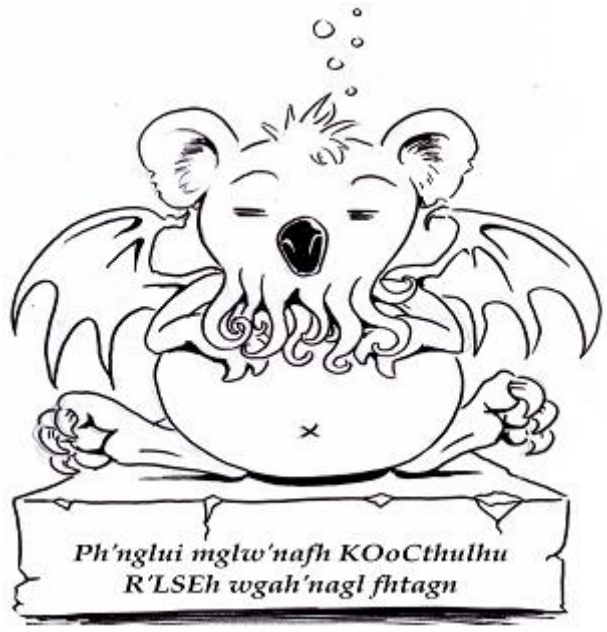
Equipe : lair_a – mengon_a – barry_g – sonier_b
Chef de projet : sonier_b

KOOC

Kind Of Objective-C

Document de Conception



Logiciel

Référence	kooc
Version	1.10
Date de version	21 / 11 / 2010
Etat	En développement

Développeurs

M. SONIER Alexandre	sonier_b	alexandre1.sonier@epitech.eu	**Chef de projet**
M. LAIR Yann	lair_a	yann1.lair@epitech.eu	
M. MENGONI Jérémy	mengon_a	jeremy1.mengoni@epitech.eu	
M. BARRY Grégory	barry_g	gregory.barry@epitech.eu	

Historique

Version	Date	Motif
1.10	21 / 11 / 2010	Object
1.9	21 / 11 / 2010	Heritage simple (headers)
1.8	20 / 11 / 2010	@virtual / Début vtable
1.7	20 / 11 / 2010	[]; pour fonctions
1.6	18 / 11 / 2010	@class / @member, sans implémentation
1.5	09 / 11 / 2010	Gestion du multi-fichiers
1.4	03 / 11 / 2010	[]; pour variables
1.3	20 / 10 / 2010	@implementation (sans [];)
1.2	19 / 10 / 2010	Base du mangling implémentée
1.1	18 / 10 / 2010	@module fonctionnel
1.0	14 / 10 / 2010	@import fonctionnel

SOMMAIRE

I. PREAMBULE

Le KOOC, pour « *Kind of Objective-C* », est un projet à réaliser, par équipe de 4 étudiants, au début de la troisième année à EPITECH.

Son but : implémenter un nouveau langage sur la base de la grammaire du C en y ajoutant le concept de programmation orientée objet, ainsi que certaines améliorations comme l'inclusion de fichiers auto-protégée contre la multiple inclusion.

Les outils : pour réaliser ce projet, des outils sont mis à notre disposition :

- La « cnorm », qui est une grammaire C implémentée en CodeWorker permettant, à partir d'un code C, de générer un AST (« *Abstract Syntax Tree* » : arbre syntaxique abstrait).
- La « patchLib », qui est une librairie regroupant de nombreuses fonctions nous permettant de modifier à notre convenance l'AST généré via la « cnorm ».
- Le « cnorm2c », qui nous permettra de régénérer un code C syntaxiquement correct via un AST.

