléments. Chacun d'eux est une chaîne qui définit une classe. Lorsqu'une fonction générique est invoquée, la classe de son premier argument est examinée.

5.4 Méthode d'utilisation

UseMethod est une fonction spéciale et elle se comporte différemment des autres appels de fonction. La syntaxe d'un appel à celui-ci est UseMethod(generic, object), où generic est le nom de la fonction générique, object est l'objet utilisé pour déterminer quelle méthode doit être choisie. UseMethod ne peut être appelé qu'à partir du corps d'une fonction.

UseMethod modifie le modèle d'évaluation de deux manières. Premièrement, lorsqu'il est invoqué, il détermine la prochaine méthode (fonction) à appeler. Il invoque ensuite cette fonction en utilisant l'environnement d'évaluation actuel ; ce processus sera décrit sous peu. La deuxième façon dont UseMethod modifie l'environnement d'évaluation est qu'il ne rend pas le contrôle à la fonction appelante.
Cela signifie que toutes les instructions après un appel à UseMethod sont garanties de ne pas être exécutées.

Lorsque UseMethod est invoquée, la fonction générique est la valeur spécifiée dans l'appel à UseMethod. L'objet sur lequel distribuer est soit le deuxième argument fourni, soit le premier argument de la fonction actuelle. La classe de l'argument est déterminée et le premier élément de celui-ci est combiné avec le nom du générique pour déterminer la méthode appropriée. Ainsi, si le générique avait le nom foo et que la classe de l'objet est "bar", alors R recherchera une méthode nommée foo.bar. Si aucune méthode de ce type n'existe, le mécanisme d'héritage décrit ci-dessus est utilisé pour localiser une méthode appropriée.

Une fois qu'une méthode a été déterminée, R l'invoque d'une manière spéciale. Plutôt que de créer un nouvel environnement d'évaluation, R utilise l'environnement de l'appel de fonction actuel (l'appel au générique). Toutes les affectations ou évaluations effectuées avant l'appel à UseMethod seront

en effet. Les arguments utilisés lors de l'appel au générique sont réapparus aux arguments formels de la méthode sélectionnée.

Lorsque la méthode est invoquée, elle est appelée avec des arguments qui ont le même nombre et portent les mêmes noms que lors de l'appel au générique. Ils sont mis en correspondance avec les arguments de la méthode selon les règles R standard pour la correspondance des arguments. Cependant, l'objet, c'est-à-dire le premier argument, a été évalué.

L'appel à UseMethod a pour effet de placer certains objets spéciaux dans le cadre d'évaluation.
Ce sont .Class, .Generic et .Method. Ces objets spéciaux sont utilisés par R pour gérer la répartition et l'héritage des méthodes. .Class est la classe de l'objet, .Generic est le nom de la fonction générique et .Method est le nom de la méthode actuellement invoquée. Si la méthode a été invoquée via l'une des interfaces internes, il peut également y avoir un objet appelé .Group. Ceci sera décrit dans la Section Section 5.6 [Méthodes de groupe], page 30. Après l'appel initial à UseMethod, ces variables spéciales, et non l'objet lui-même, contrôlent la sélection des méthodes suivantes.

Le corps de la méthode est ensuite évalué de manière standard. En particulier, la recherche de variables dans le corps suit les règles de la méthode. Donc, si la méthode a un environnement associé, celui-ci est utilisé. En effet nous avons remplacé l'appel au générique par un appel à la méthode. Toutes les affectations locales dans le cadre du générique seront reportées dans l'appel à la méthode. L'utilisation de cette fonctionnalité est déconseillée. Il est important de réaliser que le contrôle ne reviendra jamais au générique et que par conséquent, les expressions après un appel à UseMethod ne seront jamais exécutées.

Tous les arguments du générique qui ont été évalués avant l'appel à UseMethod restent
évalué.

Si le premier argument de UseMethod n'est pas fourni, il est supposé être le nom de la fonction actuelle. Si deux arguments sont fournis à UseMethod, le premier est le nom de la méthode et le second est supposé être l'objet sur lequel sera distribué. Elle est évaluée afin que la méthode requise puisse être déterminée. Dans ce cas, le premier argument de l'appel au générique n'est pas évalué et est ignoré. Il n'existe aucun moyen de modifier les autres arguments dans l'appel à la méthode ; ceux-ci restent tels qu'ils étaient dans l'appel au générique. Cela contraste avec NextMethod où les arguments de l'appel à la méthode suivante peuvent être modifiés.

5.5 Méthode suivante

NextMethod est utilisé pour fournir un mécanisme d'héritage simple.

Les méthodes invoquées suite à un appel à NextMethod se comportent comme si elles avaient été invoquées à partir de la méthode précédente. Les arguments de la méthode héritée sont dans le même ordre et portent les mêmes noms que l'appel à la méthode actuelle. Cela signifie qu'ils sont les mêmes que pour l'appel au générique. Cependant, les expressions des arguments sont les noms des arguments formels correspondants de la méthode actuelle. Ainsi, les arguments auront des valeurs qui correspondent à leur valeur au moment où NextMethod a été invoqué.

Les arguments non évalués restent non évalués. Les arguments manquants restent manquants.

La syntaxe d'un appel à NextMethod est NextMethod(generic, object, ...). Si le générique n'est pas fourni, la valeur de .Generic est utilisée. Si l'objet n'est pas fourni, le premier argument de l'appel à la méthode actuelle est utilisé. Les valeurs de l'argument ... sont utilisées pour modifier les arguments de la méthode suivante.

Il est important de réaliser que le choix de la méthode suivante dépend des valeurs actuelles de .Generic et .Class et non de l'objet. Ainsi, changer l'objet lors d'un appel à NextMethod affecte les arguments reçus par la méthode suivante mais n'affecte pas le choix de la méthode suivante.

Les méthodes peuvent être appelées directement. Si c'est le cas, il n'y aura pas de .Generic, .Class ou .Method. Dans ce cas, l'argument générique de NextMethod doit être spécifié. La valeur de .Class est considérée comme l'attribut de classe de l'objet qui est le premier argument de la fonction actuelle. La valeur de .Method est le nom de la fonction actuelle. Ces choix de valeurs par défaut garantissent que le comportement d'une méthode ne change pas selon qu'elle est appelée directement ou via un appel à un générique.

Un sujet de discussion est le comportement de l'argument ... de NextMethod. Le blanc
Le livre décrit le comportement comme suit :

- les arguments nommés remplacent les arguments correspondants dans l'appel à la méthode courante.
Les arguments sans nom sont placés au début de la liste des arguments.

Ce que j'aimerais faire c'est :

-faire d'abord la correspondance des arguments pour NextMethod ; -si l'objet ou le générique sont modifiés correctement -d'abord si un élément de liste nommé correspond à un argument (nommé ou non), la valeur de la liste remplace la valeur de l'argument. - le premier élément de liste

sans nom Valeurs à rechercher : Classe : vient en premier de .Class, en second du premier argument du méthode et dernier de l'objet spécifié dans l'appel à NextMethod

Générique : vient en premier de .Generic, si rien alors du premier argument de la méthode et s'il manque toujours lors de l'appel à NextMethod

Méthode : cela devrait simplement être le nom de la fonction actuelle.

5.6 Méthodes de groupe Pour

plusieurs types de fonctions internes, R fournit un mécanisme de répartition pour les opérateurs. Cela signifie que les opérateurs tels que == ou < peuvent voir leur comportement modifié pour les membres de classes spéciales. Les fonctions et opérateurs ont été regroupés en trois catégories et des méthodes de groupe peuvent être écrites pour chacune de ces catégories. Il n'existe actuellement aucun mécanisme pour ajouter des groupes. Il est possible d'écrire des méthodes spécifiques à n'importe quelle fonction au sein d'un groupe.

Le tableau suivant répertorie les fonctions des différents groupes.

