Compilateur

En informatique, un **compilateur**[1] est un programme qui transforme un code source en un code objet[2]. Généralement, le code source est écrit dans un langage de programmation (le *langage source*), il est de haut niveau d'abstraction, et facilement compréhensible par l'humain. Le code objet est généralement écrit en langage de plus bas niveau (appelé *langage cible*), par exemple un langage d'assemblage ou langage machine, afin de créer un programme exécutable par une machine.

Présentation générale

Un compilateur effectue les opérations suivantes : analyse lexicale, pré-traitement (préprocesseur), analyse syntaxique (*parsing*), analyse sémantique, et génération de code optimisé. La compilation est souvent suivie d'une étape d'édition des liens, pour générer un fichier exécutable. Quand le programme compilé (code objet) est exécuté sur un ordinateur dont le processeur ou le système d'exploitation est différent de celui du compilateur, on parle de compilation croisée.

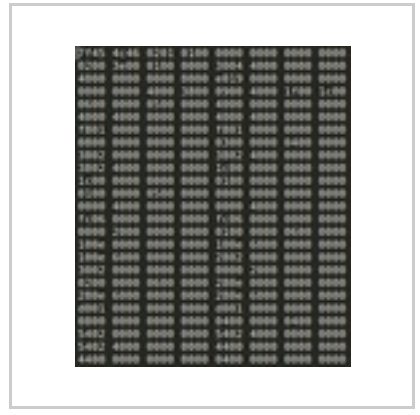
On distingue deux options de compilation :

- *Ahead-of-time* (AOT), où il faut compiler le programme avant de lancer l'application : c'est la situation traditionnelle.
- Compilation à la volée (*just-in-time*, en abrégé JIT) : cette faculté est apparue dans les années 1980 (par exemple avec Tcl/Tk).

La chaine de compilation

Un programme source en langage C

le code d'assemblage correspondant



le programme après la compilation - langage machine affiché en hexadécimal

Historique

Les logiciels des premiers ordinateurs[3] étaient écrits en langage assembleur[4]. Les langages de programmation de plus haut niveau (dans les couches d'abstraction) n'ont été inventés que lorsque les avantages apportés par la possibilité de réutiliser le logiciel sur différents types de processeurs sont devenus plus importants que le coût de l'écriture d'un compilateur. La capacité de mémoire très limitée des premiers ordinateurs a également posé plusieurs problèmes techniques dans le développement des compilateurs.

Vers la fin des années 1950, des langages de programmation indépendants des machines font pour la première fois leur apparition. Par la suite, plusieurs compilateurs expérimentaux sont développés. Le premier compilateur, A-0 System (pour le langage A-0) est écrit par Grace Hopper[5], en 1952. L'équipe FORTRAN dirigée par John Backus d'IBM est considérée comme ayant développé le premier compilateur complet[4], durant la période 1954-1957, et il s'agit du premier compilateur optimiseur, l'objectif de l'équipe étant de générer un code en langage machine quasiment aussi rapide que celui qu'aurait généré un programmeur[6]. COBOL, développé en 1959 et reprenant largement des idées de Grace Hopper[7,8] est le premier langage à être compilé sur plusieurs architectures.

Dans plusieurs domaines d'application[Lesquels ?], l'idée d'utiliser un langage de plus haut niveau d'abstraction s'est rapidement répandue. Avec l'augmentation des fonctionnalités supportées par les langages de programmation plus récents et la complexité croissante de l'architecture des ordinateurs, les compilateurs se sont de plus en plus complexifiés.

En 1962, le premier compilateur « auto-hébergé » - capable de compiler en code objet, son propre code source exprimé en langage de haut niveau - est créé, pour le Lisp, par Tim Hart et Mike Levin au Massachusetts Institute of Technology (MIT). À partir des années 1970, il est devenu très courant de développer un compilateur dans le langage qu'il doit compiler, faisant du Pascal et du C des langages de développement très populaires.

On peut aussi utiliser un langage ou un environnement spécialisé dans le développement de compilateurs : on parle lors d'outils de méta-compilation, et on utilise par exemple un compilateur de compilateur. Cette méthode est particulièrement utile pour réaliser le premier compilateur d'un nouveau langage ; l'utilisation d'un langage adapté et rigoureux[Par exemple ?] facilite ensuite mise au point et évolution.

