ELEFTHERIOU Theophanis - ELIEZER Gwennan - GINDRIER Clément
PETITEAU Matthieu - POLICET Rémi

DOCUMENTATION DE L'EXTENSION

Vous trouverez dans ce document les spécificités de notre extension

Equipe GL60

SOMMAIRE

Spécification de l'extension

Introduction

L'extension choisie par notre groupe est l'extension LINK. Elle consiste à faire une compilation séparée, c'est-à-dire à compiler différents codes sources séparément vers des fichiers dits "objets", suivie d'une édition de liens, qui consiste à récupérer plusieurs fichiers objets et en faire un seul exécutable.

Afin de réaliser notre implémentation de l'extension *LINK* du compilateur deca, nous avons eu à modifier chaque étape, et à rajouter une étape supplémentaire, l'étape D.

En effet, il faut :

- Introduire un nouvel élément de syntaxe et intégrer la définition des classes définies dans les fichiers importés dans l'arbre généré (étape A)
- Traiter cette nouvelle branche de l'arbre afin de connaître les classes importées (étape B)
- Ajuster la génération de la table des méthodes (étape C).

Ceci permettra au compilateur `decac` de générer des fichiers objets deca, d'extension `.deco`.

L'étape D correspond à un éditeur de liens, capable de récupérer plusieurs fichiers `.deco` et d'en générer un code assembleur exécutable par IMA.

Étape A

Modifications sur la syntaxe concrète

Pour adapter la syntaxe concrète à l'extension, nous avons ajouté des instructions `import <fichier_deca>` au début du fichier. Ainsi, on obtient la grammaire voulue en reprenant la grammaire fournie, et en ajoutant et remplaçant les règles des non-terminaux suivant :

prog -> **list_imports list_classes main** EOF

list_imports -> (**import**)*
import -> 'import' **fichier**
fichier -> STRING

On remarque que ces imports doivent impérativement se trouver en début de fichier, avant la liste des classes. On notera également que le chemin d'un import est calculé à partir du dossier du programme principal (à moins d'être absolu).

Modifications sur la syntaxe abstraite

La syntaxe abstraite est la structure de données fournie à l'étape B et l'étape C afin d'avoir les informations nécessaires pour les vérifications et la génération du code. L'information nécessaire pour modifier ces étapes pour l'extension est la déclaration des classes et méthodes des fichiers importés. On voit donc l'ajout d'une liste d'imports au terminal Program, d'un terminal DeclImport décrivant un fichier importé, et la suppression des corps des méthodes importées et du corps du main des fichiers importés.

On obtient la grammaire voulue en reprenant la grammaire abstraite fournie, et en ajoutant et remplaçant les règles des non-terminaux suivant :

PROGRAM
-> Program[**LIST_DECL_IMPORT**, **LIST_DECL_CLASS, MAIN**]

LIST_DECL_IMPORT
-> [**DECL_IMPORT** *]

DECL_IMPORT
-> **PROGRAM_IMPORT**

PROGRAM_IMPORT
-> DeclImport [
String↑*Address*
Program[**LIST_DECL_IMPORT**, **LIST_DECL_CLASS_IMPORT**, EmptyMain]
]

LIST_DECL_CLASS_IMPORT
-> [**DECL_CLASS***]

DECL_METHOD_IMPORT

-> DeclMethod[**IDENTIFIER**, **IDENTIFIER**, **LIST_DECL_PARAM**, EmptyMethodBody]

On peut remarquer que cette syntaxe contient plus d'informations que la syntaxe concrète. Le passage de la syntaxe concrète à la syntaxe abstraite n'est donc pas trivial. On remarquera également que nous avons conservé autant que possible les terminaux déjà existants, afin de complexifier aussi peu que possible l'arbre généré, et de factoriser le code des étapes B et C.

