Compilation
Poly:
Structure du compilateur

24 janvier 2019

1 Introduction

1.1 Conventions

Le symbole † dans le nom d'une section ou sous-section veut dire que celle-ci n'est a priori pas au programme. [1]

Le symbole ⋆ dans le nom d'une section ou sous-section veut dire que celle-ci ne concerne pas les langages interprétés et est spécifique à la compilation.(cf. section 1.2)

Comme vous le remarquerez peut-être, il y a une forte correspondance entre les deux. En fait, une grande partie de ce cours concernera aussi bien les compilateurs que les interpréteurs.

On utilisera aussi (†) et (⋆) comme des versions modérées ; entendre "à la limite du programme" et "d'importance mineur pour les interpréteurs".

Remarque : ce polycopié n'a pas vocation à remplacer le cours magistral. Il ne contient pas l'intégralité de ce qui sera vu en cours, mais juste une vue d'ensemble permettant d'articuler les différentes partie du cours. En particulier, il n'y a aucune technique abordée ici, contrairement aux deux autres polys fournis qui traitent formellement les deux parties majeurs du cours : le frontend et les machines abstraites.

1.2 Implémenter un langage

Les langages informatiques sont de trop haut niveau pour être compris par le circuit du compilateur. En particulier, un processeur n'aura pas de vue global sur le code et ne peut qu'exécuter les opérations les unes après les autres.

Il faut donc implémenter nos programmes dans un autre langage plus bas niveau.

1. Vu que le cours est en construction, il est possible que certains † disparaissent au cours du semestre, cela veut dire que j'ai finalement décider de les incorporer explicitement au cours.

Évidement, cette implémentassions ne se fait pas à la main (sinon il n'y aurait aucune raison d'écrire le programme haut niveau...). Pour ça on fait appel à un programme intermédiaire $I : \mathtt{P}(S) \to \mathtt{P}(C)$ qui va transformer les programme/procédures écrits dans le langage source S en un programme/procédure équivalent écrit dans le langage cible "C".

Il deux "types" de programmes de ce genre : les compilateurs qui s'utilise bien avant l'exécution et transforment tous programme cible en in programme source ; et les interpréteurs qui exécutant le résultat en même temps et qui disposent donc des entré ; cela permet de ne transformer que les procédures (programme + entrés).

Remarque†**.** Si un langage dispose d'un compilateur, ça ne veut pas dire qu'il n'existe pas d'interpréteur, et vice-versa. Lorsque l'on dit qu'un langage est "interprète" ou "compilé", cela veut simplement dire qu'il a été conçu pour être interprète/compilé et que l'implémentation de référence (cf. Section 4.7) est un interpréteur/compilateur. Par exemple, Python (qui est interprété) dispose maintenant d'un compilateur, et Haskell (qui est compilé) dispose aussi d'un interpréteur (ghci).

1.3 Compilateurs*

1.3.1 Principe*

Un compilateur est un programme qui prend en entré un programme p dans un langage source spécifique S et rend un programme "équivalent" q écrit dans un langage cible spécifique C. Le programme sortant, (généralement) un bitecode, peut alors être conservé dans l'état et exécuté autant de fois que l'on veut.

Le compilateur en profite aussi pour faire de la vérification, de la préexecution et de l'optimisation du code compilé. Ces vérifications, ces préexécutions, ainsi que les erreurs renvoyées au niveau compilateurs sont dit "statiques", par opposition à leurs version "dynamiques" qui arrivent pendant l'exécution.

Ce cours à pour but de décrire en détails ce que fait le compilateur.

1.3.2 Bootstrap*†

Un langage "n'existe" pas vraiment tant qu'il n'a pas été implémenté. Mais pour l'implémenter, il faut écrire le compilateur/interpréteur dans un langage de programmation... Il faut donc choisir un autre langage. Comment ont-ils fait au départ ?

Ils ont évidement écrit le premier compilateur en assembleur. Malheureusement, il n'est pas humainement possible de faire de gros programme en assembleur, on ne peut implémenter qu'une petite partie. C'est pour ça que beaucoup de compilateur bootstrapent. Cela consiste à avoir un petit compilateur écrit en assembleur ou en C^{--} (fragment minimal du C) qui ne compile qu'un noyau minimal du langage, puis un nouveau compilateur est écrit dans ce noyau minimal pour compiler une version plus large du langage, etc...