Structure et fonctionnement

La tâche principale d'un compilateur est de produire un code objet correct qui s'exécutera sur un ordinateur. La plupart des compilateurs permettent d'optimiser le code, c'est-à-dire qu'ils vont chercher à améliorer la vitesse d'exécution, ou réduire l'occupation mémoire du programme[4].

En général, le langage source est « de plus haut niveau » que le langage cible, c'est-à-dire qu'il présente un niveau d'abstraction supérieur. De plus, le code source du programme est généralement réparti dans plusieurs fichiers.

Un compilateur fonctionne par analyse-synthèse : au lieu de remplacer chaque construction du langage source par une suite équivalente de constructions du langage cible, il commence par analyser le texte source pour en construire une représentation intermédiaire qu'il traduit à son tour en langage cible.

On sépare le compilateur en au moins deux parties : une partie avant (ou frontale), parfois appelée « souche », qui lit le texte source et produit la représentation intermédiaire ; et une partie arrière (ou finale), qui parcourt cette représentation pour produire le texte cible. Dans un compilateur idéal, la partie avant est indépendante du langage cible, tandis que la partie arrière est indépendante du langage source. Certains compilateurs effectuent des traitements substantiels sur la partie intermédiaire, devenant une partie centrale à part entière, indépendante à la fois du langage source et de la machine cible. On peut ainsi écrire des compilateurs pour toute une gamme de langages et d'architectures en partageant la partie centrale, à laquelle on attache une partie avant par langage et une partie arrière par architecture.

Schéma de compilation multi-source multi-cible.

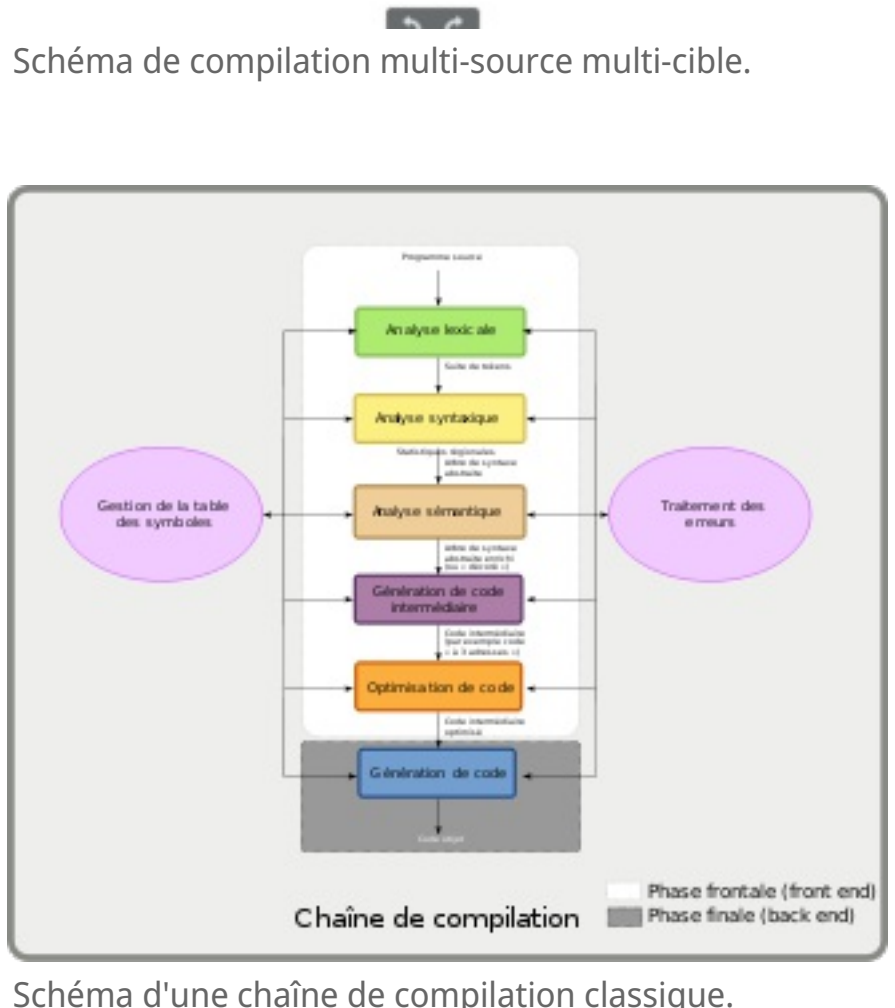


Schéma d'une chaîne de compilation classique.

