Rémi Lacroix
Aliénor Latour
Nathanaël Masri

Polytech' Paris-Sud
Cycle ingénieur 2ème année – Spécialité informatique
2010 / 2011

Projet compilation

Responsable : Frédéric Voisin

I. Avancement du projet

Globalement le projet peut être considéré comme complètement terminé. Il existe un certain nombre de problèmes connus qui sont évoqués dans la suite de ce document mais la grande majorité des vérifications contextuelles sont à priori présentes et le code généré est normalement fonctionnel. En particulier, l'exemple fourni au début du projet est correctement compilé. D'autres fichiers de test sont par ailleurs fournis dans le répertoire « exemples » du projet. Les fichiers placés dans le répertoire « contre-exemples » permettent de mettre en évidence la détection correcte d'un certain nombre d'erreurs syntaxiques, la compilation doit donc échouer pour ces fichiers.

Les problèmes restants concernent principalement les vérifications contextuelles et la génération de code liées aux expressions par défaut des attributs statiques et des paramètres de méthode mais n'empêchent pas l'utilisation du compilateur dans la majorité des cas.

L'utilisation du compilateur est très simple. Le nom fichier source doit être impérativement fourni comme premier argument lors du lancement du programme en ligne de commande. Le code généré est alors placé dans un fichier nommé à partir du nom du fichier source. Optionnellement un nom de fichier peut être imposé pour le code généré en fournissant un deuxième argument lors de l'appel du programme Si aucun nom n'est fourni, le fichier de code intermédiaire aura le nom du fichier original suivi de « _gen ».

II. Les vérifications contextuelles

Afin de simplifier au maximum les vérifications contextuelles, nous avons essayé d'effectuer le maximum de vérification possible lors de l'analyse syntaxique. En effet, l'utilisation de Bison nous déchargeait d'une partie du travail en générant tout le code lié à la grammaire. Nous avons donc fait le choix d'interdire directement dans la grammaire certaines constructions comme les affectations chaînées ou les paramètres de méthode avec une valeur par défaut non placée en fin par exemple. Néanmoins il est certain que ces cas un peu particuliers tendent à complexifier la grammaire et dans certains cas il semble plus raisonnable de faire certaines vérifications plus tardivement. C'est par exemple ce que nous avons fait pour les identificateurs spéciaux **this** et **super** que l'on aurait pu considérer comme des mots-clés et traiter avant les vérifications contextuelles mais que nous avons préféré effectuer à ce moment pour ne pas surcharger la grammaire. Dans les vérifications contextuelles sont donc présentes principalement les vérifications qu'il était impossible d'effectuer sans le contexte comme par exemple la vérification des déclarations, des portés ou encore du typage. Nous avons couvert l'ensemble des erreurs contextuelles auxquelles nous avons pensé -à l'exception de la totalité de celles concernant les expressions par défaut- cependant nous ne pouvons affirmer avec certitude qu'il n'en manque aucune autre.

Le passage d'attributs hérités étant impossible avec Bison qui ne supporte que les attributs synthétisés, nous construisons nos structures de données à la volée sans faire immédiatement les vérifications contextuelles puisque « l'environnement » contenant les déclarations précédentes n'est disponible qu'au niveau de la liste des classes. Nous attendons donc d'avoir effectué l'analyse syntaxique du code complet d'une classe avant d'effectuer les vérifications contextuelles. Ceci signifie donc que notre compilateur risque de signaler en premier lieu une erreur syntaxique qui serait pourtant située après une erreur qui sera ensuite détectée à la vérification contextuelle (par exemple une erreur de type). Dès qu'une erreur contextuelle est trouvée, nous affichons un message d'erreur informant l'utilisateur sur son type en précisant le numéro de ligne si possible (chaque nœud de l'arbre syntaxique comporte à cette effet le numéro de la ligne à laquelle il correspond) et nous arrêtons le compilateur.

a) Vérifications contextuelles implémentées

Certaines vérifications sont communes à plusieurs entités parmi les classes, attributs, variables locales, méthodes ou paramètres. Nous vérifions ainsi :

- qu'aucune entité déclarée (il peut s'agir d'une méthode, d'un attribut, d'une variable locale ou d'un paramètre) ne se nomme « this » ou « super » puisqu'il s'agit d'identificateurs spéciaux non traités comme des mots-clés ;
- qu'aucune entité n'a le même nom qu'une autre de même type et même portée ;
- et que tous les types utilisés ont été déclarés auparavant.