'Mathématiques'	abdos, acos, acosh, asin, asinh, atan, atanh, plafond, cos, cosh, cospi, cumsum, exp, sol, gamma, lgamma, log, log10, rond, signif, sin, sinh, sinpi, tan, tanh, tanpi, tronc
'Résumé'	all, any, max, min, prod, range, sum
'Opérations'	+, -, *, /, ^, < , >, <=, >=, !=, ==, %%, %/%, &, |, !

Pour les opérateurs du groupe Ops, une méthode spéciale est invoquée si les deux opérandes pris ensemble suggèrent une seule méthode. Plus précisément, si les deux opérandes correspondent à la même méthode ou si un opérande correspond à une méthode prioritaire sur celle de l'autre opérande. S'ils ne suggèrent aucune méthode, la méthode par défaut est utilisée. Soit une méthode de groupe, soit une méthode de classe domine si l'autre opérande n'a pas de méthode correspondante. Une méthode de classe domine une méthode de groupe.

Lorsque le groupe est Ops, la variable spéciale .Method est un vecteur de caractères avec deux éléments.
Les éléments de .Method sont définis avec le nom de la méthode si l'argument correspondant est un membre de la classe qui a été utilisée pour déterminer la méthode. Sinon, l'élément correspondant de .Method est défini sur la chaîne de longueur nulle, "".

5.7 Écriture de méthodes Les utilisateurs

peuvent facilement écrire leurs propres méthodes et fonctions génériques. Une fonction générique est simplement une fonction avec un appel à UseMethod. Une méthode est simplement une fonction qui a été invoquée via la répartition de méthode. Cela peut être dû à un appel à UseMethod ou NextMethod.

Il convient de rappeler que les méthodes peuvent être appelées directement. Cela signifie qu'ils peuvent être saisis sans qu'un appel à UseMethod ait été effectué et donc les variables spéciales .Generic, .Class et .Method n'auront pas été instanciées. Dans ce cas, les règles par défaut détaillées ci-dessus seront utilisées pour les déterminer.

L'utilisation la plus courante des fonctions génériques consiste à fournir des méthodes d'impression et de synthèse pour les objets statistiques, généralement le résultat d'un processus d'ajustement de modèle. Pour ce faire, chaque modèle attache un attribut de classe à sa sortie, puis fournit une méthode spéciale qui prend cette sortie et en fournit une version lisible. L'utilisateur n'a alors qu'à se rappeler que l'impression ou le résumé fourniront un bon résultat pour les résultats de toute analyse.

6 L'informatique sur le langage

R appartient à une classe de langages de programmation dans lesquels les sous-programmes ont la capacité de modifier ou de construire d'autres sous-programmes et d'évaluer le résultat en tant que partie intégrante du langage lui-même. Ceci est similaire à Lisp et Scheme et à d'autres langages de type « programmation fonctionnelle », mais contrairement à FORTRAN et à la famille ALGOL. La famille Lisp pousse cette fonctionnalité à l'extrême par le paradigme « tout est une liste » dans lequel il n'y a aucune distinction entre les programmes et les données.

R présente une interface de programmation plus conviviale que Lisp, du moins pour quelqu'un habitué aux formules mathématiques et aux structures de contrôle de type C, mais le moteur ressemble vraiment beaucoup à Lisp. R permet un accès direct aux expressions et fonctions analysées et vous permet de les modifier puis de les exécuter, ou de créer des fonctions entièrement nouvelles à partir de zéro.

Il existe un certain nombre d'applications standards de cette fonctionnalité, telles que le calcul de dérivées analytiques d'expressions ou la génération de fonctions polynomiales à partir d'un vecteur de coefficients. Cependant, il existe aussi des utilisations beaucoup plus fondamentales pour le fonctionnement de la partie interprétée de R. Certaines d'entre elles sont essentielles à la réutilisation de fonctions comme composants dans d'autres fonctions, comme les appels (certes pas très jolis) à model.frame qui sont construits dans plusieurs routines de modélisation et de traçage. D'autres utilisations permettent simplement d'associer des interfaces élégantes à des fonctionnalités utiles. À titre d'exemple, considérons la fonction courbe, qui vous permet de dessiner le graphique d'une fonction donnée sous forme d'expression comme sin(x) ou les fonctionnalités de traçage d'expressions mathématiques.

Dans ce chapitre, nous donnons une introduction à l'ensemble des fonctionnalités disponibles pour l'informatique sur le langage.

6.1 Manipulation directe des objets langage Il existe trois types d'objets langage

disponibles pour la modification, les appels, les expressions et les fonctions. À ce stade, nous allons nous concentrer sur les objets d'appel. On les appelle parfois « expressions non évaluées », bien que cette terminologie prête quelque peu à confusion. La méthode la plus directe pour obtenir un objet d'appel consiste à utiliser quote avec un argument d'expression, par exemple > e1 <- quote(2 + 2) > e2 <- quote(plot(x, y))

Les arguments ne sont pas évalués, le résultat est simplement l'argument analysé. Les objets e1 et e2 peuvent être évalués ultérieurement à l'aide de eval, ou simplement manipulés comme des données. La raison pour laquelle l'objet e2 a le mode "call" est peut-être la plus immédiatement évidente, puisqu'il implique un appel à la fonction plot avec quelques arguments. Cependant, e1 a en réalité exactement la même structure qu'un appel à l'opérateur binaire + avec deux arguments, un fait qui est clairement affiché par ce qui suit

> citation("+"(2, 2)) 2 + 2

Les composants d'un objet d'appel sont accessibles à l'aide d'une syntaxe de type liste et peuvent en fait être converti vers et depuis des listes en utilisant as.list et as.call > e2[[1]]

plot > e2[[2]]

X

> e2[[3]]

et

Lorsque la correspondance d'arguments de mots-clés est utilisée, les mots-clés peuvent être utilisés comme balises de liste :

> e3 <- quote(plot(x = âge, y = poids)) > e3$x

âge

```
> e3$y
poids
```

Tous les composants de l'objet appel ont le mode "nom" dans les exemples précédents. Cela est vrai pour les identifiants dans les appels, mais les composants d'un appel peuvent également être des constantes - qui peuvent être de n'importe quel type, même s'il est préférable que le premier composant soit une fonction pour que l'appel soit évalué avec succès - ou d'autres objets d'appel, correspondant aux sous-expressions. Les objets de nom de mode peuvent être construits à partir de chaînes de caractères en utilisant as.name, on peut donc modifier l'objet e2 comme suit

```
> e2[[1]] <- as.name("+") > e2


x + y
```

Pour illustrer le fait que les sous-expressions sont simplement des composants qui sont eux-mêmes des appels, considérer

```
> e1[[2]] <- e2 > e1


x + y + 2
```

Toutes les parenthèses de regroupement en entrée sont conservées dans les expressions analysées. Ils sont représentés comme un appel de fonction avec un argument, de sorte que 4 - (2 - 2) devient "-"(4, "(" ("-"(2, 2))) en notation préfixe. Dans les évaluations, le ' (' L'opérateur renvoie simplement son argument.

C'est un peu malheureux, mais il n'est pas facile d'écrire une combinaison analyseur/déparseur qui à la fois préserve les entrées de l'utilisateur, les stocke sous une forme minimale et garantit que l'analyse d'une expression analysée renvoie la même expression.

Il se trouve que l'analyseur de R n'est pas parfaitement inversible, pas plus que son déparseur, comme le montrent les exemples suivants

```
> str(quote(c(1,2))) langage
  c(1, 2) > str(c(1,2))


  nombre [1:2] 1 2
> deparse(quote(c(1,2))) [1] "c(1,
 2)" >
deparse(c(1,2)) [1] "c(1,
 2)" > quote("- "(2,
2))
2 - 2
> citation(2 - 2) 2 -
2
```

Les expressions analysées doivent cependant donner une valeur équivalente à l'expression d'origine (jusqu'à l'erreur d'arrondi).

...stockage interne des constructions de contrôle de flux...notez l'incompatibilité Splus...