Les étapes de la compilation incluent :

- le prétraitement, nécessaire pour certains langages comme C, qui prend en charge la substitution de macro et de la compilation conditionnelle.

  Généralement, la phase de prétraitement se produit avant l'analyse syntaxique ou sémantique ; par exemple dans le cas de C, le préprocesseur manipule les symboles lexicaux plutôt que des formes syntaxiques.

- l'analyse lexicale, qui découpe le code source en petits morceaux appelés *jetons* (*tokens*).

  Chaque jeton est une unité atomique unique de la langue (unités lexicales ou lexèmes), par exemple un mot-clé, un identifiant ou un symbole. La syntaxe de jeton est généralement un langage régulier, donc reconnaissable par un automate à états finis.
  Cette phase est aussi appelée *à balayage* ou *lexing* ; le logiciel qui effectue une analyse lexicale est appelé un analyseur lexical ou un scanner. Un analyseur lexical pour un langage régulier peut être généré par un programme informatique, à partir d'une description du langage par des expressions régulières. Deux générateurs classiques sont lex et flex.

- l'analyse syntaxique implique l'analyse de la séquence jeton pour identifier la structure syntaxique du programme.

  Cette phase s'appuie généralement sur la construction d'un arbre d'analyse ; on remplace la séquence linéaire des jetons par une structure en arbre construite selon la grammaire formelle qui définit la syntaxe du langage. Par exemple, une condition est toujours suivie d'un test logique (égalité, comparaison...). L'arbre d'analyse est souvent modifié et amélioré au fur et à mesure de la compilation. Yacc et GNU Bison sont les analyseurs syntaxiques les plus utilisés.

- l'analyse sémantique est la phase durant laquelle le compilateur ajoute des informations sémantiques à l'arbre d'analyse et construit la table des symboles.

  Cette phase vérifie le type (vérification des erreurs de type), ou l'objet de liaison (associant variables et références de fonction avec leurs définitions), ou une tâche définie (toutes les variables locales doivent être initialisées avant utilisation), peut émettre des avertissements, ou rejeter des programmes incorrects.
  L'analyse sémantique nécessite habituellement un arbre d'analyse complet, ce qui signifie que cette phase fait suite à la phase d'analyse syntaxique, et précède logiquement la phase de génération de code ; mais il est possible de replier ces phases en une seule passe.

- la transformation du code source en code intermédiaire ;
- l'application de techniques d'optimisation sur le code intermédiaire : c'est-à-dire rendre le programme « meilleur » selon son usage (voir *infra*) ;
- la génération de code avec l'allocation de registres et la traduction du code intermédiaire en code objet, avec éventuellement l'insertion de données de débogage et d'analyse de l'exécution ;
- et finalement l'édition des liens.

L'analyse lexicale, syntaxique et sémantique, le passage par un langage intermédiaire et l'optimisation forment la partie frontale. La génération de code et l'édition de liens constituent la partie finale.

Ces différentes étapes font que les compilateurs sont toujours l'objet de recherches.

Lien avec les interpréteurs

L'implémentation (réalisation concrète) d'un langage de programmation peut être interprétée ou compilée. Cette réalisation est un compilateur ou un interpréteur, et un langage de programmation peut avoir une implémentation compilée, et une autre interprétée.

On parle de compilation si la traduction est faite avant l'exécution (le principe d'une boucle est alors traduit une fois), et d'interprétation si la traduction est finie pas à pas, durant l'exécution (les éléments d'une boucle sont alors examinés à chaque usage).

L'interprétation est utile pour la mise au point ou si les moyens sont limités. La compilation est préférable en exploitation.

Problème d'amorçage (bootstrap)

Les premiers compilateurs ont été écrits directement en langage assembleur, un langage symbolique élémentaire correspondant aux instructions du processeur cible et quelques structures de contrôle légèrement plus évoluées. Ce langage symbolique doit être *assemblé* (et non compilé) et lié pour obtenir une version exécutable. En raison de sa simplicité, un programme simple suffit à le convertir en instructions machines.