Nous avons ensuite des vérifications spécifiques :

- pour une classe, nous vérifions :
  - s'il y a héritage que la classe mère précisée existe et peut être étendue (puisque les types prédéfinis ne le peuvent pas) ;
  - qu'elle n'hérite pas d'elle-même ;
  - que les paramètres du constructeur de sa classe mère sont bien corrects le cas échéant ;
  - et qu'on ne peut pas créer dynamiquement d'objets prédéfinis (classes Entier et Chaine).

- pour un attribut, nous vérifions :
  - qu'il ne masque pas d'attribut de ses classes parentes le cas échéant ;
  - que sa valeur par défaut si elle existe est d'un type compatible avec le type déclaré ;
  - et qu'il est initialisé avec une valeur par défaut ou dans le constructeur s'il est déclaré comme étant constant.

- pour une méthode, nous vérifions :
  - qu'elle n'a pas le même nom qu'un attribut de la classe ;
  - qu'elle n'est déclarée comme redéfinie uniquement si elle est effectivement redéfinie et le cas échéant qu'elle a la même signature que la méthode qu'elle redéfinit donc des paramètres de mêmes types et le même type de retour ;
  - et qu'elle a une type de retour réel compatible avec le type de retour déclaré.

- pour les paramètres de méthode, nous vérifions :
  - que leur valeur par défaut si elle existe est compatible avec le type attendu et qu'elle ne fait pas référence à « this » ou un attribut de la classe de destination (cf. partie II.b).

- pour les arguments passés à une méthode, nous vérifions :
  - qu'ils sont en nombre suffisant (en prenant en compte les valeurs par défaut) ;
  - et qu'ils sont compatibles avec les types attendus.

Par ailleurs, on vérifie :

- qu'on ne manipule pas d'identifiant non déclaré (seuls les identifiants déclarés précédemment sont connus ce qui est relativement naturel pour les classes mais qui est fait artificiellement pour les attributs et les méthodes puisqu'on ne fait la vérification qu'après avoir analysé la classe complètement) ;
- qu'on n'utilise pas de méthode non statique comme une méthode statique (c'est-à-dire avec un nom de classe comme destinataire) ;
- qu'on ne fait pas référence à un attribut non statique dans une méthode statique ;
- qu'on n'utilise « this » ni en dehors d'une classe ni une méthode statique ;

- qu'on n'utilise « super » ni en dehors d'une classe qui a hérité, ni dans une méthode statique ;
- qu'on n'utilise « super » que pour envoyer des messages ;
- qu'on n'utilise « super » lors d'un appel à une méthode que si celle-ci est redéfinie dans la classe (puisque dans le cas contraire « this » et « super » seraient équivalents) ;
- que les opérateurs arithmétiques et de comparaison ne sont utilisés que pour les Entiers ;
- qu'on ne sélectionne un attribut que dans une classe et ses sous-classes puisque les attributs se comportent comme les attributs « protected » de java dans notre langage ;
- qu'on ne sélectionne de manière statique un attribut que s'il est statique ;
- que lors d'une affectation les types sont compatibles ;
- qu'on n'affecte pas de valeur à une constante ;
- que la condition de l'expression conditionnelle d'un if-then-else est de type Entier ;
- et que les parties then et else d'une expression conditionnelle sont de types compatibles.