1.4 Interpréteurs

Un interpréteur est un programme qui prend en entré un programme p dans un environnement connu E et qui va lire une par une les instructions de p pour les transformer en instructions du langage cible C, tout en utilisant l'environnement E (qui évolue) pour savoir comment faire.

Une boucle explicite va être transformée en boucle, mais si une fonction est appelé deux fois, elle sera interprété deux fois. Par contre, si son comportement dépend de son environnement (le type sur lequel elle est utilisée par exemple), elle sera immédiatement spécifiée.

Exercice†. Le bootstrap ne fonctionne pas pour les interpréteurs. Pourquoi ?

1.5 Avantages et inconvénients†

1.5.1 Complexité†

Compilateurs†. Vu que la compilation se fait avant l'exécution et une seule fois, on a tendance à penser que l'on utilise des algorithmes très gourmands en temps. Mais en vérité ce n'est pas le cas ; la raison est simple : lorsque l'on compile un programme, on compile tout le projet d'un coup, y compris les bibliothèques ! Cela fait rapidement plus de 10 000 lignes ! Même un compilateur quadratique (en la longueur du code) serait trop lent.

Dans la pratique, on ne peut utiliser que deux types d'algorithmes : des algorithmes quasi-linéaires (nlogn en la longueur du code) et des algorithmes compositionels (que l'on applique séparément sur chaque procédure/classes/fonction).

Interpréteurs†. L'interpréteur, lui, doit être linéaire en fonction du temps d'exécution. De plus, les constantes doivent rester petites, autrement on perd en efficacité. Dans la pratique, une grade partie des vérifications et optimisations du compilateur ne sont pas faisable sur un interpréteur, car elle ralentiraient trop l'application au démarrage (même si on est plus efficaces par la suite). De plus, la grande majorité des optimisations ne font pas gagner beaucoup de temps et ne valent pas le coup de les rechercher à moins de les réutiliser des milliers de fois, ce qui n'est pas le cas de l'interpréteur.

1.5.2 Context awareness†

Compilateurs†. Vu qu'il ne connaît pas le contexte d'exécution, un compilateur doit toujours faire attention à gérer tous les cas. En particulier, il doit faire attention à ne pas dépasser sa mémoire allouée, à ne rien effacer par erreur, etc... Ce genre de contrainte peuvent rendre la compilation très complexe.

Interpréteurs†. Un interpréteur, lui, peut connaître le contexte d'exécution, il peut donc gérer plus facilement bon nombre de situations. Attention néanmoins, il n'est pas possible de tout connaître de l'état de l'ordinateur sur lequel

on tourne, et il est souvent coûteux de se rappeler de tout ce qui a été fait, du coup une bonne partie du contexte reste inconnu...

1.5.3 Avantage du compilateur

Efficacité. Sous ces conditions, un programme compilé est (presque) toujours plus rapide qu'un programme interprété.

Vérification. Pour ce qui est des propriétés de vérification, l'avantage est encore plus flagrant : Dans un langage interprété, il faut attendre d'exécuter le programme dans le bon environnement pour voire les erreurs, même les plus bêtes ; au contraire, les compilateurs récents envoient des erreurs ou des warnings pour pas mal de cas non évidents.

1.5.4 Avantage de l'interpréteur

Développement Pour le développement, avoir un interpréteur évite de perdre du temps à compiler au moindre petit test...

Universalité. Sans savoir par qui et comment un programme sera utilisé, on ne peut pas envoyer de code assembleur car toute les machines n'utilisent pas le même. Afin d'être compatible avec tout le monde sans avoir à obliger l'utilisateur à compiler le code, on va plutôt utiliser un code à interpréter. C'est pour ça que tous les langages de scripts sont interprétés. (et que quasiment tous les langages interprétés sont des langages de script...)

Mises à jours. Enfin, les langages interprétés peuvent être utiles pour des applications serveurs que l'on doit modifier souvent. En effet, cela permet d'éviter de couper le serveur le temps de faire la mise à jours. C'est pour cette raison que le PHP est interprété.

1.5.5 Stratégie hybrides†

Avoir les deux. De nombreux langages disposent à la fois d'un interpréteurs et d'un compilateur pour profiter de leurs avantages respectifs. Évidement, on ne peux pas utiliser les deux à la fois : on utilisera l'un ou l'autre selon les utilisations. De plus, on ne récupère pas tous les avantages pour des raisons de conceptions.