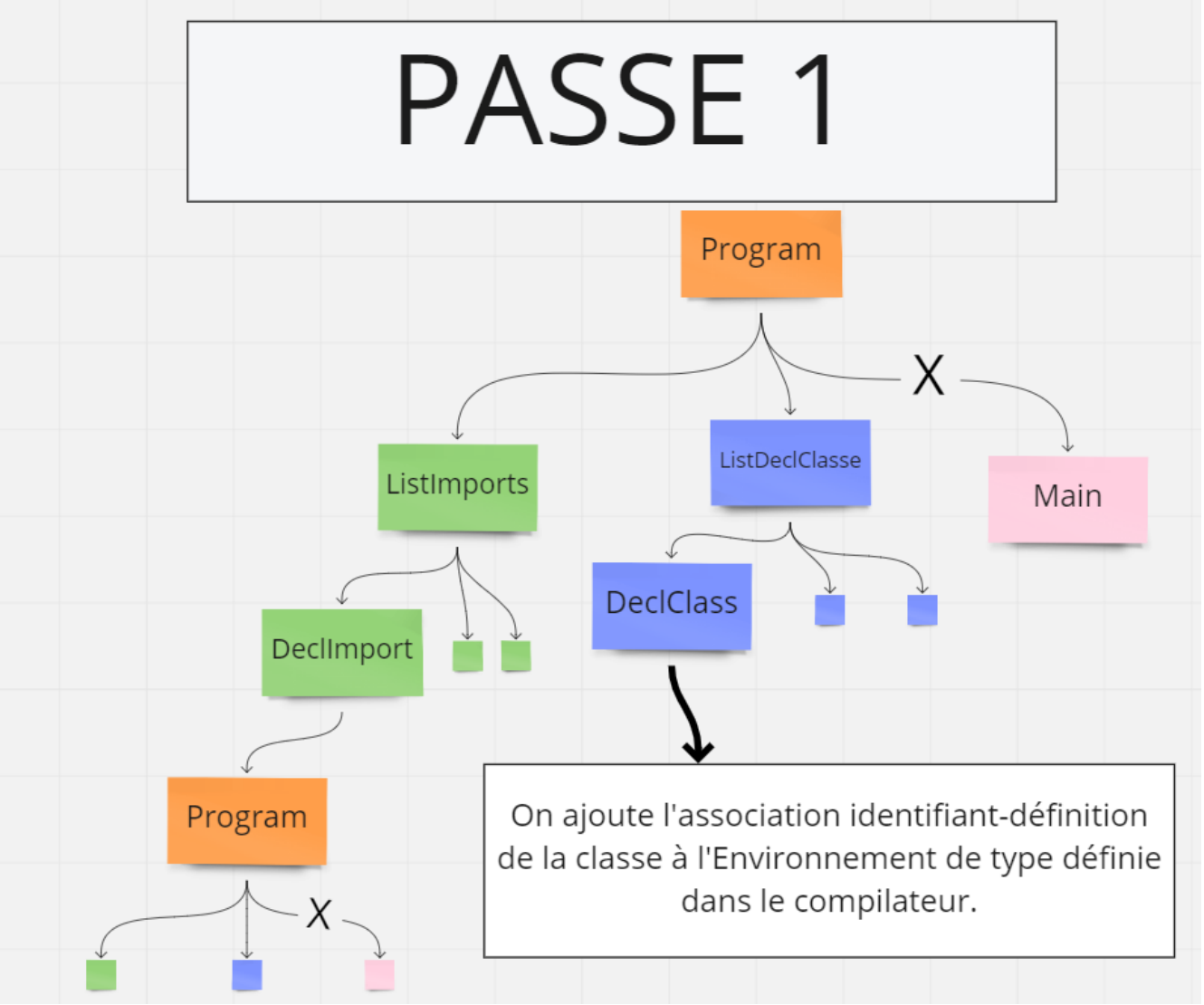
Passage de la syntaxe concrète à la syntaxe abstraite

Pour passer de la syntaxe concrète, qui décrit chaque fichier individuellement, à la syntaxe abstraite, qui résume les données du code source principal ainsi que celui du code source importé, un regroupement des information doit avoir lieu. Cela se passera lors du parsing des tokens de la syntaxe concrète. Lorsque le Parseur lira le token 'import', il lancera le lexer et un parseur capable de créer le sous-arbre spécifique pour les imports sur le fichier importé. Une fois ceci fini, il récupérera donc un arbre dont la racine est le terminal Program, et en crée un DeclImport, en y ajoutant un chemin du fichier.