6.2 Remplacements

Il n'est en effet pas fréquent que l'on veuille modifier les entrailles d'une expression comme dans la section précédente. Le plus souvent, on souhaite simplement accéder à une expression afin de la décomposer et de l'utiliser pour étiqueter des tracés, par exemple. Un exemple de ceci est visible au début de plot.default :

```
xlabel <- if (!missing(x))
      deparse(substitute(x))
```

Cela entraîne l'utilisation ultérieure de la variable ou de l'expression donnée comme argument x à tracer pour étiqueter l'axe des x.

La fonction utilisée pour y parvenir est le substitut qui prend l'expression x et remplace l'expression qui a été transmise via l'argument formel x. Notez que pour que cela se produise, x doit contenir des informations sur l'expression qui crée sa valeur. Ceci est lié au schéma d'évaluation paresseux de R (voir Section 2.1.8 [Objets Promise], page 5). Un argument formel est en réalité une promesse, un objet avec trois emplacements, un pour l'expression qui le définit, un pour l'environnement dans lequel évaluer cette expression et un pour la valeur de cette expression une fois évaluée. substitut reconnaîtra une variable promise et remplacera la valeur de son emplacement d'expression. Si le remplacement est invoqué à l'intérieur d'une fonction, les variables locales de la fonction sont également soumises à la substitution.

L'argument pour substituer ne doit pas nécessairement être un simple identifiant, il peut s'agir d'une expression impliquant plusieurs variables et la substitution se produira pour chacune d'elles. De plus, substitut a un argument supplémentaire qui peut être un environnement ou une liste dans laquelle les variables sont recherchées. Par exemple:

```
> remplacer (a + b, liste (a = 1, b = citation (x)))
1 + x
```

Notez que la citation était nécessaire pour remplacer le x. Ce type de construction s'avère pratique en relation avec les fonctionnalités permettant de mettre des expressions mathématiques dans des graphiques, comme le montre le cas suivant

```
> plot(0) >
for (i in 1:4) + text(1,
0.2 * i, substitut(x[ix] == y,
+                  list(ix = i, y = pnorm(i))))
```

Il est important de réaliser que les substitutions sont purement lexicales ; il n'y a aucune vérification que les objets d'appel résultants ont un sens s'ils sont évalués. substitut(x <- x + 1, list(x = 2)) renverra volontiers 2 <- 2 + 1. Cependant, certaines parties de R établissent leurs propres règles pour ce qui a du sens et ce qui ne l'est pas et pourraient en fait avoir un utiliser pour de telles expressions mal formées. Par exemple, l'utilisation de la fonctionnalité « mathématiques dans les graphiques » implique souvent des constructions syntaxiquement correctes, mais qui n'auraient aucun sens à évaluer, comme « {}>=40*"ans" ».

Substitute n'évaluera pas son premier argument. Cela nous amène au casse-tête de savoir comment effectuer des substitutions sur un objet contenu dans une variable. La solution est d'utiliser à nouveau substitut, comme ceci > expr <- quote(x

```
+ y) > substitut(substitute(e,
list(x = 3)), list(e = expr)) substitut(x + y, list (x = 3)) > eval(substitute(substitute(e,
list(x = 3)), list(e = expr))) 3 + y
```

Les règles exactes pour les substitutions sont les suivantes : chaque symbole de l'arbre d'analyse du premier est comparé au deuxième argument, qui peut être une liste balisée ou un cadre d'environnement. S'il s'agit d'un simple objet local, sa valeur est insérée, sauf si elle correspond à l'environnement global.
S'il s'agit d'une promesse (généralement un argument de fonction), l'expression de promesse est remplacée. Si le symbole ne correspond pas, il reste inchangé. L'exception spéciale pour les remplacements au plus haut niveau est certes particulière. Il a été hérité de S et la raison est très probablement qu'il n'y a aucun contrôle sur les variables qui pourraient être liées à ce niveau, de sorte qu'il serait préférable de simplement faire en sorte que le substitut agisse comme une citation.

La règle de substitution de promesse est légèrement différente de celle de S si la variable locale est modifiée avant l'utilisation du substitut. R utilisera alors la nouvelle valeur de la variable, tandis que S utilisera inconditionnellement l'expression de l'argument – à moins qu'il ne s'agisse d'une constante, ce qui a la curieuse conséquence que f((1)) peut être très différent de f(1) dans S. La règle R est considérablement plus propre, même si elle a des conséquences en matière d'évaluation paresseuse qui surprennent certains. Considérer

```
logplot <- function(y, ylab = deparse(substitute(y))) { y <- log(y) plot(y, ylab =
    ylab)

}
```

Cela semble simple, mais on découvrira que l'étiquette y devient une vilaine expression c(...). Cela se produit parce que les règles de l'évaluation paresseuse entraînent l'évaluation de l'expression ylab après la modification de y. La solution est de forcer ylab à être évalué en premier, c'est-à-dire logplot <- function(y, ylab =

```
deparse(substitute(y))) {
        ylab
        y <- log(y)
        plot(y, ylab = ylab)
}
```

Notez qu'il ne faut pas utiliser eval(ylab) dans cette situation. Si ylab est un objet langage ou expression, cela entraînerait également l'évaluation de l'objet, ce qui ne serait pas du tout souhaitable si une expression mathématique comme quote(log[e](y)) était transmise.

Une variante du substitut est bquote, qui est utilisée pour remplacer certaines sous-expressions par leur valeurs. L'exemple ci-dessus > plot(0)

```
> for (i in
1:4) + text(1, 0.2 * i,
substitut(x[ix] == y, list(ix =
+             i, y = pnorm(i)) ))
```

pourrait être écrit de manière plus compacte

```
sous la
forme plot(0) for(i
    in 1:4) text(1, 0.2*i, bquote( x[.(i)] == .(pnorm(i)) ))
```

L'expression est citée à l'exception du contenu des sous-expressions .(), qui sont remplacés par leurs valeurs. Il existe un argument facultatif pour calculer les valeurs dans un environnement différent. La syntaxe de bquote est empruntée à la macro LISP backquote.

6.3 En savoir plus sur l'évaluation

La fonction eval a été introduite plus tôt dans ce chapitre comme moyen d'évaluer les objets d'appel. Cependant, ce n’est pas toute l’histoire. Il est également possible de préciser l'environnement dans lequel l'évaluation doit se dérouler. Par défaut, il s'agit du cadre d'évaluation à partir duquel eval est appelé, mais il faut très souvent le définir sur autre chose.

Très souvent, le référentiel d'évaluation pertinent est celui du parent du référentiel courant (cf. ???). En particulier, lorsque l'objet à évaluer est le résultat d'une opération de substitution des arguments de la fonction, il contiendra des variables qui n'ont de sens que pour l'appelant (notez qu'il n'y a aucune raison de s'attendre à ce que les variables de l'appelant soient dans le langage lexical). portée de l’appelé). Étant donné que l'évaluation dans le frame parent se produit fréquemment, une fonction eval.parent existe comme un raccourci pour eval(expr, sys.frame(sys.parent())).

Un autre cas fréquent est l’évaluation dans une liste ou un bloc de données. Par exemple, cela se produit en relation avec la fonction model.frame lorsqu'un argument data est donné. Généralement, les termes de la formule du modèle doivent être évalués dans les données, mais ils peuvent parfois également contenir des références à des éléments dans l'appelant de model.frame. Ceci est parfois utile dans le cadre d’études de simulation. Ainsi, pour cela, il faut non seulement évaluer une expression dans une liste, mais également spécifier une enceinte dans laquelle la recherche se poursuit si la variable n'est pas dans la liste. L’appel a donc la forme

```
eval(expr, données, sys.frame(sys.parent()))
```

Notez que l'évaluation dans un environnement donné peut en réalité modifier cet environnement, la plupart évidemment dans les cas impliquant l'opérateur d'affectation, comme

```
eval(quote(total <- 0), Environment(robert$balance)) # rob Rob
```

Cela est également vrai lors de l'évaluation dans des listes, mais la liste originale ne change pas car on travaille réellement sur une copie.