Cela permet, par exemple, d'avoir du PHP plus efficace (même s'il ne sera jamais aussi efficace que les autres langages compilés car il n'a pas été conçu pour être compilé) et au Haskell de pouvoir se tester rapidement en phase de développement (par contre l'interpréteur est trop inefficace pour tout autre utilisation).

La compilation à l'écriture et à la volée. Pour rattraper leur défauts implicites, les langages interprétés ont un peu de compilation : Pendant que vous écrivez votre code, certains IDEs font quelques vérifications statique (oubli de " ;", mais aussi utilisation des variables...).

Juste avant l'exécution, certains langages interprétés peuvent aussi faire de la compilation à la volée, c'est à dire qu'ils font de petites passes de compilation, appelées Jit (pour "just in time"), pour récupérer quelques optimisations importantes.

La JVM En Java (ainsi qu'en Scala), la compilation s'arrête à la fin du middleend et il n'y a pas de Backend. La sortie du compilateur (la commande `javac`) est un binaire de l'assembleur abstrait de java, le *Java bitecode*. Le rôle du Backend est joué par un programme de virtualisation de la JVM (la machine abstraite de Java) sur lequel on va exécuter le code assembleur abstrait.

2 Frontend

2.1 Programmes et ASTs

Structure statique d'un programme Un programme n'est qu'une chaîne de caractères. Pourtant, un programmeur expérimenté qui voit du code repère immédiatement les différentes fonctions, variables, boucles, etc...

C'est ce que l'on appel la structure statique du programme : on n'a pas besoin de comprendre un programme, et encore moins de l'exécuter pour en voir la structure. La très grande majorité des langages de programmation imposent énormément de structure statique à leurs programme. Celle-ci est capturée par la notion d'AST.

ASTs Un arbre syntactique abstrait, ou AST (pour *Abstract Syntactic tree*), un l'arbre dont les noeuds sont des opérateurs (au sens large : `if..then.else` est un opérateur, comme l'est la déclaration d'une boucle ou d'une fonction) et ses fils sont ses arguments.

Le but principal du frontend est de prendre un programme sous forme de fichier texte et d'en extraire son AST.

Information sémantique En plus de l'AST, le frontend doit aussi extraire des informations dites sémantique, comme le typage (pour les langage au typage statique), mais aussi les listes des fonctions, classes, variables globales, etc, qui peuvent être utilisés dans le programme avant leur définition.

Programmes sans structure statique[†] Certains (rares) langages interprétés ont une structure qui dépend de leur exécution (nombre d'arguments variables, etc...) ; leur interpréteur ne peut pas commencer par une frontend classique. C'est aussi le cas d'une grande partie des structures de fichiers qui ne sont pas prévus pour être manipulés par l'homme (ex : vidéo) et qui contiennent des

sous-structures encodés dans des normes très différentes (ex : un PDF peut contenir des image, des vidéo, des liens, etc...).

2.2 Analyse lexicale

L'analyse lexicale consiste à transformer un code sous forme textuelle en une suite de token, séparant ainsi les mots-clés, les variables, les entiers, etc...

2.2.1 Théorie

Token Un token est une boite contenant une donnée et indiquant ce que c'est. Voir le cours de frontend pour une définition plus formelle.

Lexem Un lexem est une chaîne de caractère représentant une composante atomique de notre code. Ainsi la chaîne `"2389"` est un lexem, tout comme `"maVariable"`, mais la chaîne `"25+3"` contient 3 lexems, qui sont `25`, `+` et `3`.

Lexeur Un lexeur est un programme qui prend une chaîne de caractères, la décompose en lexems, met chaque lexem dans un token et rend une séquence de token avec les lexems dedans. Ainsi, sur la chaîne `"25+3"`, un lexeur rendra quelque-chose comme `[<NOMBRE>("25"),<OPERATION>("+"),<NOMBRE>("3")]`, c'est à dire une suite de trois tokens, le premier nommé `<NOMBRE>` et contenant `"25"`, etc...