Étape B

Problématique

Initialement, l'étape B se déroule en 3 passes.
La passe 1 sert à définir un environnement de types qui pourra être ensuite utilisé dans les deux passes suivantes et elle aurait dû se dérouler initialement de cette manière :



Ainsi, à la fin de cette passe, nous aurions disposé d'un environnement de type complet.

Cependant, et c'est le point crucial qui nous a fait changer notre implémentation, dans un cas comme celui-ci nous parcourons les classes dans l'ordre dans lequel elles ont été définies dans le code, et cela présente des inconvénients majeurs.

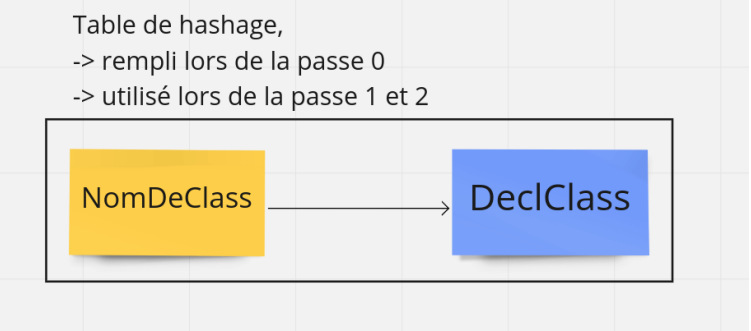
En effet, nous avons besoin que la classe mère d'une classe soit proprement définie afin de définir sa fille. Autrement dit, nous ne pouvions écrire nos classes que dans l'ordre de hiérarchie de classe, et devions toujours définir la classe mère avant la classe fille.

Cela présente de nombreux désavantages lors de l'édition de lien, car cela nécessitait que l'on ne puisse importer que des classes mère, et qu'en plus l'ordre d'import soit respecté pour que les Classes soient parcouru dans le bon sens.
Cela n'est bien sûr pas envisageable, nous avons donc choisi d'ajouter une Passe 0 afin de permettre le parcours des classes dans un ordre différent, un ordre qui nous permettrait de faire fit de l'ordre de déclaration des classes dans les fichiers mais de pouvoir toujours définir entièrement nos classes.