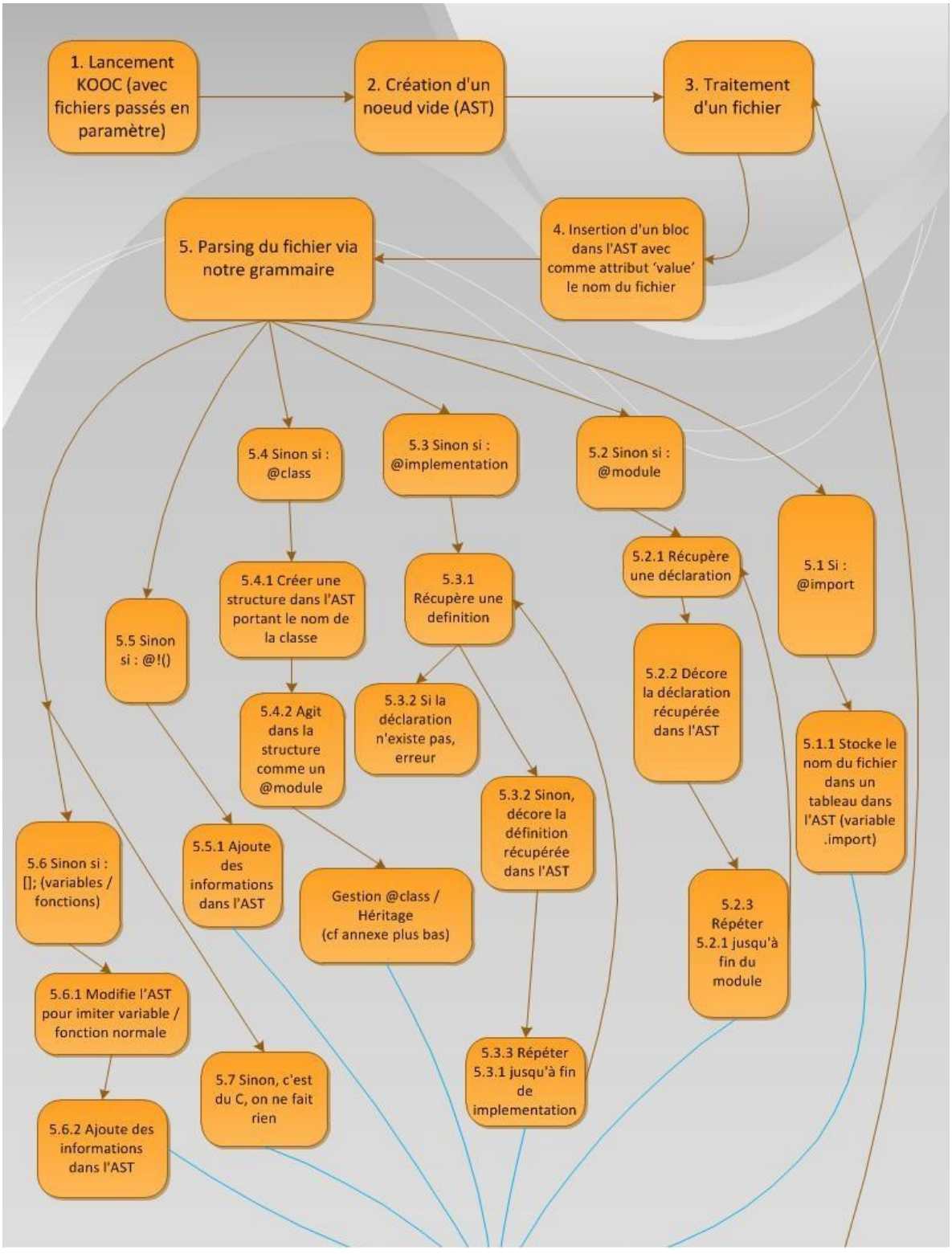
Comment y parvenir : l'architecture du programme KOOC se construira de la manière suivante :

- Appel à l'exécutable 'codeworker' qui utilisera la grammaire « cnorm » pour parser les fichiers *.kc *.kh passés en paramètre.
- Surcharge de la grammaire de la « cnorm », à l'aide d'une grammaire implémentée par nos soins afin de gérer les différentes fonctionnalités apportées par le KOOC.
- Modification, via la « patchLib », de l'AST généré précédemment afin d'obtenir la structure voulue.
- Régénération du code KOOC parsé dans les fichiers *.c *.h correspondants.