Générateur de lexeur Il est possible d'écrire son lexeur à la main. Mais lorsque l'on a beaucoup de tokens différents et/ou que l'on cherche à être très rapide, on utilise un générateur de lexeur. Celui-ci prend, pour chaque token, une expression régulière [2] identifiant les lexems que l'on peut mettre dans ce token. Il produit un lexeur qui décompose la chaîne de caractère en entrée en suite de token contenant un lexem accepté. Le lexeur généré est très rapide car il utilise toute la machinerie automate pour rester en temps linéaire. [3]

Échec et choix des lexeurs générés Il se peut qu'il n'y ai pas moyen de décomposer le texte entré en lexems correctes, dans le programme n'est pas correcte et on considère qu'il y a une erreur lexical. Il se peut aussi qu'il y ai plusieurs décompositions correctes, dans ce cas, le générateur fait des choix de priorité, comme décrits dans le cours de frontend. Enfin, il se peut aussi que le lexeur généré n'arrive pas à trouver la décomposition correcte en lexems (car il a fait de mauvais choix au début lorsqu'il y avait plusieurs possibilités), dans ce cas on renvoi aussi une erreur lexical, mais c'est situations sont très très rare dans la pratique.

2. Si vous ne savez pas ce qu'est une expression régulière, un rappel peut se trouver dans le cours de frontend.

3. En fait quadratique dans des cas horribles à cause de la stratégie de choix, voir le cours de frontend.

2.2.2 Pratique

Séparateurs En pratique, les choses ne sont pas aussi propre qu'en théorie. Le premier bémol est celui des séparateurs : on peut ajouter des lexèmes que disparaissent et ne créent pas de token. C'est le cas des espace, des retours à la ligne, ou des commentaires par exemple. On les appel des séparateurs car on les utilisent en principalement pour séparer des tokens

Transformation du contenu du token Un autre manquement à la théorie : on transforme tout de suite le contenu des tokens pour prendre moins de place. Par exemple, dans un token devant contenir un entier, on ne met pas la chaîne de caractère, mais l'entier représenté ; ainsi le passage du lexeur sur `"25+3"` rendrait plutôt quelque chose comme `[<NOMBRE>(25),<OPERATION>("+"),<NOMBRE>(3)]`.

D'autres changements "sales" On peut ajouter d'autres bidouillage, par exemple pour lexer le python, il faut se souvenir de la taille de la dernière indentation et, à chaque retour à la ligne, renvoyer un token de fermeture de bloc, un token d'ouverture de bloc ou rien selon que la nouvelle indentation soit plus petite, plus grande ou égale...

Disparition du (générateur de) lexeur† Dans les générateurs de passeur récent, le générateur de lexeur est inclus et le générateur produit fait l'analyse lexical et syntaxique à la fois, cela permet d'utiliser plusieurs lexeurs à la fois (en changeant suivant le contexte syntaxique), d'être plus rapide, et d'éviter des erreurs lexicales. On pourrait faire de même à la main, mais c'est vite très complexe.

2.3 Analyse syntactique

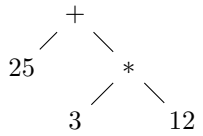
L'analyse syntactique est le passage de la séquence de tokens à l'AST du programme.

L'insuffisance de l'analyse lexicale Mise à part dans le cas d'un programme assembleur (qui est constitué une suite d'instructions extrêmement simples), l'analyse lexical ne permet pas d'explorer toute la structure du programme. Ainsi dans un petit programme comme `print 3+5`, il faut savoir que le `print` s'applique au résultat de `3+5`, il faut donc voir `3+5` comme une entité unique, alors qu'il s'agit de trois tokens... Ce qu'il faut récupérer c'est l'arborescence de sous programme appelée AST.

2.3.1 Théorie

Parseur Un parseur est un programme qui prend une liste de tokens, et la structure pour former un arbre étiqueté représentant l'imbrication des instructions. Ainsi, la liste `[<NOMBRE>("25"),<OPERATION>("+"),<NOMBRE>("3"),`

`<OPERATION>("*"),<NOMBRE>("12")]`, deviendra, par exemple, l'arbre



Arbre syntaxique Un arbre syntaxique pour une grammaire [4] est un arbre dont les feuilles sont étiquetés par des terminaux (souvent des token) et les noeud par des non terminaux de la grammaire, et tel que pour tous noeud étiqueté par un non terminal N dont les fils sont étiquetés par des terminaux et non terminaux formant un mot ω, $N \mapsto \omega$ est une règle de la grammaire. Voir le cours de frontend pour plus de détails.