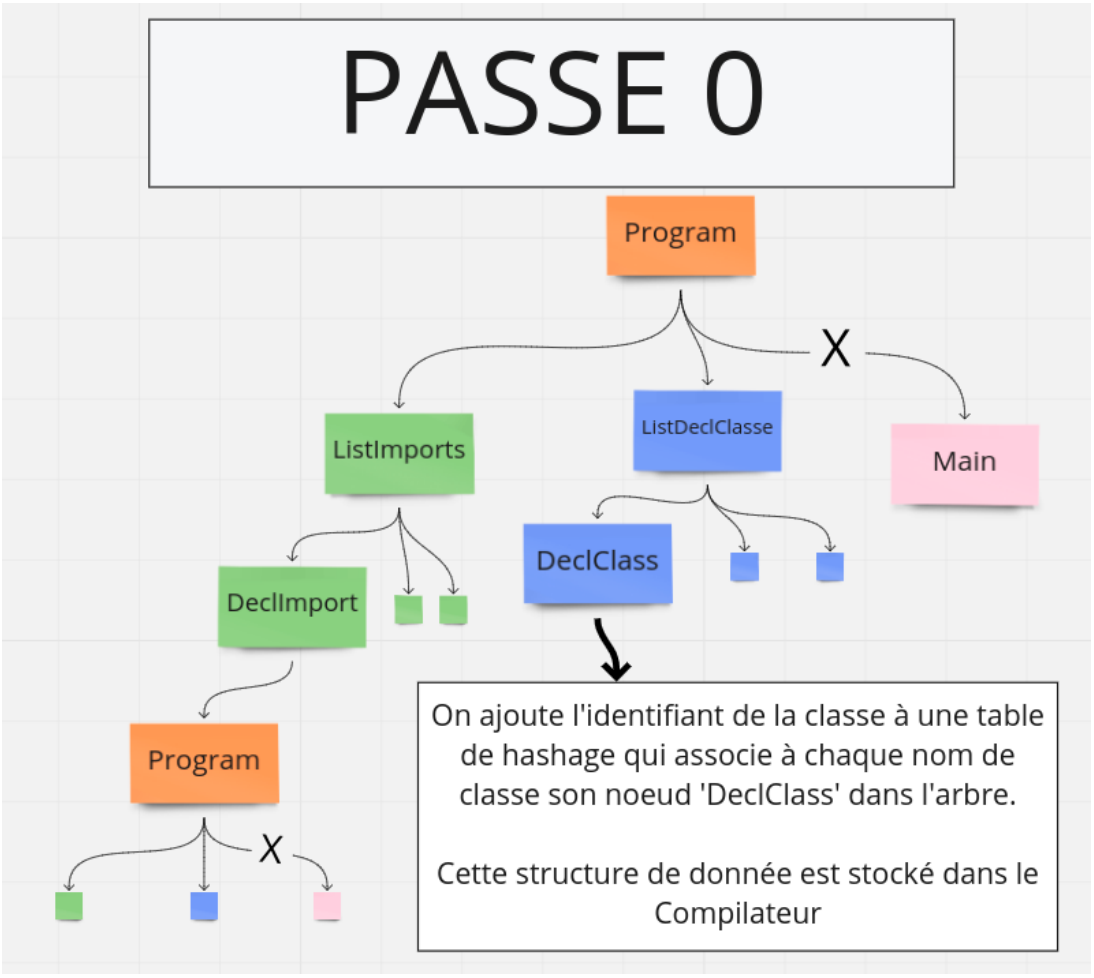
Passe 0

Afin de permettre l'édition de lien lors de cette étape, à partir de l'arbre généré lors de l'étape A, on réalise une passe supplémentaire sur l'arbre.

Le but de cette passe consiste à établir une table de correspondance entre les symboles et leur nœud dans l'arbre.
Cette table de hachage est stockée dans le compilateur, ce qui permet d'y avoir accès tout au long du parcours de l'arbre.



Cela permet ensuite de réaliser le parcours de classe non plus dans le sens de l'arbre et donc d'écriture des classes dans le code, mais dans le sens d'héritage à l'aide d'un appel récursif.