b) Améliorations possibles

Un des points les plus complexes au niveau des vérifications contextuelles est la vérification des valeurs par défaut des attributs et des paramètres de fonction. Nous savons que notre gestion n'est pas parfaite du tout à ce niveau.
En particulier concernant les paramètres de méthode, se pose la question de savoir dans quel contexte est exécutée l'expression correspondant à la valeur par défaut. En effet si l'utilisateur fait référence à un attribut « foo », fait-il référence à l'attribut « foo » de la classe appelante ou de la classe destinataire ? Il semblerait plus logique de considérer la deuxième possibilité comme étant plus plausible (c'est d'ailleurs le choix fait par C++ par exemple) mais dans notre cas il était largement plus simple de générer du code intermédiaire permettant d'évaluer l'expression par défaut avant l'appel et non juste après, autrement dit dans le contexte de l'appelant et non de l'appelé. Par conséquent pour éviter les problèmes à l'exécution du code, nous avons pris le parti de bloquer toute référence à « this » ou à un attribut non statique dans les paramètres de méthode. Cependant il s'agit clairement d'une solution permettant de palier au manque de temps pour résoudre le problème de manière plus adéquate.
Concernant les attributs statiques, une vérification contextuelle est manquante et son absence fait que le code intermédiaire généré peut parfois planter à l'exécution. En raison des choix que nous avons fait pour la génération du code (qui sont décrits plus largement dans la partie III.b), il ne peut y avoir aucun appel de méthode ou à un constructeur dans l'expression par défaut des attributs statiques sous peine de faire planter l'interpréteur (ce qui signifie en fait que seules des constantes littérales sont actuellement acceptables). Les vérifications contextuelles pour ses expressions par défaut sont bien faites en prenant en compte que le contexte est statique mais nous n'avions pas prévu d'avoir à interdire tout appel et nous avons manqué de temps pour ajouter cette vérification. Par ailleurs, cette limitation semble un peu trop restrictive et il serait probablement plus judicieux de modifier la génération de code pour qu'elle ne soit plus nécessaire plutôt que de se contenter de l'ajouter.

Un autre point d'amélioration possible concerne le traitement des erreurs contextuelles qui pourrait être amélioré pour le confort de l'utilisateur.
Une première étape serait de systématiquement afficher la ligne correspondant à l'erreur détectée. Cela impliquerait principalement de stocker la ligne correspondant à chacune des structures créées lors des vérifications contextuelles. C'est actuellement le cas pour les arbres syntaxiques correspondant aux expressions mais l'information est manquante pour les différentes listes de classe, variables, méthodes, etc manipulées lors des vérifications.
Une seconde amélioration serait de signaler plusieurs erreurs syntaxiques et/ou contextuelles en une

fois plutôt que d'arrêter totalement la compilation à la première erreur détectée. L'utilisateur pourrait ensuite éventuellement corriger plusieurs erreurs en une passe et gagner du temps. Cependant notre décision de ne pas procéder ainsi est motivée par le fait qu'il faudrait alors avoir une détection d'erreur plus intelligente que celle actuelle pour qu'elle soit réellement utile. En effet, une erreur comme l'oubli d'un point-virgule ou d'une parenthèse à des effets de bord très important et le fait de continuer bêtement l'analyse risquerait de noyer l'utilisateur sous un déluge de fausses erreurs directement liées à la première. Dans l'état actuel des choses et au vu de l'objectif avant tout scolaire du projet, il nous a paru plus raisonnable de se limiter à la première erreur rencontrée pour plus de clarté.

III. Génération de code intermédiaire

a) Organisation mémoire des classes

Nous avons fait le choix de stocker en premier lieu dans la pile les adresses des tables des sauts de chaque classe déclarée mais également d'utiliser la pile pour stocker les attributs statiques partagés par toutes les instances d'une classe.

Seule l'adresse de la table des sauts apparaît dans la pile puisqu'elle est en fait stockée dans le tas, sous la forme d'un bloc mémoire possédant autant de champs que la classe possède de méthodes. L'utilisation de tables des sauts permet principalement de simplifier la liaison dynamique en cas d'utilisation du polymorphisme. Étant donné que dans notre langage les méthodes statiques ne peuvent pas être redéfinies, il aurait été possible de les gérer différemment sans table des sauts mais nous avons préféré les intégrer afin d'unifier la gestion des appels. A la suite de l'adresse de la table des sauts d'une classe, se trouvent donc dans la pile les éventuels attributs statiques de la classe. Il aurait été également possible d'allouer un bloc de mémoire dans le tas et de ne stocker là aussi que l'adresse du bloc d'attributs statiques. Pour résumer pour chaque classe on peut retrouver l'adresse de la table des sauts à partir du décalage à effectuer à partir du fond de pile et de même pour chaque attribut statique, on dispose de son index comme un décalage à effectuer à partir du fond de pile.