Générateurs de parseurs Il est vite très difficile d'écrire le parseur à la main, on utilise donc un générateur de lexeur. Celui-ci prend en entrée une grammaire dont les terminaux sont les noms des token. Il produit un parseur qui transforme toute liste de token correcte en un arbre syntaxique dont les feuilles, lues de gauche à droite, forment la liste de tokens entrés. Le parseur résultant s'exécute en temps linéaire.

Échec du parseur généré Le parseur généré va (et doit) échouer lorsqu'il n'y a aucun arbre syntaxique possible pour la liste de tokens entrée. Cela veut dire que le programme n'était pas syntaxiquement correcte (par exemple si il manque une parenthèse).

Ambigüité Mais que se passe-t-il si il y a plusieurs arbres syntaxiques possibles ? En choisir un arbitrairement est dangereux car le programmeur pensais peut-être à l'autre dont le comportement à l'exécution sera complètement différent. C'est pourquoi on ne peut pas légitimement procéder de manière arbitraire et on impose aux grammaires utilisées pour la génération de parseur d'être non-ambigüe, c'est à dire de ne jamais avoir deux arbres syntaxique pour un même mot de terminaux ; dans le cas contraire, la génération de parseur va échouer. Voir le cours de frontend pour plus de détails.

Complexité du problème La réalité est même plus vicieuse que ça : le parsing à partir d'une grammaire arbitraire est en fait un problème superquadratique (même complexité que la multiplication matricielle), ce qui n'est pas envisageable pour de la compilation. On doit donc se restreindre à un sous-ensemble de grammaire, qui, dans les fait, est strictement plus petit que les grammaires non-ambigües. Cela veut dire qu'il y a forcément des grammaires non-ambigües qui échoueront à la génération de parseur. De plus, il existe plusieurs algorithmes de parsing qui reconnaissent plusieurs classes de grammaires.

4. Si vous ne savez pas ce qu'est une grammaire, un rappel peut se trouver dans le cours de frontend.

Générateurs de parseurs ascendants Un premier ensemble d'algorithmes de génération de parseurs sont les algorithmes de génération de parseurs ascendant. Les principaux membres de cette classe sont les générateurs de parseurs LR_0, SLR_k, LALR et LR_1. Tous utilisent de la déterminisation d'automates avec différents niveaux de subtilités. Voir le cours de frontend pour les détails. À retenir que LR_1 est de loin le plus expressif des algorithmes, mais il consomme beaucoup de mémoire ce qui a été longtemps rédhibitoire. Ces algos sont utilisés dans par les générateurs de parseurs qui ont été prévus pour faire de la compilation (et non pas juste du parsing de fichiers ou commandes).

Générateurs de parseurs descendants$^{(\dagger)}$ Un second ensemble d'algorithmes de génération de parseurs sont les algorithmes de génération de parseurs descendants. Les principaux membres de cette classe sont les générateurs de parseurs LL_1, LL_k et LL_*. L'idée de LL_1 est de décider quelle règle appliquer à dès le premier token vu, pour LL_k, il existe un entier k tel qu'il suffit de regarder les k premiers tokens pour choisir la règle, enfin, dans LL_*, le nombre de token à regarder est non borné, mais on ne peux que vérifier qu'ils respectent une expression régulière. La majorité des générateur de parseurs Java, comme JavaCC ou Antler, mais aussi de plusieurs langages inspirés (Python, Scala...) utilisent des algorithmes descendants. Les algorithmes descendants sont aussi utilisés pour les combinateurs de parseurs (voir le paragraphe 2.3.3 dédié) car ils ne demandent pas de post-traitement [5] et sont dont plus applicatifs.

Comparaison entre les algos classiques LL_1 est moins expressif que LL_k qui l'est moins que LL_*, de plus LR_0 est moins expressif que SLR qui l'est moins que LALR qui l'est moins que LR_1. La seule autre inclusion est LL_1 qui est moins expressif que LR_1, les autres sont formellement incomparables. Néanmoins, Il y a peu d'exemples qui sont LL_* et pas LALR en choisissant correctement l'ensemble de tokens, tendit qu'il y a de nombreux exemples pratiques qui sont SLR mais pas LL_*. À noter aussi que LR_0 est inutile en pratique car très peu expressif...