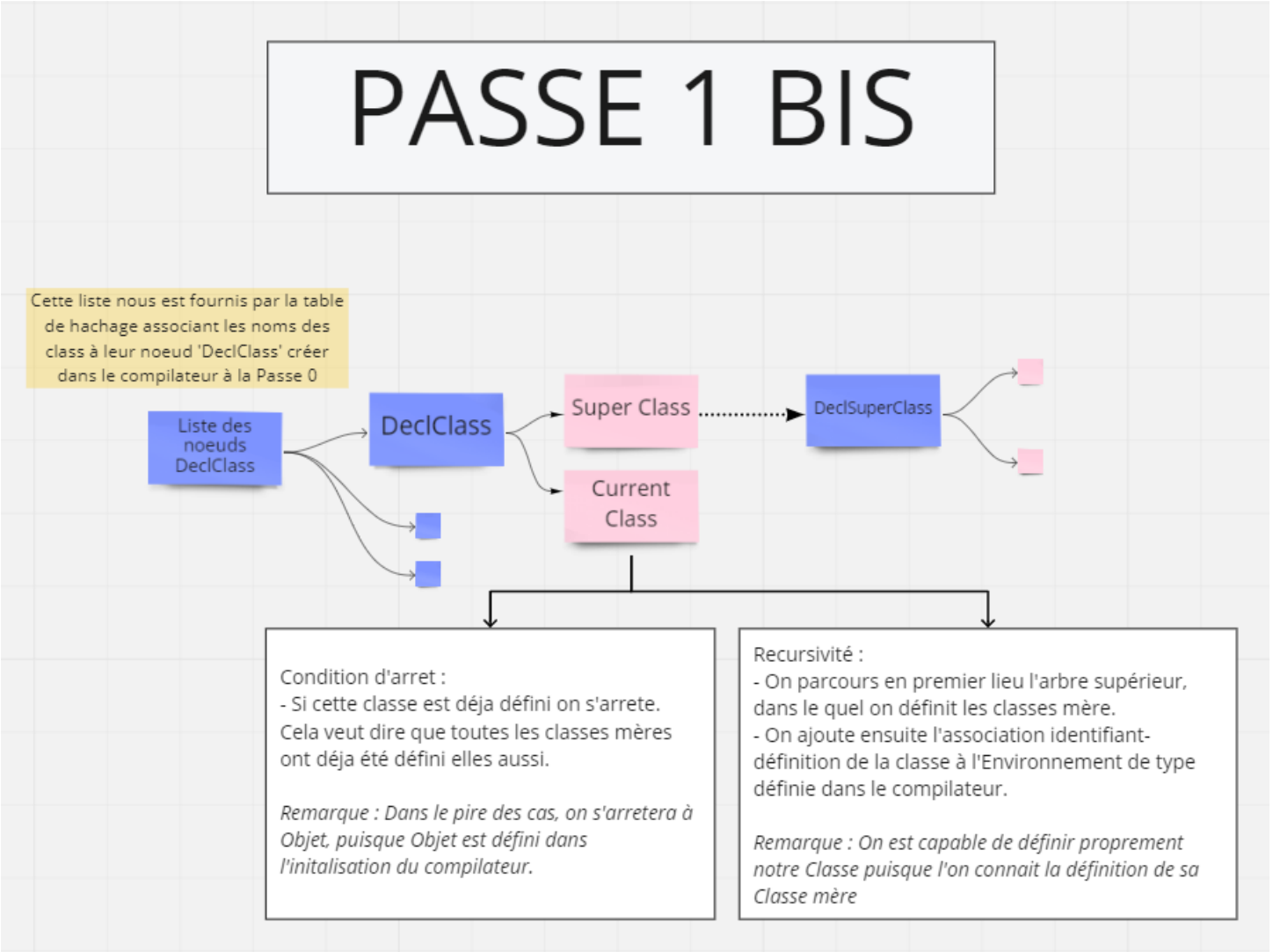


À la fin de la passe 0, on a donc une table de hachage contenant tous les noms de classes que nous souhaitons définir lors de la passe 1.

Appel Récursif

Lors de la Passe 1 et de la Passe 2, nous souhaitons que les classes parentes de la classe que nous cherchons à définir soient déjà définies, il nous faut donc parcourir les classes dans l'ordre d'héritage et non plus dans l'ordre de définition.
Cela est maintenant possible grâce à la nouvelle table de hachage dont nous disposons.
L'étape B gagne alors à partir de ce moment nettement en indépendance par rapport à l'étape A pour les passes 1 et 2 dans le sens où elle ne dépend plus que partiellement de l'arbre généré en étape A.
En effet, on ne travaille maintenant plus que sur les nœuds DeclClass et on se passe des nœuds Program, Main, ListDeclClass, ListImports et Imports.

On peut maintenant, grâce à un appel récursif, parcourir les classes non plus dans le sens d'appel de l'arbre, mais dans le sens d'héritage des différentes classes.