6.4 Évaluation des objets d'expression Les objets de mode

"expression" sont définis dans la Section 2.1.4 [Objets d'expression], page 4. Ils sont très similaires aux listes d'objets d'appel.

```
> ex <- expression(2 + 2, 3 + 4) > ex[[1]] 2 + 2 >
ex[[2]] 3 + 4



> évaluation(ex)
 [1] 7
```

Notez que l'évaluation d'un objet expression évalue chaque appel tour à tour, mais la valeur finale est celle du dernier appel. À cet égard, il se comporte presque de la même manière que l'objet de langage composé quote({2 + 2; 3 + 4}). Cependant, il existe une différence subtile : les objets d'appel ne peuvent pas être distingués des sous-expressions dans un arbre d'analyse. Cela signifie qu'ils sont automatiquement évalués de la même manière qu'une sous-expression. Les objets d'expression peuvent être reconnus lors de l'évaluation et, en un sens, conservent leur caractère cité. L'évaluateur n'évaluera pas un objet d'expression de manière récursive, uniquement lorsqu'il est transmis directement à la fonction eval comme ci-dessus. La différence peut être vue comme ceci :

```
> eval(substitute(mode(x), list(x = quote(2 + 2)))) [1] "numérique" >

eval(substitute(mode(x), list(x = expression(2 + 2)) )) [1] "expression"
```

Le deparser représente un objet d'expression par l'appel qui le crée. Ceci est similaire à la façon dont il gère les vecteurs numériques et plusieurs autres objets qui n'ont pas de représentation externe spécifique. Cependant, cela conduit à la confusion suivante :

```
> e <- citation(expression(2 + 2))
> et
expression(2 + 2) >
mode(e) [1]
 "appel" > ee
<- expression(2 + 2)
> oui
expression(2 + 2) >
mode(ee) [1]
 "expression"
```

C'est-à-dire que e et ee semblent identiques une fois imprimés, mais l'un est un appel qui génère un objet d'expression et l'autre est l'objet lui-même.

6.5 Manipulation des appels de fonction Il est possible pour une

fonction de découvrir comment elle a été appelée en regardant le résultat de sys.call comme dans l'exemple suivant d'une fonction qui renvoie simplement son propre appel :

```
> f <- function(x, y, ...) sys.call() > f(y = 1, 2, z = 3, 4)
```

f(y = 1, 2, z = 3, 4)

Cependant, cela n'est pas vraiment utile, sauf pour le débogage, car cela nécessite que la fonction garde une trace de la correspondance des arguments afin d'interpréter l'appel. Par exemple, il doit être capable de voir que le deuxième argument réel correspond au premier argument formel (x dans l'exemple ci-dessus).

Le plus souvent, on nécessite l'appel avec tous les arguments réels liés aux formels correspondants.
À cette fin, la fonction match.call est utilisée. Voici une variante de l'exemple précédent, une fonction qui renvoie son propre appel avec des arguments correspondants > f <- function(x, y, ...)

match.call() > f(y = 1, 2, z = 3, 4) f(x = 2, y = 1, z = 3, 4)

Notez que le deuxième argument correspond désormais à x et apparaît dans le champ correspondant.
position dans le résultat.

L'utilisation principale de cette technique est d'appeler une autre fonction avec les mêmes arguments, en supprimant éventuellement certains et en en ajoutant d'autres. Une application typique est visible au début de la fonction lm :

```
mf <- cl <- match.call() mf$singular.ok
<- mf$model <- mf$method <- NULL mf$x <- mf$y <- mf$qr <-
mf$contrasts <- NULL mf$drop.unused.levels <- TRUE mf[[1]] <-
as.name("model.frame") mf <- eval(mf,
sys.frame(sys.parent()))
```

Notez que l'appel résultant est évalué dans le cadre parent, dans lequel on peut être certain que les expressions impliquées ont un sens. L'appel peut être traité comme un objet de liste où le premier élément est le nom de la fonction et les éléments restants sont les expressions d'argument réelles, avec les noms d'argument formels correspondants sous forme de balises. Ainsi, la technique pour éliminer les arguments indésirables est d'attribuer NULL, comme vu aux lignes 2 et 3, et pour ajouter un argument, on utilise l'affectation de liste taguée (ici pour passer drop.unused.levels = TRUE) comme à la ligne 4. Pour changer le nom de la fonction appelée, attribuez-le au premier élément de la liste et assurez-vous que la valeur est un nom, soit en utilisant la construction as.name("model.frame") ici, soit en guillemet(model.frame).

La fonction match.call a un argument expand.dots, un commutateur qui, s'il est défini sur FALSE, permet tous les arguments ... doivent être collectés comme un seul argument avec la balise ....

```
> f <- function(x, y, ...) match.call(expand.dots = FALSE) > f(y = 1, 2, z = 3, 4) f(x = 2, y
= 1, . ..
                                    = liste(z = 3, 4))
```

L'argument ... est une liste (une liste de paires pour être précis), pas un appel à une liste comme c'est le

cas dans S : > e1 <- f(y = 1, 2, z = 3, 4)$...
> e1
$z
[1] 3

[[2]]
[1] 4

L'une des raisons d'utiliser cette forme de match.call est simplement de se débarrasser de tous les arguments ... afin de ne pas transmettre d'arguments non spécifiés à des fonctions qui ne les connaissent peut-être pas. Voici un exemple paraphrasé de plot.formula :

```
m <- match.call(expand.dots = FALSE) m$... <- NULL
m[[1]] <-
"model.frame"
```

Une application plus élaborée se trouve dans update.default où un ensemble d'arguments supplémentaires facultatifs peuvent ajouter, remplacer ou annuler ceux de l'appel d'origine :

```
extras <- match.call(expand.dots = FALSE)$... if (length(extras)
> 0) {
    existant <- !is.na(match(names(extras), noms(appel))) pour (a dans les
    noms(extras)[existing]) call[[a]] <- extras[[a]] if (any (!existing)) { call <- c(as.list(call),
    extras[!existing]) call <-
        as.call(call)

    }
}
```

Notez que l'on prend soin de modifier les arguments existants individuellement en cas d'extras[[a]] == NULL. La concaténation ne fonctionne pas sur les objets d'appel sans la coercition, comme indiqué ; c'est sans doute un bug.

Deux autres fonctions existent pour la construction d'appels de fonction, à savoir call et do.call.

L'appel de fonction permet de créer un objet appel à partir du nom de la fonction et de la liste des arguments

```
> x <- 10,5
> appeler("tour", x)
tour(10.5)
```

Comme on le voit, la valeur de x plutôt que le symbole est insérée dans l'appel, elle est donc nettement différente de round(x). Le formulaire est utilisé assez rarement, mais il est parfois utile lorsque le nom d'une fonction est disponible sous forme de variable caractère.

La fonction do.call est apparentée, mais évalue l'appel immédiatement et prend les arguments d'un objet de mode "liste" contenant tous les arguments. Une utilisation naturelle de ceci est lorsque l'on souhaite appliquer une fonction comme cbind à tous les éléments d'une liste ou d'un bloc de données.

```
is.na.data.frame <- function (x) {
    y <- do.call(cbind, lapply(x, is.na)) rownames(y) <-
    row.names(x)
    et
}
```

D'autres utilisations incluent des variations sur les constructions comme do.call("f", list(...)). Cependant, il faut être conscient que cela implique l'évaluation des arguments avant l'appel de fonction réel, ce qui peut mettre en échec les aspects d'évaluation paresseuse et de substitution d'arguments dans la fonction elle-même. Une remarque similaire s'applique à la fonction d'appel.

6.6 Manipulation des fonctions

Il est souvent utile de pouvoir manipuler les composants d'une fonction ou d'une fermeture. R fournit un ensemble de fonctions d'interface à cet effet.

corps Renvoie l'expression qui est le corps de la fonction.

formals Renvoie une liste des arguments formels de la fonction. Ceci est une liste de paires.

environnement
Renvoie l'environnement associé à la fonction.

corps<- Cela définit le corps de la fonction sur l'expression fournie.

formels<-
Définit les arguments formels de la fonction sur la liste fournie.

environnement<-

Définit l'environnement de la fonction sur l'environnement spécifié.