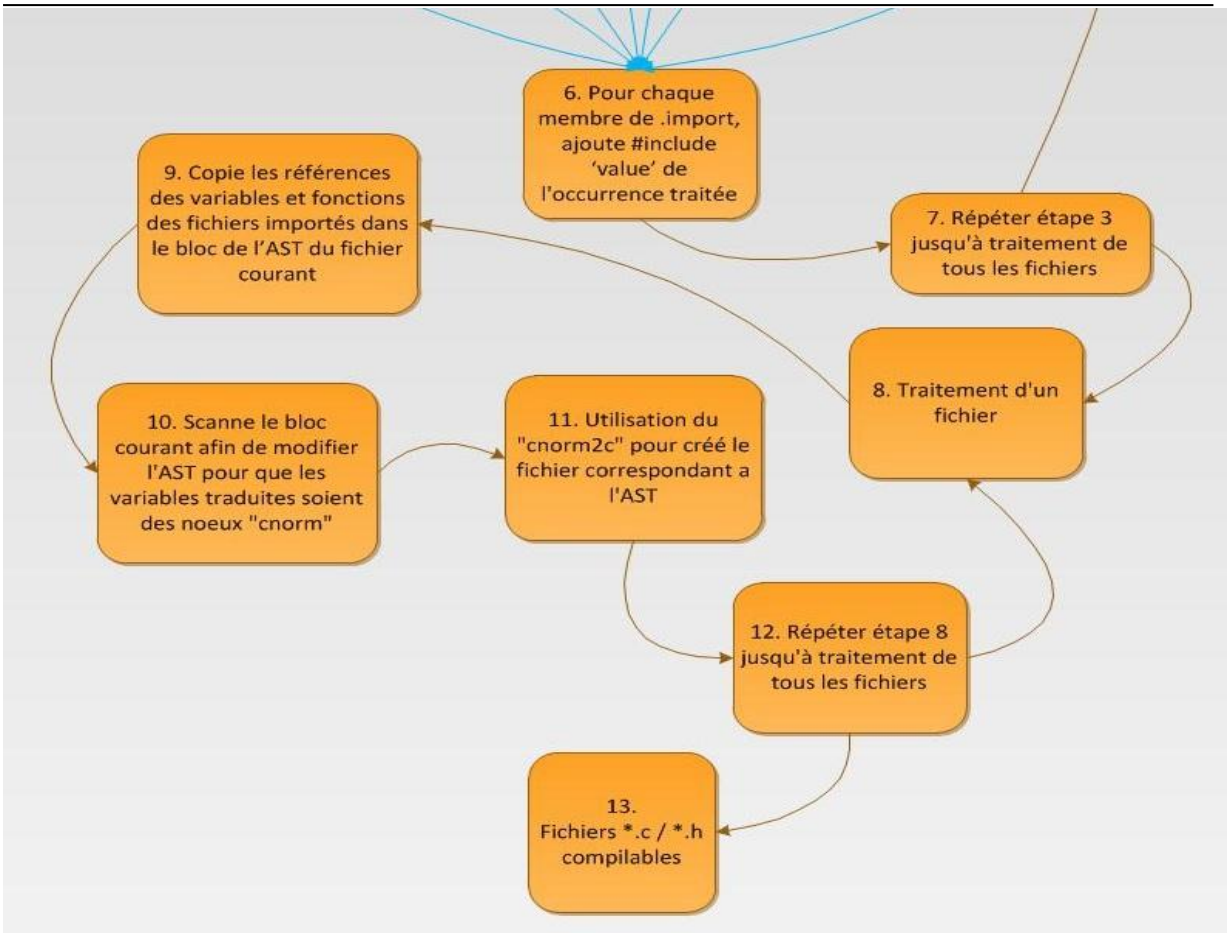
*NB : Dans le cas d'un passage de plusieurs fichiers, nous traitons en premier les fichiers *.kh, puis les *.kc. Ceci nous permettra de connaitre au préalable toutes les fonctions contenues dans des* `@module` *qui seront par la suite implémentées dans les* `@implementation`*.*

Schéma explicatif du déroulement du programme 'KOOC'