Autres générateurs† Il existe quelques autres générateurs, mais ils sont pour la plupart anecdotiques ou spécifique à certaines problématiques, comme les algorithmes de Earley et GLR qui sont les algos capturant aussi des grammaires ambigües ou contextuelles au prix d'une complexité élevée.

2.3.2 Pratique

Priorité En pratique, il y a des grammaires ambigües que l'on a envie d'utiliser, comme par exemple :

$$S := \langle \texttt{INT} \rangle \mid S{+}S \mid S{*}S$$

5. entendre déterminisation et minimisation des algos LR

Il y a Ambigüité, car on ne sais pas si $1 + 2 * 3$ veut dire $(1 + 2) * 3$ ou $1 + (2 * 3)$. Mais on est tous d'accord pour dire que c'est $1 + (2 * 3)$. C'est parce que la multiplication est connue comme prioritaire sur l'addition. Dans certains générateurs de parseurs (généralement les générateurs de type LR), on peut préciser de tels priorités entre règles pour désembigüiser la grammaire. Voir le cours de frontend pour les détails.

Associativité Donner une priorité ne permet pas tout à fait de désembigüiser la grammaire ci-dessus car, par exemple, on ne sais pas si $1 + 2 + 3$ veut dire $(1 + 2) + 3$ ou $1 + (2 + 3)$. Moralement, ce n'est pas grave car l'addition est associative, mais ce n'est pas le cas de tous les opérateurs et l'ordinateur ne le sait pas. Plutôt que de préciser que l'on est complètement associatif, on choisi plutôt de préciser une semi-associativité : on parle alors d'associativité à gauche ($1+2+3$ vaut $(1+2)+3$) et associativité à droite ($1+2+3$ vaut $1+(2+3)$). Cela permet de préciser la semi-associativité d'opérateurs non associatifs. Exemple : l'application en OCaml/Haskell est associative à gauche tendit que la flèche des types est associative à droite.

Le A de AST Le mot AST veut dire *abstract syntactic tree*, ce n'est donc pas l'arbre syntaxique mais une version "abstraite". Celà veut dire que l'on se permet de faire des simplifications à la volée, comme d'enlever les parenthèses (qui ne sont plus utile une fois que l'on a construit l'arbre). C'est une étape que l'on écrit directement dans le générateur de parseur.

Interpréter sans générer l'AST Lorsque l'on veut juste interpréter le langage parsé et que celui-ci est simple, on peut directement faire l'interprétation dans le parseur sans générer d'AST, on gagne ainsi du temps, et surtout de la mémoire. C'est ce que l'on fait, par exemple, dans le second exercice du TP avec la calculatrice : on calcule le résultat à la volée.

2.3.3 D'autres types d'outils†

Parser la grammaire du langage utilisé Lorsque l'on travail dans un langage et que l'on a besoin de parser ce même langage, on a généralement envie de récupérer le parseur fournit avec le compilateur (qui sera probablement meilleurs que ce qu'on écrirait). C'est généralement possible grâce à des outils *ad hoc*, ceux-ci permettent même souvent d'élargir la syntaxe de manière relativement naturelle.

Combinateurs de parsing Les générateurs de parseurs dynamique ont la cote de nos jours : il s'agit de générateurs de parseurs qui sont intégralement inclue dans notre code, le parseur étant calculé dynamiquement juste avant le parsing. Celà permet de faire du parsing même si on ne connaît pas en avance l'intégralité de la syntaxe (on peut rajouter des bouts de syntaxe pendant l'exé-

cution). Par contre, on perd évidement en efficacité du parseur et souvent aussi en pouvoir expressif du générateur.

2.4 Analyse sémantique$^{(\star)}$

3 Backend et Middleend

3.1 Langages assembleurs

3.2 Machine virtuelle*

3.3 Optimisations*†

3.4 Vérification*†

3.5 GC*†

4 Structure des compilateurs commerciaux*†

4.1 Langages intermédiaires*†

4.2 Multiples frontends*†

4.3 Multiples phases*†

4.4 Encore d'autres vérifications*†

4.5 Encore d'autres optimisations*†

4.6 Machines virtuelles*†

4.7 Sémantique et compilateur/interpréteurs de référence†