Il est également possible de modifier les liaisons de différentes variables dans l'environnement de la fonction, en utilisant du code dans le sens de evalq(x <- 5, Environment(f)).

Il est également possible de convertir une fonction en liste en utilisant as.list. Le résultat est la concaténation de la liste des arguments formels avec le corps de la fonction. Inversement, une telle liste peut être convertie en fonction en utilisant as.function. Cette fonctionnalité est principalement incluse pour la compatibilité S.
Notez que les informations sur l'environnement sont perdues lorsque as.list est utilisé, alors que as.function a un argument qui permet de définir l'environnement.

7 Interfaces système et en langue étrangère

7.1 Accès au système d'exploitation

L'accès au shell du système d'exploitation se fait via le système de fonctions R. Les détails différeront selon plate-forme (voir l'aide en ligne), et tout ce que l'on peut supposer en toute sécurité est que le premier argument sera une commande de chaîne qui sera passée pour exécution (pas nécessairement par un shell) et le le deuxième argument sera interne et, s'il est vrai, collectera la sortie de la commande dans un Vecteur de caractère R.

Les fonctions system.time et proc.time sont disponibles pour le timing (bien que les informations disponible peut être limité sur les plates-formes non-Unix).

Les informations de l'environnement du système d'exploitation peuvent être consultées et manipulées avec

Sys.getenv	Variables d'environnement du système d'exploitation
Sys.putenv	
Sys.getlocale	Paramètres régionaux du système
Sys.putlocale	
Sys.localeconv	
Sys.time	Heure actuelle
Sys.timezone Un	Fuseau horaire

ensemble uniforme de fonctions d'accès aux fichiers est fourni sur toutes les plateformes :

fichier.access	Vérifier l'accessibilité des fichiers
fichier.append	Concaténer des fichiers
fichier.choisir	Demander à l'utilisateur le nom du fichier
fichier.copie	Copier des fichiers
fichier.create	Créer ou tronquer un fichier
le fichier existe	Test d'existence
fichier.info	Informations diverses sur les fichiers
fichier.remove	supprimer des fichiers
fichier.renommer	renommer des fichiers
fichier.show	Afficher un fichier texte
dissocier	Supprimez des fichiers ou des répertoires.

Il existe également des fonctions permettant de manipuler les noms et les chemins de fichiers de manière indépendante de la plate-forme. chemin.

nom de base	Nom de fichier sans répertoire
nom de répertoire	Nom du répertoire
fichier.chemin	Construire le chemin du fichier
chemin.expand	Développez ~ dans le chemin Unix

7.2 Interfaces en langues étrangères

Voir la section « Interfaces système et en langue étrangère » dans Écriture d'extensions R pour plus de détails sur ajout de fonctionnalités à R via du code compilé.

Les fonctions .C et .Fortran fournissent une interface standard au code compilé qui a été lié à R, soit au moment de la construction, soit via dyn.load. Ils sont principalement destinés au C compilé et le code FORTRAN respectivement, mais la fonction .C peut être utilisée avec d'autres langages qui peut générer des interfaces C, par exemple C++.

Les fonctions .Call et .External fournissent des interfaces qui permettent du code compilé (principalement code C compilé) pour manipuler des objets R.

7.3 .Interne et .Primitive

Les interfaces .Internal et .Primitive sont utilisées pour appeler du code C compilé dans R au moment de la construction. Voir la section « .Internal vs .Primitive » dans R Internals.

8 Gestion des exceptions

Les fonctionnalités de gestion des exceptions dans R sont fournies via deux mécanismes. Des fonctions telles que l'arrêt ou l'avertissement peuvent être appelées directement ou des options telles que « avertir » peuvent être utilisées pour contrôler la gestion des problèmes.

8.1 stop Un

appel à stop arrête l'évaluation de l'expression actuelle, imprime l'argument du message et renvoie l'exécution au niveau supérieur.

8.2 warn La fonction

warn prend un seul argument qui est une chaîne de caractères. Le comportement d'un appel à warn dépend de la valeur de l'option "warn". Si « avertir » est négatif, les avertissements sont ignorés. S'il est égal à zéro, ils sont stockés et imprimés une fois la fonction de niveau supérieur terminée. S'il y en a un, ils sont imprimés au fur et à mesure qu'ils surviennent et s'il y en a 2 (ou plus), les avertissements sont transformés en erreurs.

Si "warn" vaut zéro (valeur par défaut), une variable last.warning est créée et les messages associés à chaque appel à warn sont stockés, séquentiellement, dans ce vecteur. S'il y a moins de 10 avertissements, ils sont imprimés une fois l'évaluation de la fonction terminée. S'il y en a plus de 10, un message indiquant le nombre d'avertissements survenus est imprimé. Dans les deux cas, last.warning contient le vecteur de messages et les avertissements fournissent un moyen d'y accéder et de l'imprimer.

8.3 à la sortie

Une fonction peut insérer un appel à on.exit à tout moment dans le corps d'une fonction. L'effet d'un appel à on.exit est de stocker la valeur du corps afin qu'il soit exécuté à la sortie de la fonction. Cela permet à la fonction de modifier certains paramètres du système et de garantir qu'ils sont réinitialisés aux valeurs appropriées une fois la fonction terminée. L'exécution de on.exit est garantie lorsque la fonction se termine soit directement, soit à la suite d'un avertissement.

Une erreur dans l'évaluation du code on.exit provoque un saut immédiat au niveau supérieur sans traitement ultérieur du code on.exit.

on.exit prend un seul argument qui est une expression à évaluer lorsque la fonction est quittée.

8.4 Options d'erreur

Il existe un certain nombre de variables d'options qui peuvent être utilisées pour contrôler la façon dont R gère les erreurs et les avertissements. Ils sont répertoriés dans le tableau ci-dessous.

'avertir' Contrôle l'impression des avertissements.

'warning.expression' Définit une expression qui doit être évaluée lorsqu'un avertissement se produit. L'impression normale des avertissements est supprimée si cette option est définie.

'erreur' Installe une expression qui sera évaluée lorsqu'une erreur se produit. L'impression normale des messages d'erreur et des messages d'avertissement précède l'évaluation de l'expression.

Les expressions installées par options("error") sont évaluées avant que les appels à on.exit ne soient effectués.

On peut utiliser options(error = expression(q("yes"))) pour que R quitte lorsqu'une erreur a été signalée. Dans ce cas, une erreur entraînera l'arrêt de R et l'environnement global sera sauvegardé.

9 Débogage

Le débogage du code a toujours été un peu un art. R fournit plusieurs outils qui aident les utilisateurs à trouver des problèmes dans leur code. Ces outils arrêtent l'exécution à des points particuliers du code et l'état actuel du calcul peut être inspecté.

La plupart du débogage s'effectue soit via des appels au navigateur, soit via un débogage. Ces deux fonctions reposent sur le même mécanisme interne et fournissent toutes deux à l'utilisateur une invite spéciale. N'importe quelle commande peut être saisie à l'invite. L'environnement d'évaluation de la commande est l'environnement actuellement actif. Cela vous permet d'examiner l'état actuel de toutes les variables, etc.

Il existe cinq commandes spéciales que R interprète différemment. Ils sont,

'DROITE'	Passez à l'instruction suivante si la fonction est en cours de débogage. Continuez l'exécution si le navigateur a été invoqué.
'c'	
'suite'	Continuez l'exécution.
'n'	Exécutez l'instruction suivante dans la fonction. Cela fonctionne également depuis le navigateur.
'où'	Afficher la pile d'appels
'Q'	Arrêtez l'exécution et passez immédiatement au niveau supérieur.

S'il existe une variable locale portant le même nom que l'une des commandes spéciales répertoriées ci-dessus, sa valeur est accessible en utilisant get. Un appel à get avec le nom entre guillemets récupérera la valeur dans l'environnement actuel.

