Le scoping correspond à l’ensemble de règles que R utilise pour associer une valeur à un symbole. Acquérir une bonne connaissance de ces règles permet :

* D’exploiter les possibilités de la programmation fonctionnelle
* De ne pas se perdre dans l’utilisation de la NSE

H. Wickham distingue deux types de scoping : lexical et dynamic scoping. Le lexical scoping s’appuie sur la façon dont des fonctions ont pu être emboitées à leur création et non à lors de leur appel pour rechercher une valeur. R implémente le lexical scoping à travers 4 principes :

- Name masking : « quand R ne trouve pas une valeur dans l’environnement local, il remonte d’un cran ». Le name masking doit désigner le fait qu’un même symbôle peut être associé à deux valeurs différentes dans deux environnements différents mais il n’y a pas de collision car la valeur choisie sera la première rencontrée sur le search path. La seconde est masquée. Ou alors il s’agit de dire que ce n’est pas parce qu’un symbole est lié à une valeur quelque part qu’il est visible de partout. https://en.wikipedia.org/wiki/Name\_resolution\_(programming\_languages)#Name\_masking

- Names vs Functions : la recherche de fonctions qui sont des objets associés à une valeur suit le même principe. Dans le cas où il semble évident qu’on veut avoir recours à une fonction mais qu’une valeur est aussi associé au même nom, R va privilégier la fonction. Ex c() et si c <- 1.

- A fresh start : Dans le cas des fonctions, l’environnement d’éxécution est détruit une fois toutes les instructions du corps de la fonction exéctutées. Toutes les variables qui ont pu y être définies sont perdues et devront être recréées à la prochaine exécution.

- Dynamic lookup : Le lexical scoping dit où chercher les valeurs et non quand. Comme une fonction va remonter les parents de son enclosing environment si jamais certaines de ses variables sont unbound (?), son comportement va dépendre de l’état des variables définies à l’extérieur de son environnement d’exécution.

Dynamic scoping (source de confusion par rapport au terme dynamic lookup du lexical scoping): désigne l’idée qu’une fonction ne va pas remonter les parents de son enclosing environment mais va aller regarder dans son calling environment. Peu de langages l’implémentent comme la règle générale car pour comprendre comment une fonction fonctionne devient compliqué car il faut savoir à la fois comment elle a été définie mais aussi dans quel contexte elle a été appelée.

Scoping = portée | portée lexical | portée dynamique

<https://fr.wikipedia.org/wiki/Port%C3%A9e_(informatique)>

<https://en.wikipedia.org/wiki/Scope_(computer_science)#Lexical_scope_vs._dynamic_scope>

Par rapport au wiki français : remarque sur les variables globales, il semble que le concept n’existe pas vraiment en R (?)

<https://cran.r-project.org/doc/manuals/R-lang.html#Scope-of-variables>

Le fait de rendre accessible des variables définies plus haut dans le call stack s’appelle du dynamic scoping. On va chercher des variables dans des environnements d’exécution (éphémères) non encore détruits. La liaison (binding) d’une variable est déterminée par la plus récente (en termes de temps) définition de la variable. Cela va contre le lexical (ou static) scoping où la liaison peut être simplement déterminée en regardant la position de la variable dans le code source. Ici la liaison dépend de où (quand) on se trouve dans l’exécution : la variable peut être accédée de la position actuelle dans le call stack, puis modifiée pour être réaccédée plus loin. Lexical scoping : les liaisons peuvent se déterminer à la compilation (compile time), pour le dynamic scoping cela se fait à l’exécution (run time).

En programmation, on appelle scope (portée) d’une liaison (name binding), c’est à dire de l’association d’un nom[[1]](#footnote-0) à une entité (code, données), la partie du programme où cette liaison est valide. Dit autrement, il s’agit de l’ensemble des partie du programme où la liaison de tel nom avec telle entité est valide, où l’on peut se référer à cette entité avec ce nom. Ailleurs, ce ce nom peut être associé à une entité différente voire à rien du tout (unbound). Quand on se place du point de vue de l’entité, le scope désigne l’ensemble des parties du programme où celle-ci est dite visible.