Les compilateurs actuels sont généralement écrits dans le langage qu'ils doivent compiler ; par exemple un compilateur C est écrit en C, SmallTalk en SmallTalk, Lisp en Lisp, etc. Dans la réalisation d'un compilateur, une étape décisive est franchie lorsque le compilateur pour le langage X est suffisamment complet pour se compiler lui-même : il ne dépend alors plus d'un autre langage (même de l'assembleur) pour être produit.

Il est complexe de détecter un bogue de compilateur. Par exemple, si un compilateur C comporte un bogue, les programmeurs en langage C auront naturellement tendance à mettre en cause leur propre code source, non pas le compilateur. Pire, si ce compilateur buggé (version V1) compile un compilateur (version V2) non buggé, l'exécutable compilé (par V1) du compilateur V2 pourrait être buggé. Pourtant son code source est bon. Le *bootstrap* oblige donc les programmeurs de compilateurs à contourner les bugs des compilateurs existants.

Compilateur simple passe et multi passe

La classification des compilateurs par nombre de passes a pour origine le manque de ressources matérielles des ordinateurs. La compilation est un processus coûteux et les premiers ordinateurs n'avaient pas assez de mémoire pour contenir un programme devant faire ce travail. Les compilateurs ont donc été divisés en sous programmes qui font chacun une lecture de la source pour accomplir les différentes phases d'analyse lexicale, d'analyse syntaxique et d'analyse sémantique.

L'aptitude à combiner le tout en un seul passage a été considérée comme un avantage, car elle simplifie l'écriture du compilateur, qui s'exécute généralement plus rapidement qu'un compilateur multi passe. Ainsi, dus aux ressources limitées des premiers systèmes, de nombreux langages ont été spécifiquement conçus afin qu'ils puissent être compilés en un seul passage (par exemple, le langage Pascal).

Compilateur multi-passes typique.

Structure non linéaire du programme

Dans certains cas, telle ou telle fonctionnalité du langage requiert que son compilateur effectue plus d'une passe. Par exemple, considérons une déclaration figurant à la ligne 20 de la source qui affecte la traduction d'une déclaration figurant à la ligne 10. Dans ce cas, la première passe doit recueillir des renseignements sur les déclarations, tandis que la traduction proprement dite ne s'effectue que lors d'un passage ultérieur.

Optimisations

Le fractionnement d'un compilateur en petits programmes est une technique utilisée par les chercheurs intéressés à produire des compilateurs performants. En effet, l'inconvénient de la compilation en une seule passe est qu'elle ne permet pas l'exécution de la plupart des optimisations sophistiquées nécessaires à la génération de code de haute qualité. Il devient alors difficile de dénombrer exactement le nombre de passes qu'un compilateur optimisant effectue.

Fractionnement de la démonstration de correction

Démontrer la correction d'une série de petits programmes nécessite souvent moins d'effort que de démontrer la correction d'un plus grand programme unique équivalent.

Compilateur de compilateur

Un compilateur de compilateur est un programme qui peut générer une, voire toutes les parties d'un compilateur. On peut par exemple compiler les bases d'un langage, puis, utiliser les bases du langage pour compiler le reste.

Qualité

Optimisation

Selon l'usage et la machine qui va exécuter un programme, on peut vouloir optimiser la vitesse d'exécution, l'occupation mémoire, la consommation d'énergie, la portabilité sur d'autres architectures, ou le temps de compilation.

Préservation sémantique

Il existe des compilateurs qui sont vérifiés mathématiquement. Ces compilateurs garantissent que les propriétés de sécurité prouvées sur le code source sont également valables pour le code compilé exécutable[9,10]. Ce type de compilateurs est notamment utilisé pour le développement d'algorithmes de contrôle de vol et de navigation dans l'aviation[11] ou dans le domaine de l'énergie nucléaire[12].

Chaîne de compilation

Compilation croisée

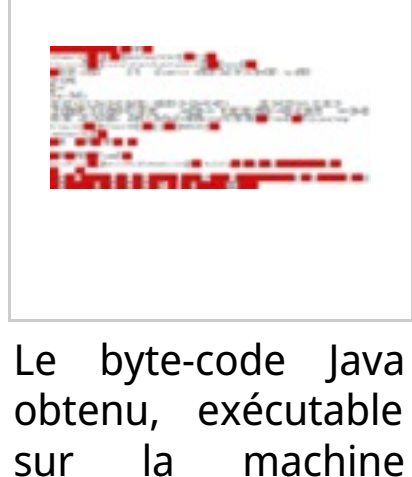
La compilation croisée fait référence aux chaînes de compilation capables de traduire un code source en code objet dont l'architecture processeur diffère de celle où la compilation est effectuée. Ces chaînes sont principalement utilisées en informatique industrielle et dans les systèmes embarqués.