Le débogueur donne accès uniquement aux expressions interprétées. Si une fonction appelle une langue étrangère (telle que C), aucun accès aux instructions dans cette langue n'est fourni. L'exécution s'arrêtera à la prochaine instruction évaluée dans R. Un débogueur symbolique tel que gdb peut être utilisé pour déboguer le code compilé.

Navigateur 9.1

Un appel au navigateur de fonctions amène R à interrompre l'exécution à ce stade et à fournir à l'utilisateur une invite spéciale. Les arguments du navigateur sont ignorés.

```
> foo <- fonction(s) { + c <- 3

+ navigateur()
+ }
> foo(4)
Appelé depuis : foo(4)
Parcourir[1]> s [1]
 4
Parcourir[1]> get("c") [1] 3

Parcourir[1]>
```

9.2 debug/undebug Le débogueur

peut être invoqué sur n'importe quelle fonction en utilisant la commande debug(fun). Par la suite, chaque fois que cette fonction est évaluée, le débogueur est invoqué. Le débogueur permet de contrôler l'évaluation des instructions dans le corps de la fonction. Avant que chaque instruction ne soit exécutée, elle est imprimée et une invite spéciale est fournie. N'importe quelle commande peut être donnée, celles du tableau ci-dessus ont une signification particulière.

Le débogage est désactivé par un appel à undebug avec la fonction comme argument.

```
> debug(mean.default) >
Mean(1:10)
débogage dans : Mean.default(1:10) debug :
{ if (na.rm)

        x <- x[!is.na(x)] trim <-
    trim[1] n <- length(c(x,
    récursive = TRUE)) if (trim > 0) { if (trim >= 0.5)
    return( median(x,
        na.rm = FALSE)) lo <-
            floor(n * trim) + 1

        salut <- n + 1 - bas
        x <- sort(x, partial = unique(c(lo, hi)))[lo:hi] n <- hi - lo + 1

    }
    somme(x)/n
}
Parcourir[1]>
débogage : if (na.rm) x <- x[!is.na(x)]
Parcourir[1]>
débogage : trim <- trim[1]
Parcourir[1]>
débogage : n <- length(c(x, récursif = TRUE))
Parcourir[1]> c
en sortant de : mean.default(1:10) [1] 5.5
```

9.3 trace/untrace Une autre

façon de surveiller le comportement de R consiste à utiliser le mécanisme de trace. trace est appelée avec un seul argument qui est le nom de la fonction que vous souhaitez tracer. Le nom n'a pas besoin d'être cité mais pour certaines fonctions vous devrez le mettre entre guillemets afin d'éviter une erreur de syntaxe.

Lorsque trace a été invoquée sur une fonction, chaque fois que cette fonction est évaluée, l'appel à celle-ci est imprimé. Ce mécanisme est supprimé en appelant untrace avec la fonction comme argument. > trace("[<-")

```
> x <- 1:10 >
x[3] <- 4 trace :
"[<-"(*tmp*, 3, valeur = 4)
```

9.4 traçabilité

Lorsqu'une erreur a provoqué un saut au niveau supérieur, une variable spéciale appelée .Traceback est placée dans l'environnement de base. .Traceback est un vecteur de caractères avec une entrée pour chaque appel de fonction qui était actif au moment où l'erreur s'est produite. Un examen du .Traceback peut être effectué

par un appel au traçage.

10 Analyseur

L'analyseur est ce qui convertit la représentation textuelle du code R en une forme interne qui peut ensuite être transmise à l'évaluateur R, ce qui provoque l'exécution des instructions spécifiées.
Le formulaire interne est lui-même un objet R et peut être enregistré et manipulé d'une autre manière dans le système R.

10.1 Le processus d'analyse

10.1.1 Modes d'analyse L'analyse dans R

se présente sous trois variantes différentes :

- La boucle lecture-évaluation-
- impression • Analyse des
- fichiers texte • Analyse des chaînes de caractères

La boucle lecture-évaluation-impression constitue l'interface de ligne de commande de base avec R. L'entrée textuelle est lue jusqu'à ce qu'une expression R complète soit disponible. Les expressions peuvent être réparties sur plusieurs lignes de saisie. L'invite principale (par défaut '> ') indique que l'analyseur est prêt pour une nouvelle expression, et une invite de continuation (par défaut '+ ') indique que l'analyseur attend le reste d'une expression incomplète. L'expression est convertie en forme interne lors de la saisie et l'expression analysée est transmise à l'évaluateur et le résultat est imprimé (sauf s'il est spécifiquement rendu invisible). Si l'analyseur se trouve dans un état incompatible avec la syntaxe du langage, une « erreur de syntaxe » est signalée et l'analyseur se réinitialise et reprend la saisie au début de la ligne de saisie suivante.

Les fichiers texte peuvent être analysés à l'aide de la fonction parse. Cela se fait notamment lors de l'exécution de la fonction source, qui permet de stocker des commandes dans un fichier externe et de les exécuter comme si elles avaient été tapées au clavier. Notez cependant que l'intégralité du fichier est analysée et la syntaxe vérifiée avant toute évaluation.

Les chaînes de caractères, ou leurs vecteurs, peuvent être analysées à l'aide de l'argument text= pour analyser. Les chaînes sont traitées exactement comme s'il s'agissait des lignes d'un fichier d'entrée.

10.1.2 Représentation interne Les expressions

analysées sont stockées dans un objet R contenant l'arbre d'analyse. Une description plus complète de ces objets peut être trouvée dans la Section 2.1.3 [Objets langage], page 3, et la Section 2.1.4 [Objets d'expression], page 4. En bref, chaque expression R élémentaire est stockée sous forme d'appel de fonction, sous la forme d'un list avec le premier élément contenant le nom de la fonction et le reste contenant les arguments, qui peuvent à leur tour être d'autres expressions R. Les éléments de la liste peuvent être nommés, correspondant à une correspondance étiquetée d'arguments formels et réels. Notez que tous les éléments de la syntaxe R sont traités de cette manière, par exemple l'affectation x <- 1 est codée comme "<-"(x, 1).

10.1.3 Déparsing Tout objet R

peut être converti en expression R à l'aide de deparse. Ceci est fréquemment utilisé en relation avec la sortie des résultats, par exemple pour l'étiquetage des parcelles. Notez que seuls les objets du mode « expression » peuvent rester inchangés en réanalysant le résultat de la déparsation. Par exemple, le vecteur numérique 1:5 sera analysé comme "c(1, 2, 3, 4, 5)", qui sera analysé comme un appel à la fonction c. Dans la mesure du possible, l'évaluation de l'expression déparsée et réanalysée donne le même résultat que l'évaluation de l'originale, mais il existe quelques exceptions gênantes, impliquant principalement des expressions qui n'ont pas été générées à partir d'une représentation textuelle en premier lieu.

10.2 Commentaires

Les commentaires dans R sont ignorés par l'analyseur. Tout texte compris entre le caractère # et la fin de la ligne est considéré comme un commentaire, sauf si le caractère # se trouve à l'intérieur d'une chaîne entre guillemets. Par exemple,

```
> x <- 1 # Ceci est un commentaire...
> et <- "#... mais ce n'est pas le cas."
```

10.3 Jetons

Les jetons sont les éléments de base d'un langage de programmation. Ils sont reconnus lors de l'analyse lexicale qui (du moins conceptuellement) a lieu avant l'analyse syntaxique effectuée par l'analyseur lui-même.

10.3.1 Constantes

Il existe cinq types de constantes : entière, logique, numérique, complexe et chaîne.

De plus, il existe quatre constantes spéciales : NULL, NA, Inf et NaN.

NULL est utilisé pour indiquer l'objet vide. NA est utilisé pour les valeurs de données absentes (« Non disponible »). Inf désigne l'infini et NaN n'est pas un nombre dans le calcul à virgule flottante IEEE (résultats des opérations respectivement 1/0 et 0/0, par exemple).

Les constantes logiques sont VRAIES ou FAUX.