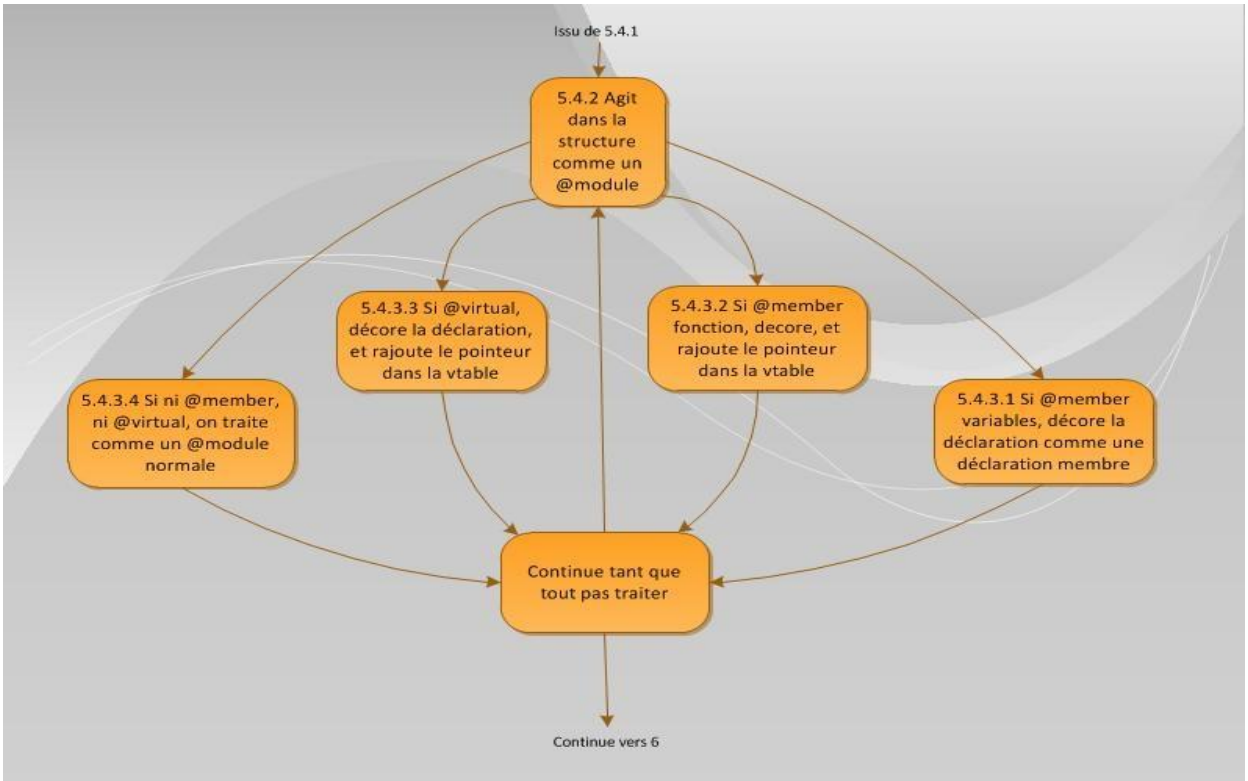
Diagramme fonctionnel de notre programme KOOC :



Première partie. Suite à la page suivante.



Suite du diagramme fonctionnel



Annexe héritage

II. MANGLING

1. Utilité du Mangling

Le mangling, ou décoration de symbole, répond au problème de la surcharge de fonctions. En effet, KOOC se voulant être un langage orienté objet, celui-ci doit permettre la surcharge de fonctions, impliquant ainsi que certaines d'entre elles aient le même nom, ce qui n'est pas syntaxiquement correct avec la grammaire C que nous utilisons. Pour pallier ce problème, il nous faut « décorer » les fonctions déclarées dans le code KOOC.
Cela nous permet, en outre, de gérer les problèmes de résolution de noms entre variables du même nom, déclarées dans des `@module` (partie déclarative du code KOOC) différents.
Pour cette décoration, la gestion des types des variables est la suivante :

Types gérés :
- Char
- Short
- Int
- Long
- Long long
- Float
- Double
- Long double

Types non gérés :
- Struct
- Union
- Enum
- Static
- Typedef
- Const
- Volatile

Dans la décoration, les types apparaissent selon leur nom. Il faut néanmoins distinguer deux cas particuliers : les pointeurs et les arrays. Ceux-ci apparaissent de la manière suivante au sein de notre mangling :

- Array : `array` suivi du nombre de cases de l'array.
  Exemple :

```
int tab[4]; // De type 'signed_int_array4' dans le mangling
```

- Pointeurs : `point`, répété en fonction du niveau du pointeur.
  Exemple :

```
int **tab; // De type 'signed_int_point_point' dans le mangling
```

2. Notre solution

Nous avons donc vu qu'il nous fallait décorer toute variable ou fonction que nous rencontrons dans la déclaration d'un `@module`. Cette décoration s'effectue selon le modèle suivant :