En pratique et par abus de langage, on utilise le terme scope pour désigner l’ensemble des entités visibles / des noms valides dans une certaine portion du programme.[[2]](#footnote-1) Le terme plus juste pour ces ensembles serait celui d’environnement ou de contexte. L’expression «portion du programme» peut se référer à une portion du code source (en pratique, une zone de texte), on parle alors de lexical scoping[[3]](#footnote-2). Dans certains langages en revanche, «portion du programme» se réfère à une période particulière de l’exécution, on parle alors de dynamic scoping.

La résolution de noms (name résolution) basée sur le lexical scope est en pratique la plus évidente à implémenter et à utiliser : il suffit de lire le code source à rebours afin de savoir quelle entité est associée à quel nom et l’implémentation ne demande que de tenir une liste de noms et d’environnements.

Dans le lexical scoping, la résolution de noms ne dépend que de la localisation dans le code source et de l’environnement lexical (lexical context) associé à cet endroit alors qu’en dynamic scoping elle dépend de l’état du programme en train de s’exécuter au moment où le nom est rencontré. Ce dernier état est déterminé par l’environnement d’exécution (execution / calling context). Dans le premier cas si on ne parvient pas à résoudre le nom dans un «bloc», on poursuit la recherche en regardant dans le «bloc» le contenant, etc. Dans le second cas, si l’exploration de l’environnement d’exécution de la calling function ne donne rien, on remonte à l’environnement de la fonction qui a appelé la calling function, etc le long du call stack. Dans le deux cas on cherche d’abord une définition locale de la variable.

«Lexical resolution can be determined at compile time while dynamic resolution can in general only be determined at run time.»

Remarque : En POO, il est fréquent de devoir déterminer quelle implémentation d’une opération polymorphique (méthode ou fonction (ex: print)) doit être appelée à l’exécution. Cette opération appelée dynamic dispatch.

Le scope est un aspect important de la name resolution[[4]](#footnote-3). Les règles permettant de l’établir sont appelées «scoping rules».

Types de scope :

La scoping rule la plus simple est le scope global : les entités de scope global sont visibles dans l’ensemble du programme, la relation identifiant-entité y est valide partout. Le scoping modulaire (modular scoping) le plus simple consiste en deux niveaux de scope : scoping global et un scope local à l’intérieur de fonctions. Certains langages (comme C) impémentent le block scope qui permet de restreindre le scope à des sous-ensembles de fonctions. Certains langages fonctionnels implémentent même l’expression scope où le scope est réduit à une seule expression.

Savoir où commence et où finit le scope peut être subtil. En C, le scope commence à la déclaration, ainsi différents noms déclarés au sein d’un même bloc peuvent avoir différents scopes. Dans d’autres langages comme Python ou R, le scope commence au début du bloc considéré (ex: au début du corps d’une fonction) et tout les noms d’un même bloc possèdent le même scope peut importe où ils s’y trouvent. Attention, ne pas confondre avec le block scope (que Python ne supporte d’ailleurs pas) qui se réfère a fait qu’une variable locale puisse sortir du scope avant la fin d’une fonction par exemple.

Exemple d’expression scope : celui de la «variable muette» des comprehension lists ou des générateurs en Python.

Block scope : La plupart mais pas tous les block-structured langages permettent que le scope puisse se réduire à un bloc de code (block scope). Tout se passe comme si le bloc générait son environnement d’exécution, quand l’exécution du bloc est terminée, les noms qui y avaient été définis sont perdus. De tels blocs sont souvent contenus dans le corps de fonction restreignant ainsi le scope à une partie de la fonction. Attention tous les langages autorisant le recours au block scope (Perl, C et descendants) ne l’implémentent pas de la même façon. Les blocs ne se sont pas utilisés que dans le corps de fonctions, ils le sont aussi pour des structures de contrôle (if, for, while, etc.) : c’est par exemple comme si chaque bloc d’un if générait son propre environnement d’exécution. Attention par contre, si le bloc est utilisé pour assigner à une valeur à une variable qu’on souhaite réutiliser plus tard, cette dernière doit être déclarée en dehors du bloc. Dans certains langages, ce concept peut être mis à profit pour construire des closures.