Autres compilations

Byte code ou *code octet*

Certains compilateurs traduisent un langage source en langage machine *virtuel* (dit langage intermédiaire), c'est-à-dire en un code (généralement binaire) exécuté par une machine virtuelle : un programme émulant les principales fonctionnalités d'un ordinateur. De tels langages sont dits semi-compilés. Le portage d'un programme ne requiert ainsi que le portage de la machine virtuelle, qui sera de fait soit un interprète, soit un second traducteur (pour les compilateurs multi-cibles). Ainsi, des compilateurs traduisent Pascal en P-Code, Modula 2 en M-Code, Simula en S-code, ou plus récemment du code Java en bytecode Java (code objet).

Un court programme en Scala.



Le byte-code Java obtenu, exécutable sur la machine virtuelle.

Exemples

Quand la compilation repose sur un byte code, on parle de compilation à la volée. On utilise alors des machines virtuelles comme la machine virtuelle Java avec laquelle on peut notamment compiler du Scala. Il est possible dans certains langages d'utiliser une bibliothèque permettant la compilation à la volée de code entré par l'utilisateur, par exemple en C avec libtcc[13].

D'autres compilateurs traduisent un code d'un langage de programmation vers un autre. On les appelle des transcompilateurs, ou bien encore par anglicisme, des transpileurs ou transpilateurs[14]. Par exemple, le logiciel LaTeX[15] permet, à partir d'un code source en LaTeX, d'obtenir un fichier au format PDF (avec par exemple la commande pdflatex sous Ubuntu) ou HTML. Autre exemple, LLVM est une bibliothèque aidant à réaliser des compilateurs, également utilisée par AMD pour développer « HIP », un transcompilateur de code CUDA (langage spécifique à NVIDIA et très utilisé) afin de l'exécuter sur les processeurs graphiques d'AMD.

Le code source.

Le code obtenu après compilation.

Aperçu du document pdf.

Certains compilateurs traduisent, de façon incrémentale ou interactive, le programme source (entré par l'utilisateur) en code machine. On peut citer comme exemple certaines implantations de Common Lisp (comme SBCL (en)[16]).

Annexes

Sur les autres projets Wikimedia :
Compilation, sur Wikiversity

Bibliographie

- Alfred Aho, Monica Lam, Ravi Sethi et Jeffrey Ullman (trad. de l'anglais par Philippe Deschamp, Bernard Lorho, Benoît Sagot et François Thomasset), *Compilateurs : Principes, techniques et outils* [« Compilers: Principles, Techniques, and Tools »], France, Pearson Education, novembre 2007, 2e éd. (1re éd. 1977), 901 p. (ISBN 978-2-7440-7037-2, présentation en ligne (http://dragon2007.free.fr/))

  Appelé aussi le *Dragon book*

Articles connexes

- Interprète
- Low Level Virtual Machine
- Compilation à la volée
- Compilation incrémentale
- Compilation anticipée
- Décompilateur, programme qui traduit un langage de bas niveau vers un langage de plus haut niveau
- GCC est une suite de compilation particulièrement connue, beaucoup utilisée pour les langages C et C++, mais également Java ou encore Ada.
- Clang est un front-end pour les langages de la famille du C, utilisant le back-end LLVM
- Javac, le compilateur Java le plus répandu
- GHC, un compilateur pour Haskell
- De nombreux autres[Lesquels ?], pour les mêmes langages et pour d'autres[Lesquels ?]

Liens externes

- (en) Liste de compilateurs gratuits et/ou libres (http://www.idiom.com/free-compilers)
- Cours plutôt complet (http://pauillac.inria.fr/~maranget/X/compil/poly/index.html) et contenant des exemples en C/ASM.