Les constantes numériques suivent une syntaxe similaire à celle du langage C. Ils se composent d'une partie entière composée de zéro ou plusieurs chiffres, suivi éventuellement de « ». et une partie fractionnaire de zéro ou plusieurs chiffres éventuellement suivie d'une partie exposant constituée d'un « E » ou d'un « e », d'un signe facultatif et d'une chaîne d'un ou plusieurs chiffres. La partie fractionnaire ou décimale peut être vide, mais pas les deux à la fois.

Constantes numériques valides : 1 10 0,1 ,2 1e-7 1,2e+7

Les constantes numériques peuvent également être hexadécimales, commençant par « 0x » ou « 0x » suivi de zéro ou plusieurs chiffres, « a-f » ou « A-F ». Les constantes hexadécimales à virgule flottante sont prises en charge en utilisant la syntaxe C99, par exemple « 0x1.1p1 ».

Il existe désormais une classe distincte de constantes entières. Ils sont créés en utilisant le qualificatif L à la fin du numéro. Par exemple, 123L donne une valeur entière plutôt qu'une valeur numérique.
Le suffixe L peut être utilisé pour qualifier n'importe quel nombre non complexe dans le but de créer un entier. Il peut donc être utilisé avec des nombres donnés en notation hexadécimale ou scientifique. Cependant, si la valeur n'est pas un entier valide, un avertissement est émis et la valeur numérique est créée. Ce qui suit montre des exemples de constantes entières valides, des valeurs qui généreront un avertissement et donneront des constantes numériques et des erreurs de syntaxe.

Constantes entières valides : 1L, 0x10L, 1000000L, 1e6L Constantes
numériques valides : 1.1L, 1e-3L, 0x1.1p-2 Erreur de syntaxe : 12iL
0x1.1 Un avertissement est émis

pour les valeurs décimales qui contiennent un point décimal inutile, par exemple 1 .L.
C'est une erreur d'avoir un point décimal dans une constante hexadécimale sans l'exposant binaire.

Notez également qu'un signe précédent (+ ou -) est traité comme un opérateur unaire et non comme faisant partie du constante.

Des informations à jour sur les formats actuellement acceptés peuvent être trouvées sur
?Constantes numériques.

Les constantes complexes ont la forme d'une constante numérique décimale suivie de « i ». Notez que seuls les nombres purement imaginaires sont des constantes réelles, les autres nombres complexes sont analysés par des opérations unaires ou binaires sur des nombres numériques et imaginaires.

Constantes complexes valides : 2i 4.1i 1e-2i

Les constantes de chaîne sont délimitées par une paire de guillemets simples ('') ou doubles ("") et peuvent contenir tous les autres caractères imprimables. Les guillemets et autres caractères spéciaux dans les chaînes sont spécifiés en utilisant des séquences d'échappement :

\'	simple citation
\"	double citation
\n	nouvelle ligne (alias 'saut de ligne', LF)
\r	retour chariot (CR)
\t	caractère de tabulation
\b	retour arrière
\un	cloche
\F	saut de page
\dans	onglet vertical
\\	barre oblique inverse elle-même
\nnn	caractère avec un code octal donné – séquences d'un, deux ou trois chiffres dans la plage 0 ... 7 sont acceptés.
\xnn	caractère avec un code hexadécimal donné – séquences d'un ou deux chiffres hexadécimaux (avec entrées 0 ... 9 A ... F a ... f).
\unnnn \u{nnnn}	(où les paramètres régionaux multi-octets sont pris en charge, sinon une erreur). Caractère Unicode avec code hexadécimal donné – séquences de quatre chiffres hexadécimaux maximum. Le personnage doit être valide dans la langue actuelle.
\Unnnnnnnnn \U{nnnnnnnn}	(où les paramètres régionaux multi-octets sont pris en charge, sinon une erreur). Caractère Unicode avec code hexadécimal donné – séquences de huit chiffres hexadécimaux maximum.

Un guillemet simple peut également être intégré directement dans une chaîne délimitée par des guillemets doubles et vice versa.

Un 'nul' (\0) n'est pas autorisé dans une chaîne de caractères, donc l'utilisation de \0 dans une constante de chaîne se termine la constante (généralement avec un avertissement) : les caractères suivants jusqu'au guillemet fermant sont scannés mais ignoré.

10.3.2 Identifiants

Les identifiants sont constitués d'une séquence de lettres, de chiffres, du point ('.') et du trait de soulignement. Elles doivent ne commence pas par un chiffre ou un trait de soulignement, ni par un point suivi d'un chiffre.

La définition d'une lettre dépend de la locale actuelle : le jeu précis de caractères autorisés est donné par l'expression C (isalnum(c) || c == '.' || c == '_') et inclura les accents lettres dans de nombreux endroits d'Europe occidentale.

Notez que les identifiants commençant par un point ne sont pas répertoriés par défaut par la fonction ls et que ... et ..1, ..2, etc. sont spéciaux.

Notez également que les objets peuvent avoir des noms qui ne sont pas des identifiants. Ceux-ci sont généralement accessibles via get et assign, bien qu'ils puissent également être représentés par des chaînes de texte dans certaines circonstances limitées lorsqu'il n'y a pas d'ambiguïté (par exemple "x" <- 1). Comme get et assign ne sont pas limités à les noms qui sont des identifiants, ils ne reconnaissent pas les opérateurs d'indice ni les fonctions de remplacement. Les paires suivantes ne sont pas équivalentes

x$a<-1 assign("x$a",1)
x[[1]] obtenir("x[[1]]")
noms(x)<-nm assign("noms(x)",nm)

10.3.3 Mots réservés

Les identifiants suivants ont une signification particulière et ne peuvent pas être utilisés pour les noms d'objets

sinon, répétez la fonction pendant la prochaine pause
VRAI FAUX NULL Inf NaN

NA NA_integer_ NA_real_ NA_complex_ NA_character_ ... ..1 ..2 etc.

10.3.4 Opérateurs spéciaux R autorise

les opérateurs infixes définis par l'utilisateur. Ceux-ci se présentent sous la forme d'une chaîne de caractères délimitée par le caractère '%'. La chaîne peut contenir n'importe quel caractère imprimable sauf '%'. Les séquences d'échappement pour les chaînes ne s'appliquent pas ici.

Notez que les opérateurs suivants sont prédéfinis

%% %*% %/% %in% %o% %x%

10.3.5 Séparateurs Bien qu'il

ne s'agisse pas strictement de jetons, les étendues de caractères d'espacement (espaces, tabulations et sauts de page, sur les paramètres régionaux Windows et UTF-8, d'autres caractères d'espacement Unicode1) servent à délimiter les jetons en cas d'ambiguïté (comparez x<-5 et x < -5).

Les nouvelles lignes ont une fonction qui est une combinaison de séparateur de jetons et de terminateur d'expression. Si une expression peut se terminer à la fin de la ligne, l'analyseur supposera que c'est le cas, sinon la nouvelle ligne est traitée comme un espace. Des points-virgules (';') peuvent être utilisés pour séparer des expressions élémentaires sur une même ligne.

Des règles spéciales s'appliquent au mot-clé else : à l'intérieur d'une expression composée, une nouvelle ligne avant else est supprimée, alors qu'au niveau le plus externe, la nouvelle ligne termine la construction if et une else suivante provoque une erreur de syntaxe. Ce comportement quelque peu anormal se produit car R doit être utilisable en mode interactif et il doit ensuite décider si l'expression d'entrée est complète, incomplète ou invalide dès que l'utilisateur appuie sur RET.

La virgule (',') est utilisée pour séparer les arguments de fonction et plusieurs indices.

10.3.6 Jetons d'opérateur R utilise les

jetons d'opérateur suivants

+ - * / %% ^ >	arithmétique
>= < <= == !=	relationnel
! & | ~	modèle logique formules
-> <-	affectation
$	liste indexation
:	séquence

(Plusieurs opérateurs ont une signification différente dans les formules de modèle)

10.3.7 Regroupement Les

parenthèses ordinaires — '(' et ')' — sont utilisées pour un regroupement explicite au sein des expressions et pour délimiter les listes d'arguments pour les définitions de fonctions et les appels de fonctions.