Function scope : La plupart des langages les plus répandus permettent de créer des variables locales à l’intérieure de fonction, ces variables sortant du contexte quand la fonction retourne son résultat. Dans la plupart des cas, la durée de vie de la variables est même égale au temps d’exécution de la fonction[[5]](#footnote-4) : la variable est créée à l’appel de la fonction (ou au moment de sa déclaration dans le corps de celle-ci) puis détruite à la fin de son exécution.

R est un lexically scoped language. Les variables libres d’une fonction sont déterminées par l’environnement dans lequel la fonction a été créée. Des fonctionnalités comme parent.frame() laissent en revanche au programmeur la possibilité de simuler l’effet du dynamic scoping.

En lexical scoping, une variable locale n’est en contexte que dans le contexte lexical de la fonction où elle est définie. Si cette dernière appelle une autre fonction, ses variables locales sortent du contexte jusqu’à ce que les fonctions appelées se soient exécutées après quoi elles reviennent dans le contexte. En dehors du texte de la fonction, la variable n’existe pas. En particulier, les fonctions appelées n’ont pas accès aux variables locales de la fonction qui les a invoquées. Chaque fonction a son scope sans qu’ils puissent se recouvrir et donc interférer (pas de risque de name collision). Dans le cas du dynamic scoping, les variables locales restent en contexte lors de l’appel d’autres fonctions et le reste jusqu’à ce que la fonction dans laquelle elles sont définies ait fini de s’exécuter. Une fois la fonction exécutée, ses variables locales n’existent plus. Ces règles font qu’il est possible que deux identifiants identiques puissent se retrouver en contexte en même temps (par exemple si la fonction qui appelle et celle appelée utilisent toute les deux une variable locale n), la variable locale de la fonction qui appelle resterait alors en contexte mais serait masquée (shadowed) par celle du la fonction appelée. En lexical scoping, dans le cas particulier où une fonction serait définie puis appelée à l’intérieur d’une enclosing function, les variables locales de la dernière sont accessibles à la première comme variables non-locales. Les variables locales de l’enclosing function et la fonction qui y est définie font en effet partie du même contexte lexical. Le function scope s’applique aux fonctions anonymes.

Dans les cas où les fonctions seraient aussi des first-class objects qui peuvent ainsi être créées localement dans le corps d’une fonction puis retournées, le function scope devient plus compliqué. Toutes les variables non-locales dans la nested function peuvent se résoudre dans l’enclosing context. Il n’est alors pas simplement retourné une fonction mais une closure qui regroupe la fonction et son enclosing context, la fonction pouvant plus tard être appelée dans un autre contexte. Cette possibilité est plus exigeante pour le compilateur/interpréteur et complique l’analyse du programme. Une implémentation correcte du lexical scope dans des langages où les fonctions sont des first-class objects impose que chaque fonction porte avec elle le registre des variables dont elle dépend. L’association d’une fonction et de cet environnement étant appelé une closure.

Il existe d’autre types de scope propres à certains langages : module scope (Python et d’autres), file scope (C, C++).

Global scope : on évite en général de trop y déclarer de variables de façon à éviter les collisions et le masquage non-intentionnel. Toutefois des outils comme les namespaces permettent sont utilisés de façon à éviter les collisions.

En lexical scoping, un nom se réfère à son environnement lexical. C’est une propriété du texte du programme et est indépendant de la pile d’exécution (call stack). Comme la mise en correspondance ne requiert qu’une analyse du texte statique du programme, ce type de scoping est également appelé statique scoping. Le scoping statique a l’avantage de permettre de raisonner avec des références à des objets comme de simples substitutions de noms. Il est également plus aisé de rendre le code modulaire et de raisonner dessus, chaque contexte local pouvant être considéré isolément. A l’opposé, le dynamic scoping impose au programmeur d’anticiper tout les contextes possibles dans lequel un module de code peut être invoqué.