La passe 2 suit le même schéma, mais lors de la récursion, ce sont les méthodes de la classe et ses champs que l'on ajoutera dans l'environnement d'expression de la Classe courante.
Il reste nécessaire de parcourir les classes parentes car l'on en a besoin pour pouvoir définir correctement les méthodes et les champs. En effet, on doit leur attribuer un index qui dépend du nombre de méthodes et de champs du présent dans la Classe parente.

Étape C

Afin de permettre des appels sur les classes importées, on ajoute les tables des méthodes des classes importées dans la table courante. Comme dans la liste d'imports, nous avons également les imports des imports eux-mêmes, la table générée contiendra toutes les classes dont on peut avoir besoin dans l'exécutable final.
De plus, il faut ajouter à l'assembleur généré les éléments afin que l'étape D puisse se passer correctement. Il faut donc être conforme au nommage des labels indiqué ci-dessous et ajouter des commentaires indiquant les limites des sections du fichier.

Étape D

L'étape D consiste à concaténer les différents fichiers *.deco*.
Pour ce faire, le *linker* va modifier les fichiers pour assurer une bonne cohésion lors de la concaténation.

Nommage des labels

Pour cela, il va s'assurer de l'unicité des labels. En effet, la numérotation des labels se fait de façon indépendante au fichier, il est donc possible qu'il y ait de la redondance entre les différents *.deco*. Par exemple, chaque fichier ayant une boucle while a au moins un label de la forme "while..0", il peut donc y en avoir autant que de fichiers.
Ainsi, le *linker* va placer devant les labels assembleur le numéro du fichier. On aura donc le label "*<nomLabel>*" qui deviendra "*file.<numéro_de_fichier>.<nomLabel>*". Cela assure l'unicité de chaque label.
De plus, pour être certain de remplacer seulement les labels que l'on souhaite modifier, nous avons choisi comme convention de mettre par défaut les caractères ".." dans chaque label qui devra être renommé. Cela assure l'unicité du nom à rechercher. Un seul point ne suffit pas, car le point est utilisé comme séparateur par défaut, et en particulier dans le nom des méthodes (sous la forme de "*methodBody.<nomDeClasse>.<nomDeMethode>*"), dont le nommage dépend du fichier source. Par exemple, une classe nommée "overflow" avec une méthode nommée "error" pourrait voir son label modifié pendant la modification des labels de la forme "overflow.error", label qui pourrait être utilisé pour la gestion d'erreurs et qui était modifié dans une première version du linker.
Le point étant interdit dans le nom des méthodes et des classes, on ne peut pas y avoir un deuxième point d'affilée dans le nom des labels à modifier, il n'y a alors aucun risque modification de labels dont le nommage dépend du fichier compilé.
Cela nous amène à un second point: le nommage des labels relatifs aux classes. Comme nous l'avons sous-entendu, les labels relatifs aux classes ne sont pas modifiés pendant l'édition de liens. Étant donné que le corps des méthodes n'est défini que dans un seul fichier objet (car c'est une information qui est supprimée lors d'un l'import pendant la phase

de la compilation), il n'y a que ce fichier objet qui définit le label marquant le début du corps de la méthode. Il ne faut donc pas modifier ces labels.

De plus, il est également possible de n'utiliser que les messages d'erreurs du programme principal et de supprimer celui des
, le linker se chargera ainsi de ne garder que les messages du fichier principal, qui, lorsque l'option de *decac* "-c" est activée, contiendra exceptionnellement tous les messages d'erreurs possibles, et plus seulement ceux qui sont appelés dans le programme.

Gestion des sections

La deuxième partie de linker consiste à gérer les sections qui seront présentes dans l'exécutable final. Suite à la manière dont le code est généré, nous avons défini 4 sections: la construction des tables des méthodes, le programme principal (main), le corps des méthodes et les messages d'erreurs. Chacune d'elles doit être présente dans chaque fichier objet, afin de pouvoir prendre chacun de ces fichiers comme programme principal s'il le faut.
Cependant, lors de l'édition de liens, seule la section du corps des méthodes est intéressante à l'import. Les sections de construction des tables des méthodes et de messages d'erreurs sont en double, et le programme principal est inutile. Il faut donc récupérer le premier fichier objet spécifié en entier, et y ajouter la section du corps des méthodes des autres fichiers objets, après avoir modifié les labels qui peuvent être les mêmes d'un fichier à l'autre.