Notes et références

1. Un compilateur désigne aussi un auteur qui assemble des extraits d'écrits divers pour en faire une œuvre ; voir le sens 1 du mot *compilateur* dans le Wiktionnaire.
2. Jacques Menu, *Compilateurs avec C++*, Addison-Wesley, 1994, p. 30
3. Par exemple sur l'EDSAC, comme le décrit Alan Turing dans sa conférence, lors de l'inauguration, (en) A. M. Turing, « Checking a Large Routine », dans *Report of a Conference on High Speed Automatic Calculating Machines*, *Univ. Math. Lab.*, Cambridge, p. 67-69 (1949) in Morris, F. L. et C. B. Jones, « An Early Program Proof by Alan Turing », *Ann. Hist. Comp.*, vol. 6, no 2, avril 1984, p. 139-143 (lire en ligne (https://fi.ort.edu.uy/innovaportal/file/20124/1/09-turing_checking_a_large_routine_earlyproof.pdf)).
4. Cf. Jérôme Feldman et Marcel Berger (dir.), *Les progrès des mathématiques*, Paris, éditions Belin, coll. « Pour la Science », 1981, 167 p. (ISBN 2-902918-14-3), « Les langages de programmation », p. 102-113.
5. (en) Susan Ware (dir.), Stacy Braukman *et al.*, *Notable American Women : A Biographical Dictionary : Completing the Twentieth Century*, vol. 5, Harvard University Press, 2005, 729 p. (ISBN 978-0-674-01488-6, présentation en ligne (http://www.hup.harvard.edu/catalog.php?isbn=9780674014886)), p. 309-311.
6. (en) John Backus, « The history of FORTRAN I, II, and III », *ACM SIGPLAN Notices*, vol. 13, no 8, août 1978, p. 165-180 (DOI 10.1145/960118.808380 (https://dx.doi.org/10.1145/960118.808380), lire en ligne (http://www.softwarepreservation.org/projects/FORTRAN/paper/p165-backus.pdf)).
7. Vicki Porter Adams, « Captain Grace M. Hopper: the Mother of COBOL », *InfoWorld*, vol. 3, no 20, 5 octobre 1981, p. 33 (ISSN 0199-6649 (https://www.worldcat.org/issn/0199-6649&lang=fr), lire en ligne (https://books.google.co.uk/books?id=JT0EAAAAMBAJ&pg=RA1-PA33)).
8. Mitch Betts, « Grace Hopper, mother of Cobol, dies », *Computerworld*, vol. 26, no 1, 6 janvier 1992, p. 14 (ISSN 0010-4841 (https://www.worldcat.org/issn/0010-4841&lang=fr), lire en ligne (https://books.google.co.uk/books?id=J-_T3bxgvMwC&pg=PA14)).
9. (en) Xavier Leroy, « A Formally Verified Compiler Back-end », *Journal of Automated Reasoning*, vol. 43, no 4, 4 novembre 2009, p. 363 (ISSN 1573-0670 (https://www.worldcat.org/issn/1573-0670&lang=fr), DOI 10.1007/s10817-009-9155-4 (https://dx.doi.org/10.1007/s10817-009-9155-4), lire en ligne (https://doi.org/10.1007/s10817-009-9155-4), consulté le 18 novembre 2021)
0. (en) Daniel Kästner, Jörg Barrho, Ulrich Wünsche et Marc Schlickling, « CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler », *INRIA*, 31 janvier 2018, p. 1 (lire en ligne (https://hal.inria.fr/hal-01643290), consulté le 18 novembre 2021)
1. (en) webmaster@absint.com, « CompCert: formally verified optimizing C compiler » (https://www.absint.com/compcert/index.htm), sur *www.absint.com* (consulté le 18 novembre 2021)
2. (en) webmaster@absint.com, « AbsInt: User Story MTU Friedrichshafen » (https://www.absint.com/mtu_fh.htm), sur *www.absint.com* (consulté le 18 novembre 2021)
3. « Découvrez le cours "Compilation à la volée avec libtcc" sur @OpenClassrooms » (https://openclassrooms.com/courses/compilation-a-la-volee-avec-libtcc), sur *OpenClassrooms* (consulté le 21 novembre 2016).
4. « tranpiler » (https://fr.wiktionary.org/wiki/transpiler), sur *wiktionary.org*, 16 novembre 2017 (consulté le 24 avril 2018)
5. (en) « LaTeX - A document preparation system » (https://www.latex-project.org/), sur *www.latex-project.org* (consulté le 21 novembre 2016).
6. (en) « SBCL User Manual » (http://www.sbcl.org/manual/#Editor-Integration), sur *www.sbcl.org* (consulté le 21 novembre 2016).