En dynamic scope, un identifiant global fait référence à l’identifiant associé à l’environnement le plus récent. Le dynamic scope est peu fréquent dans les langages récents. A chaque identifiant est en fait associée une pile de liaisons globale (global stack of bindings) (d’autres implémentations du dynamic scope existent). Introduire une nouvelle variable locale x place cette liaison au sommet de la pile globale de x, la liaison étant sortie de la pile lorsque lorsque le programme en quitte le scope. Evaluer x produit toujours la valeur de la liaison située au sommet de la pile. L’évaluation ne peut pas être faire à la compilation, la pile n’existant qu’à l’exécution.

Généralement, certains blocs sont conçus de façon à créer des liaisons ont une durée de vie égale au temps d’exécution du bloc ajoutant ainsi des éléments de lexical scoping dans l’élaboration du dynamic scope. Comme un morceau de code peut être appelé depuis différents endroits dans différents situation, le dynamic scoping rend difficile d’anticiper quelles liaisons vont s’appliquer lors de l’utilisation d’une variable. Ce mécanisme peut en revanche apporter de la flexibilité mais demande une documentation précise de chaque variable et une certaine prudence quant à la possibilité d’interférences entre différentes parties du programme. L’utilisation du dynamic scoping fait aussi perdre les bénéfices de la transparence référentielle, d’où la dangerosité de son utilisation. Les langages shell (bash, PowerShell) utilisent le dynamic scoping.

Une des raison d’être du scope est d’aider à éviter les name collisions : on autorise des identifiants identiques à se référer à des entités distinctes avec la restriction que les scopes de ces identifiants doivent être distincts. Cette restriction est parfois peu pratique quand différents éléments doivent rester accessibles tout au long du programme. Différentes techniques existent alors pour éviter les name collisions associée au fait de conserver de nombreux éléments en global scope. L’implémentation de ces techniques peut être très diverse mais toute visent à pouvoir organiser les identifiants globaux (les namespaces en C++ ou C# par exemple).

Remarque : il semble qu’une fonction définie à l’intérieur d’une autre fonction (où elle peut éventuellement être appelée) est appelée une nested function. Tout les langages ne les autorisent pas (C par exemple).

Rq : <https://en.wikipedia.org/wiki/Executable>

Rq : name masking et variable shadowing (ou juste shadowing) ont l’air d’être des concepts proches

<https://en.wikipedia.org/wiki/Variable_shadowing>

Définition : Le code source (source code) d’un programme correspond à une collection d’instructions possiblement accompagnées de commentaires. Le code source est écrit dans un langage de programmation sous une forme compréhensible par l’homme. On l’oppose au code binaire, machine décrivant les instructions à exécuter par un microprocesseur sous forme numérique. Suivant les cas, le code source peut être soit transformé en code machine par un compilateur, le code compilé pouvant être stocké pour une utilisation ultérieure, soit directement interprété et exécuté[[6]](#footnote-5).

Identifiant (Identifier) : un identifiant est une unité lexicale (token) aussi appelée symbole, permettant de nommer des entités d’un langage. Ces dernières peuvent être des variables, types[[7]](#footnote-6), labels[[8]](#footnote-7), fonctions (appelées aussi procédures ou routines) ou encore des packages. Toutes les séquences de caractères ne sont pas autorisées pour servir d’identifiant (pas d’espace, ne pas commencer par un chiffre, caractères interdits) certains mots pouvant aussi être réservés (if, for, etc.). Les règles sont propres à chaque langage. Dans certains langages, les identifiants peuvent être des first-class entities et manipulés et évalués comme n’importe quel objet (à la base de la NSE en R).

Une variable : Une variable est l’association d’un identifiant, un symbole avec un emplacement de stockage identifié par une adresse mémoire. Le contenu de l’emplacement mémoire contient une certaine quantité d’information correspondant à la valeur de la variable. La valeur d’une variable peut le plus souvent être accédée ou modifiée à n’importe quel moment (mais il existe certains paradigmes de programmation où la valeur de la variable est au contraire figée pour l’intégralité du temps d’exécution). Suivant le type system (typage ?) du langage, une variable ne peut stocker qu’un certain type[[9]](#footnote-8) ou au contraire peut toujours être associée à la valeur courante et donc stocker indifféremment n’importe lequel des types supportés par le langage.