Analyse bibliographique

L'extension est fortement inspirée de la compilation du langage C, mais en reprenant l'absence de fichier d'entête comme en Java ou Python. Ainsi, nous avons surtout voulu reproduire leurs fonctionnalités. Nous nous sommes également documentés, principalement sur Wikipédia, sur le format interne d'un fichier objet :
https://fr.wikipedia.org/wiki/Code_objet
https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
https://en.wikipedia.org/wiki/Object_file

Choix de conception, d'architecture et d'algorithmes

Le premier choix à faire était sur le format du fichier objet. Tout comme les fichiers objets C, nous avons repris le langage machine, en y intégrant les informations nécessaires pour la partie édition de liens.
Étant donné la simplicité du format des fichiers exécutables par ima (langage assembleur et non pas binaire, adresses via des labels), nous n'avons que besoin de supprimer les parties

dupliquées de code (construction de la table des classes, gestion des erreurs, programme principal), en reprenant l'idée de sections, et d'assurer l'unicité des labels pouvant se trouver en double entre les fichiers compilés.

Le deuxième choix était de faire en sorte que la mécanique d'importation ne concerne que les classes des autres fichiers deca indiqués, et pas le programme principal. Les classes sont directement accessibles tant qu'elles sont importées et les imports des fichiers importés sont également disponibles.

Le choix suivant était sur le moment de la vérification vis-à-vis des classes importées. Nous avons choisi de faire toutes les vérifications de contexte lors de la compilation, nécessitant donc l'import de la déclaration des classes lors de la compilation.
Il a donc fallu modifier la syntaxe abstraite pour y intégrer les classes importées et leur déclaration. Il a aussi fallu prévoir un moyen dans l'étape A de récupérer les définitions de classes, afin de remplir l'arbre.

Un autre choix majeur était de réutiliser les classes dans l'arbre généré par le parseur, permettant de limiter fortement la charge de travail ajoutée par l'extension sur les étapes B et C.

Méthodes de validation

Pour valider l'extension, chaque étape peut être testée spécifiquement dans un premier temps. Nous avons donc une méthode différente par étape:

- Pour l'étape A, la vérification que le lexème est bien pris en compte se fait simplement avec le script test_lex fourni, en ayant le mot 'import' dans le fichier testé et en s'assurant que le lexème correspondant est bien généré, en plus d'un tests vérifiant si dans une chaîne de caractères elle est bien ignorée.
  Ensuite, nous pouvons utiliser test_synt pour vérifier que l'arbre abstrait est bien complété par les déclarations du fichier importé.
- Pour l'étape B, la validation est faite
- Pour l'étape C, nous vérifions tout à la main que le *.deco* généré correspond à la spécification (nommage des labels et commentaires). Toute erreur supplémentaire ressortira lors de la validation de l'étape D.
- Pour l'étape D, nous vérifions à la main que, lorsque deux fichiers objets venant de l'étape D sont liés entre eux, les bonnes parties de chaque fichier sont conservées, et que les labels sont bien renommés.

Une fois ces vérifications terminées, il reste à tester la compilation entière. Pour cela, nous testons d'une manière similaire au compilateur complet: on crée un script capable d'itérer sur les tests, de les compiler un à un, de faire l'édition de liens, et d'exécuter le résultat.
Nous avons créé des tests simples, qui important simplement un autre fichier, puis des tests qui utilisent les classes des imports pour vérifier que la position de la table des classe est correcte.

Résultats de la validation

Nos tests passent tous, et l'expérience en ligne de commandes correspond bien à ce que l'on a spécifié: une compilation en deux étapes, la première avec *decac -c* et la deuxième avec *linker*.