Les accolades — '{' et '}' — délimitent des blocs d'expressions dans les définitions de fonctions, les expressions conditionnelles les sions et les constructions itératives.

---

[1] tels que U+A0, espace insécable, et U+3000, espace idéographique.

10.3.8 Jetons d'indexation L'indexation

des tableaux et des vecteurs est effectuée à l'aide des crochets simples et doubles, '[]' et '[[]]'.
De plus, l'indexation des listes balisées peut être effectuée à l'aide de l'opérateur « $ ».

10.4Expressions

Un programme R consiste en une séquence d'expressions R. Une expression peut être une expression simple composée uniquement d'une constante ou d'un identifiant, ou elle peut être une expression composée construite à partir d'autres parties (qui peuvent elles-mêmes être des expressions).

Les sections suivantes détaillent les différentes constructions syntaxiques disponibles.

10.4.1 Appels de fonction

Un appel de fonction prend la forme d'une référence de fonction suivie d'une liste d'arguments séparés par des virgules dans un ensemble de parenthèses.

référence_fonction (arg1, arg2, ..., argn)

La référence de la fonction peut être soit

- un identifiant (le nom de la fonction) • une chaîne de texte (idem, mais pratique si la fonction a un nom qui n'est pas un identifiant valide) • une expression (qui doit être évaluée comme un objet fonction)

Chaque argument peut être balisé (tag=expr), ou simplement être une simple expression. Il peut également être vide ou il peut s'agir d'un des jetons spéciaux ..., ..2, etc.

Une balise peut être un identifiant ou une chaîne de texte.

Exemples:

```
f(x)
g(tag = valeur, , "nom 5)
impair"("balise étrange" = 5, y) (fonction(x) x^2)(5)
```

10.4.2 Opérateurs infixe et préfixe L'ordre de priorité (le

plus élevé en premier) des opérateurs est

```
::
$ @
^
- +                                (unaire)
:
%xyz% |> * /
+ -                                (binaire)
> >= < <= == !=
!
& &&
| ||
~                                  (unaire et binaire)
-> ->>
<- <<-
=                                  (comme mission)
```

Notez que : précède le binaire +/-, mais pas ^. Par conséquent, 1:3-1 est 012, mais 1:2^3 est 1:8.

L'opérateur d'exponentiation '^' et l'affectation de gauche plus les opérateurs moins '<- - = <<-' pas 43 alors que groupe de droite à gauche, tous les autres opérateurs groupent de gauche à droite. Autrement dit, 2 ^ 2 ^ 3 , , vaut 28 1 - 1 - 1 vaut −1, pas 1.

Notez que les opérateurs %% et %/% pour le reste entier et la division ont une priorité plus élevée que la multiplication et la division.

Bien qu'il ne s'agisse pas strictement d'un opérateur, il convient également de mentionner que le signe '=' est utilisé pour baliser les arguments dans les appels de fonction et pour attribuer des valeurs par défaut dans les définitions de fonctions.

Le signe '$' est en quelque sorte un opérateur, mais n'autorise pas les côtés droits arbitraires et est discuté dans la Section 10.4.3 [Constructions d'index], page 50. Il a une priorité plus élevée que n'importe lequel des autres opérateurs.

La forme analysée d'une opération unaire ou binaire est complètement équivalente à un appel de fonction avec l'opérateur comme nom de fonction et les opérandes comme arguments de fonction.

Les parenthèses sont enregistrées comme équivalentes à un opérateur unaire, de nom "(", même dans les cas où les parenthèses pourraient être déduites de la priorité des opérateurs (par exemple, a * (b + c)).

Notez que les symboles d'affectation sont des opérateurs au même titre que les symboles arithmétiques, relationnels et logiques. Toute expression est également autorisée du côté cible d'une affectation, en ce qui concerne l'analyseur (2 + 2 <- 5 est une expression valide en ce qui concerne l'analyseur. L'évaluateur s'y opposera cependant). Des commentaires similaires s'appliquent à l'opérateur de formule modèle.

10.4.3 Constructions d'index

R a trois constructions d'indexation, dont deux sont syntaxiquement similaires bien qu'avec une sémantique quelque peu différente :

objet [arg1 , ... , argn] objet [[arg1 , ... , argn

L'objet peut formellement être n'importe quelle expression valide, mais il est entendu comme désignant ou évalué comme un objet sous-définissable. Les arguments sont généralement évalués sous forme d'indices numériques ou de caractères, mais d'autres types d'arguments sont possibles (notamment drop = FALSE).

En interne, ces constructions d'index sont stockées sous forme d'appels de fonction avec le nom de fonction "[" respectivement "[[".

La troisième construction d'indice est

objet balise $

Ici, l'objet est comme ci-dessus, tandis que la balise est un identifiant ou une chaîne de texte. En interne, il est stocké comme appel de fonction avec le nom "$"

10.4.4 Expressions composées Une expression

composée est de la forme

{ expr1 ; expr2 ; ..... ; expression }

Les points-virgules peuvent être remplacés par des nouvelles lignes. En interne, ceci est stocké sous forme d'appel de fonction avec "{" comme nom de fonction et les expressions comme arguments.

10.4.5 Éléments de contrôle de débit

R contient les structures de contrôle suivantes en tant que constructions syntaxiques spéciales

if (cond) expr if (cond)
expr1 else expr2 while (cond) expr répéter
expr pour (var dans la liste)
expr

Les expressions de ces constructions seront généralement des expressions composées.

Dans les constructions de boucle (while, répéter, for), on peut utiliser break (pour terminer la boucle) et suivant (pour passer à l'itération suivante).

En interne, les constructions sont stockées sous forme d'appels de fonction : "if"(cond, expr) "if"(cond, expr1, expr2) "while"(cond, expr) "repeat"(expr) "for"(var, list, expr) "break"() "suivant"()

10.4.6 Définitions des fonctions

Une définition de fonction est de la forme

fonction (arglist) corps

Le corps de la fonction est une expression, souvent une expression composée. La liste argliste est une liste d'éléments séparés par des virgules dont chacun peut être un identifiant, ou de la forme 'identifiant = défaut', ou le jeton spécial .... La valeur par défaut peut être n'importe quelle expression valide.

Notez que les arguments de fonction, contrairement aux balises de liste, etc., ne peuvent pas avoir de « noms étranges » donnés sous forme de chaînes de texte.

En interne, une définition de fonction est stockée sous forme d'appel de fonction avec le nom de fonction function et deux arguments, la liste d'arguments et le corps. La liste d'arguments est stockée sous forme de liste de paires balisée où les balises sont les noms d'arguments et les valeurs sont les expressions par défaut.

10.5 Directives

L'analyseur ne prend actuellement en charge qu'une seule directive, #line. Ceci est similaire à la directive du préprocesseur C du même nom. La syntaxe est

#line nn ["filename"] où nn est un numéro de ligne entier et le nom de fichier facultatif (entre guillemets doubles obligatoires) nomme le fichier source.

Contrairement à la directive C, #line doit apparaître comme les cinq premiers caractères d'une ligne. Comme en C, les entrées nn et "filename" peuvent en être séparées par des espaces. Et contrairement au C, tout texte suivant sur la ligne sera traité comme un commentaire et ignoré.

Cette directive indique à l'analyseur que la ligne suivante doit être considérée comme étant la ligne nn du fichier nom de fichier. (Si le nom du fichier n'est pas donné, il est supposé être le même que pour la directive précédente.) Ceci n'est généralement pas utilisé par les utilisateurs, mais peut être utilisé par les préprocesseurs afin que les messages de diagnostic fassent référence au fichier d'origine.

Fonction et indice variable

#

$

.

[

UN

B

D

ET

F

g

je

M

N

Ô

P.

Q

R.

S

T

DANS

DANS

Index des concepts

Annexe A Références

Richard A. Becker, John M. Chambers et Allan R. Wilks (1988), Le nouveau langage S. Chapman & Hall, New York. Ce livre est souvent appelé le « Livre Bleu ».