Le scope n’impose pas sous certaines restrictions que l’identifiant d’une variable soit unique.

On peut distinguer plusieurs caractéristiques pour une variables :

- Nom : son identifiant.

- Type : Suivant le langage, il peut avoir dû être explicitement donné par le programmeur ou inférer par l’ordinateur de façon à ce qu’il sache comment stocker la valeur (le type donne la longueur de la séquence (ex: 32bits) et comment elle doit être interprétée (ex: si on stocke un caractère)).

- Adresse : adresse mémoire où la valeur est stockée.

- Portée : scope de son identifiant.

- Visibilité : Règles qui fixe qui peut utiliser la variable (quand elle est évidemment en contexte). Désigne des mots-clé comme public, private, protected, etc. ou le masquage d’une variable par une autre. Variable masquée est dite hors-de-portée.

- Durée de vie : Temps d’exécution pendant lequel la variable existe.

Remarque : dans certains langages, la notion de variable est complétée par la notion de pointeur qui est en fait une variable dont la valeur est une adresse mémoire. Cette adresse peut suivant les cas être accessible au programmeur (C++) ou lui être inaccessible (Java, PHP).

On distingue généralement cinq opérations sur les variables, chacune pouvant revêtir des formes syntaxiques différentes :

- la déclaration permet de déclarer un nom de variable, éventuellement de lui associer un type,

- la définition permet d'associer une zone mémoire qui va être utilisée pour stocker la variable, comme lorsqu'on lui donne une valeur initiale,

- l'affectation consiste à attribuer une valeur à une variable,

- la lecture consiste à utiliser la valeur liée à la variable,

- la suppression réalisée soit automatiquement soit par une instruction du langage.

Certains langages (de paradigme réflexif ?) permettent de considérer le nom d’une variable comme un objet comme un autre pouvant en particulier devenir la valeur d’une autre variable. R en particulier donne cette possibilité qui est une des bases sur lesquelles repose la métaprogrammation dans ce langage.

1. Aussi désigné sous le terme plus général d’identifiant (identifier) [↑](#footnote-ref-0)
2. On parle parfois aussi de bloc de code, dans les langages structurés par bloc (block-structured programming langage), le bloc a une signification particulière. Du point de vue du scoping, toutes les instructions situées au sein du bloc sont traitées comme s’il s’agissait d’une seule. [↑](#footnote-ref-1)
3. Appelé parfois static scope [↑](#footnote-ref-2)
4. https://en.wikipedia.org/wiki/Name\_resolution\_(programming\_languages) [↑](#footnote-ref-3)
5. Des langages comme C donne en effet la possibilité de créer des variables locales statiques dont la durée de vie est celle du programme mais qui ne sont dans le contexte qu’à l’intérieur de la fonction. Comme un variable statique globale, la variable est créée à l’initialisation du programme puis détruite lorsque son exécution se termine. En revanche son caractère local fait qu’elle n’est visible, en contexte qu’à l’intérieur de la fonction où elle est définie. [↑](#footnote-ref-4)
6. Fonde l’opposition en langage compilé et langage interprété [↑](#footnote-ref-5)
7. Un type ou data type est une classification des données qui indique au compilateur/interpréteur comment le programmeur compte utiliser les données. Le type défini aussi les opération qui pourront être effectuées sur les données et la façon dont les données de ce type seront stockées. Les types les plus courants sont : entiers (integers), booléens, les caractères, nombres à virgule flottante, etc. [↑](#footnote-ref-6)
8. Un label est une séquence de caractères qui sert de lien vers un autre endroit, un autre bloc du code source (pour les langages qui implémentent cette fonctionnalité). [↑](#footnote-ref-7)
9. C’est par exemple le cas des langages à typage statique comme Java où une variable est aussi associée à un type. A l’opposé, le typage dynamique ne demande pas au programmeur de déclarer explicitement le type de chacune de ses variables. Le typage est alors réalisé à la volée par l’ordinateur lors de l’exécution. [↑](#footnote-ref-8)