[M|P|V]_lMnomInst_typeSymb_typeRet_lDnomDecl[_param_typeParam]

Facultatif : M variable/fonction membre, P pointeur fonction membre, V fonction virtuelle.
lM : lenghtModule, correspond à la taille du nom du module.
nomInst : le nom de l'instance (module ou classe) dans lequelle la variable/fonction est déclarée.
typeSymb : type du symbole. Peut avoir les valeurs suivantes : `variable`, `prototype`.
typeRet : s'il s'agit d'une variable, c'est le type de celle-ci. Si au contraire, nous avons affaire à une fonction, cela sera le type de retour de cette dernière. Si jamais ce type est composé, par exemple `signed int`, les espaces sont remplacés par des '_' dans la décoration.
lD : lenghtDeclaration, correspond à la taille du nom de la variable/fonction.
nomDecl : il s'agit simplement du nom de la variable/fonction.
typeParam : facultatif. Présent dans le cas où il s'agit d'une fonction. Rassemble les types de tous les paramètres de la fonction selon le même modèle que **typeRet**.

Plusieurs exemples afin d'illustrer notre solution de mangling :

```
@module         bobo
{
  int           a;
  char          a;
  long          a;
  int           tab[2][3][4];
  void          blabla(void);
  int           blabla();
  int           blabla(int, char);
  int* const*   ahah(char*, const char*);
}
```

Les variables et fonctions du module `bobo` seront décorées respectivement de la manière suivante :

```
signed int          _4bobo_variable_signed_int_1a;
signed char         _4bobo_variable_signed_char_1a;
signed long int     _4bobo_variable_signed_long_1a;
signed int          _4bobo_variable_signed_int_array2_array3
                    _array4_3tab [2][3][4];

void                _4bobo_function_void_6blabla_param_void(void);
signed int          _4bobo_function_signed_int_6blabla_param_void();
signed int          _4bobo_function_signed_int_6blabla_param_
                    signed_int_signed_char(signed int, signed char);
signed int* const*  _4bobo_function_signed_int_point_const_point_
                    4ahah_param_signed_char_point_
                    const_signed_char_point(signed char*,
                    const char*);
```

III. MOTS CLES PROPOSES

1. @import

a. *Description*

`@import` est un mot-clé KOOC qui a pour ambition de remplacer la directive préprocesseur `#include` pour l'implémenter de manière plus intelligente. En effet, en C, si un fichier à inclure venait à ne pas être protégé contre la double inclusion, cela pourrait provoquer une inclusion en boucle du même fichier, pour finalement aboutir sur une erreur empêchant la compilation de votre programme :

```
#include nested too deeply
```

Cela peut aussi induire des multiples définitions de mêmes prototypes de fonctions.
Ainsi, `@import` se veut plus intelligent que `#include` et il gère donc directement la protection contre la double inclusion.

Exemple d'utilisation de `@import` :

```
@import "example.kh"
@import <iostream>
```

b. *Solution envisagée, finalement non retenue*

Dans un premier temps, nous avons tenté de parser directement, à l'aide de notre grammaire, le `@import` dans le code KOOC de sorte que nous puissions le remplacer directement dans le code C régénéré de la façon suivante :
Code KOOC :

```
@import "example.kh"
```

Code C régénéré :

```
#ifndef EXAMPLE_KH
#define EXAMPLE_KH
#include "example.kh"
#endif
```

Cependant, cette solution s'est avérée impossible à mettre en place. En effet, dans notre tentative, nous nous sommes rendu compte que la « cnorm » mise à notre disposition ne gérait pas les directives préprocesseur. Nous avons donc abandonné cette solution.

c. *Solution retenue*

On parse les fichiers `*.kc` afin de retrouver toutes les occurrences de `@import` de sorte que nous puissions les stocker dans un arbre. Seulement, à chaque insertion, on vérifie au préalable si le nœud se rapportant au fichier à ajouter n'existe pas déjà au sein de l'arbre, ce qui nous permet de gérer, de façon intelligente, le problème de la multiple inclusion. Au final, on possède un arbre qui contient une seule occurrence des noms des fichiers à inclure. Il nous suffit alors simplement d'utiliser la directive préprocesseur `#include` dans les fichiers de sortie `*.c` pour les inclure.