Adresse table des sauts classe C
Adresse table des sauts classe B
Attribut statique 2 classe A
Attribut statique 1 classe A
Adresse table des sauts classe A

Exemple de l'état de la pile au lancement du programme

En plus des ces éléments propres à chaque classe qui sont définis statiquement au lancement du programme, il est nécessaire de définir l'organisation des instances des classes créées dynamiquement pendant l'exécution.

A chaque fois qu'un objet est instancié, on crée un nouveau bloc de mémoire dans le tas qui possède autant de champs que la classe possède d'attributs non statiques accessibles (en incluant ceux des éventuelles classes parentes) plus un qui permet de stocker l'adresse vers la table des sauts. De cette manière quelque soit le type visible de l'objet on peut appeler, sans connaître son type réel, la bonne méthode. On place l'adresse de la table des sauts dans le champ 0 du bloc et pour chaque attribut de la classe, on connaît l'index correspondant dans le bloc de mémoire.

Par exemple soit les deux classes suivantes : une classe A qui possède un attribut non statique foo et un attribut statique bar et une classe B qui hérite de A et qui possède un attribut non statique baz.

Une instance de B se présente alors de la façon suivante dans le tas :

2	Attribut B::baz
1	Attribut A::foo
0	Adresse de la table des sauts de B

b) Organisation du code correspondant aux classes

Le code généré en premier est le code correspondant aux classes. On y trouve le code des constructeurs et méthodes.

Dans notre implémentation, le constructeur est en fait divisé en deux parties auxquelles correspondent deux étiquettes bien distinctes, ceci afin de gérer plus facilement l'héritage. En effet lors d'un héritage, on veut allouer un unique objet dans le tas (organisé comme décrit plus haut) mais on doit quand même faire appel aux constructeurs de toutes les classes parentes. Nous avons donc établi la séparation suivante avec deux étiquettes :

- <nomdeclasse>_Alloc gère l'allocation de la mémoire dans le tas et l'initialisation du champ correspondant à la table de sauts puis fait appel au constructeur ;
- <nomdeclasse>_Const gère l'appel aux constructeurs parents si nécessaire (en commençant par celui le plus haut dans l'arborescence d'héritage) et comprend le code correspondant au corps du constructeur s'il est défini. Il lance pour finir le processus d'initialisation par défaut des attributs non statiques auquel correspond une troisième étiquette <nomdeclasse>_Init.

Chaque méthode possède une étiquette construite sous la forme <nomdeclasse>_<nomdeméthode>.

Après le code correspondant à la définition des classes, on trouve le code construisant les tables de sauts et les initialisant avec les adresses de différentes méthodes. Ce code gère aussi la réservation de l'espace mémoire nécessaire aux attributs statiques dans la pile et leur éventuel initialisation par défaut. C'est d'ailleurs ce qui provoque le problème évoqué plus haut qui fait que l'utilisation d'une méthode ou d'un constructeur dans l'expression par défaut d'un attribut statique provoque un plantage de l'interpréteur. En effet à ce moment, il n'y a pas encore eu d'appel à l'instruction START puisqu'il est fait juste au début du programme principal. Le pointeur de frame n'est donc pas encore initialisé et il est impossible d'utiliser l'instruction CALL qui est nécessaire pour faire un appel de méthode ou de constructeur. Il faudrait donc séparer l'allocation mémoire de l'initialisation et faire l'initialisation après l'appel à START mais par manque de temps, cela n'a pas été fait.