2. @module / @implementation

a. *Description*

`@module` équivaut à la partie déclarative du code KOOC. Il permet ainsi la déclaration des variables et des prototypes de fonctions qui seront implémentées par la suite dans le bloc `@implementation`. Point important, les variables initialisées dans le `@module` le sont en réalité au début de `@implementation`. De plus, nous avons fait **le choix d'interdire l'implémentation de fonctions étrangères dans un `@implementation`,** étrangères dans le sens où elles n'auraient pas été préalablement déclarées dans le `@module` correspondant, cela par souci de clarté du code produit. En conséquence, comme en C++ où l'on doit mettre le nom de la classe devant le nom des fonctions (`nomClasse::nomFonction`), on ne peut donc implémenter que les fonctions déclarées dans le `@module` du même nom.

```
@module Example
{
  // Déclaration de variables globales
  int     a = 42;
  double  a = 4.2;
  char*   a;

  // Déclaration des prototypes de fonctions qui seront
  // implémentées dans « @implementation Example »
  int     function(void);
  int     function(int);
  double  function(void);
}

@implementation Example
{
  // A ce moment, les trois variables précédemment
  // déclarées dans « @module Example » doivent être initialisées
  int     function(void)
```

```
  {
    return (42);
  }

  int     function(int var)
  {
    return (var * [Example function]);
  }

  double  function(void)
  {
    return (4.2);
  }
}
```

b. *Solution envisagée pour `@module`, finalement non retenue*

Concernant la gestion du `@module`, nous avons d'abord envisagé d'effectuer une analyse, nœud après nœud, de l'AST produit par l'utilisation de la « cnorm ».
Cependant, nous avons vite écarté cette solution car celle-ci nécessitait d'inhiber le code KOOC afin d'éviter les messages d'erreur qu'engendrait la « cnorm » pour ensuite modifier, « en brut », l'AST. Nous avons estimé cette solution peu optimisée et difficile à mettre en place, ce pourquoi nous ne l'avons pas retenue.

c. *Solution retenue pour `@module`*

Il nous a paru plus judicieux de surcharger, grâce à la directive `#overload`, `translation_unit` afin d'être en mesure de parser directement le code KOOC. Au final, nous reprenons la déclaration préalablement effectuée dans la « cnorm » pour y ajouter un appel à notre fonction de génération de noms de déclaration : le mangling.
Cela nous permet donc de décorer seulement le code KOOC, sans pour autant porter atteinte à l'intégrité du code C présent dans le programme.

d. *Solution envisagée pour `@implementation`, finalement non retenue*

La première solution que nous avons envisagée pour concevoir le `@implementation` fut de créer une map contenant les noms des fonctions rencontrées ainsi que, par correspondance, le mangling de ces fonctions. De cette manière, il nous suffisait de faire des comparaisons une fois dans le `@implementation` afin de remplacer en « matchant » le nom des fonctions rencontrées. Toutefois, nous avons décidé de ne pas retenir cette solution du fait que cela aurait été lourd à mettre en place (comparaison très précise des noms, des types des paramètres, etc...), et peu optimisé car nous aurions dû créer une map à l'intérieur de l'AST. Jugeant cette solution inadéquate, nous nous sommes donc orienter vers une nouvelle solution.

e. *Solution retenue pour* `@implementation`

La deuxième solution qui s'est présentée à nous est celle que nous avons retenue. Elle consiste à « mangler » les fonctions que l'on rencontre au sein du `@implementation` et, via une comparaison avec l'AST, on vérifie que cette fonction existe, ou bien, dans le cas contraire, nous générons une erreur. Cette solution s'est avérée beaucoup plus simple et plus facilement applicable, dans le sens où nous pouvons facilement réutiliser du code existant ce qui nous permet de gérer tous les cas de figures sans souci en l'adaptant rapidement.

f. *Traduction des appels KOOC : variables*

Afin de faire appel aux variables préalablement déclarées dans un `@module`, KOOC utilise une syntaxe qui lui est propre.
Exemple d'un `@module` avec une variable quelconque :

```
@module   example
{
  int     a;
}
```

Pour faire appel à la variable `a` du module `example`, la syntaxe utilisée est :

```
[example.a]
```

On distinguera dans la suite de cette partie les expressions simples des expressions complexes. On s'aidera de l'exemple ci-dessus pour expliquer notre implémentation de cette problématique.