c) Appels de méthode

Afin de pouvoir générer le code correspondant aux appels de méthodes, il est nécessaire de définir l'organisation de la pile juste avant l'appel sous la forme de ce qu'on appel un tableau d'activation. Ce tableau est parfois utilisé pour stocker la sauvegarde du pointeur de frame et du pointeur d'instruction juste avant de faire le saut à l'instruction correspondant au début de la méthode. Ces données sont en effet indispensables pour pouvoir poursuivre l'exécution du programme lorsque la méthode a terminé son exécution. Cependant dans notre cas, la sauvegarde était effectuée automatiquement par l'interpréteur en utilisant une pile annexe donc nous n'avions pas besoin de gérer cette partie de l'appel. Le tableau d'activation reste indispensable pour pouvoir passer à la méthode des arguments et pour pouvoir récupérer sa valeur de retour. Par ailleurs dans le contexte de la programmation orientée objet, il est également nécessaire de connaître le destinataire de l'appel afin de pouvoir savoir quelle instance est impactée si la méthode utilise des attributs de la classe. Nous avons fait le choix suivant pour l'organisation du tableau d'activation en pile : en premier (au plus bas dans la pile donc) se trouve un espace réservé à la valeur de retour de la

méthode, suivent ensuite les éventuels arguments de la méthode puis au dernier lieu l'adresse de l'instance à qui est destinée l'appel.

Adresse de l'instance destinataire de l'appel
Argument 2
Argument 1
Espace réservé pour la valeur de retour

Exemple dans le cas de l'appel d'une méthode avec deux paramètres

Lors d'un appel à une méthode, on commence par construire ce tableau d'activation et en particulier dans un soucis de simplicité on a fait le choix de remplacer à ce moment les paramètres non fournis de la méthode mais possédant une expression par défaut par la valeur correspondant à l'évaluation de l'expression. Ce choix à le désavantage de faire que l'expression est évaluée dans le contexte de l'appelant et non de l'appelé et implique la vérification contextuelle qui a été évoquée plus haut.

L'adresse de l'instance de destination a été placée en dernier dans la pile dans le but de simplifier la génération de code correspondant à l'appel proprement dit. En effet, il est nécessaire de récupérer dans la table des sauts de l'objet l'adresse de la méthode ce qui implique d'avoir son adresse à disposition.

Dans le corps de la méthode, l'accès aux éléments du tableau est réalisé en utilisant un décalage par rapport au nouveau pointeur de frame. Il s'agit donc d'un décalage négatif, par exemple à l'index -1 on trouve ainsi toujours l'adresse de l'objet destinataire du message. A la fin de la méthode, on place la valeur de retour dans l'espace qui lui a été réservé et on reprend le cours normal de l'exécution. Juste après le retour de la méthode, on dépile tous les éléments du tableau d'activation sauf la valeur de retour qui peut alors être utilisée.

d) Organisation des variables

Juste après l'appel à une méthode, les variables locales à la méthode sont allouées dans la pile. Pour chaque variable locale, on connaît donc son index (positif cette fois) par rapport au pointeur de frame. Pour des soucis de simplification, nous avons fait le choix d'allouer dès le début de la méthode toutes les variables locales dont on pourrait avoir besoin y compris celles des sous-blocs. Ces variables pourraient théoriquement être allouées plus tard lors de l'entrée dans le bloc en question mais se poserait alors le problème de pouvoir indexer ces variables dans la pile. Notre implémentation possède l'inconvénient d'allouer potentiellement des variables locales inutiles car jamais utilisées. Ça peut être le cas par exemple si deux variables x et y sont déclarées respectivement dans le bloc « then » et dans le bloc « else » d'une structure conditionnelle. Par définition seule une des deux variables sera réellement utile.

e) Améliorations possibles

Outre les améliorations relatives à l'évaluation des expressions par défaut des attributs statiques et des paramètres qui ont été largement évoquées plus haut, il faut noter que le code généré n'est globalement pas optimisé que ça soit en terme ne nombre d'instructions ou d'occupation mémoire. En effet en plus de la non optimisation de l'allocation des variables mémoires, il arrive régulièrement que des instructions sans aucun effet soient générées. Ainsi opérations POPN 0 et PUSHN 0 peuvent être générées (par exemple pour effectuer la gestion de la mémoire pour une méthode n'allouant en fait pas de variable locale). Par ailleurs, par soucis de simplicité les étiquettes générées automatiquement sont presque toujours suivies par une opération NOP alors qui pourrait la plupart du temps être supprimée en tenant compte du code qui va suivre.