- `[example.a] = 'expression_simple'`

Lorsque l'on rencontre `[example.a]` dans le code, on effectue, au sein du nœud de l'AST se rapportant à cette variable, les modifications suivantes :

- On crée une variable `conflict` initialisée à `module`.
- On crée une variable `module` contenant le nom du module (dans notre exemple, il s'agirait de `example`).
- On remplit la variable `value` du nœud avec le nom non décoré de notre variable (dans notre exemple, il s'agirait de `a`).

Le reste des variables du nœud est directement géré par la « cnorm ».
Une fois le parsing de toutes les variables terminé, on utilise une fonction récursive pour scanner l'AST et ainsi mangler les noms des variables en les remplaçant dans la variable `value` correspondante.

Pourquoi ?

A la base, nous avions envisagé de chercher l'existence de la variable puis de mangler le nom directement. Cependant, nous nous sommes confrontés à un problème : celui de l'autocast. (comprendre l'appel KOOC `@!(type)`). En effet, lorsque la variable du module se trouve à gauche de l'expression, le membre de droite n'est pas encore parsé. Ainsi, lorsque l'on traite l'expression, nous ne pouvons pas récupérer le type de celui-ci pour résoudre les problèmes d'ambiguïté dans le cas où plusieurs variables d'un même module auraient le même nom mais des types différents. C'est pourquoi nous avons choisi la solution de scanner l'AST une fois le parsing entièrement fini.

- `[example.a] = 'expression_complexe'`

Dans le cas d'expressions plus complexes, la solution que nous avons retenu est d'améliorer notre fonction de scan récursif de l'AST pour que le mangling affecte dans un premier temps les membres de droite de l'expression, suivis des membres de gauche, et ce en allant le plus loin possible dans l'AST via la récursivité. Cheminement dans chaque nœud concerné :

- Aller dans le `block .right` le plus bas possible.
- Effectuer notre mangling
- Créer une variable dans le membre contenant le `.right` tout juste manglé, celle-ci contenant le `.ctype` du `.right`.
- Ainsi, on peut mangler le `.right` précédent.

Exemple :

```
.block
{
.block
     {
     .left
     .right
          {
          .left
          .right
               {
               .left
               .right
                    {
                    .ctype = int
                    }
               .ctype = .right.ctype = int // Notre parse
               }
          .ctype = .right.ctype = int // Notre parse
          }
     }
}
```

De cette façon, on connait le type de chaque `.right` et celui-ci est déterminé en prenant celui de l'expression qu'il contient. Le parsing s'effectue donc sans encombre.

g. Traduction des appels KOOC : fonctions

Tout comme pour les variables, KOOC utilise une syntaxe particulière pour se référer aux fonctions préalablement déclarées dans un `@module`.
Exemple d'un `@module` avec un prototype de fonction quelconque :

```
@module   example
{
  int     add(int, int);
}
```

Pour faire appel à la fonction `add` du module `example`, la syntaxe utilisée est :

```
[example add :42 :42] // <=> add(42, 42)
```

Concernant la gestion de la traduction des appels KOOC relatifs aux fonctions, celle-ci est sensiblement identique au traitement des appels KOOCS pour les variables. (*cf. h. Traduction des appels KOOC : variables*). La différence notable sera que nous travaillerons sur un bloc de l'AST différent, propre à la fonction concernée et non plus sur un bloc de variable. Cela aura donc aussi une incidence sur la nature des modifications, mais conceptuellement parlant, la méthode reste la même : modification des blocs concernés dans l'AST et traitement final en scannant par la suite l'intégralité de l'arbre binaire syntaxique.

Exemple d'AST avec appel KOOC fonction

3. @class / @member

a. Description

Comme KOOC se veut être orienté objet, il nous faut les outils adéquats. Le mot clé `@class` est là pour répondre à ce besoin puisqu'il va nous permettre d'implémenter des classes qui seront par la suite instanciées. Le mot clé `@class` se veut être une amélioration du `@module`. Le mot clé `@member` va nous permettre de préciser quelles variables et fonctions feront partie intégrante de la classe. *A contrario*, toute variable ou fonction déclarée dans un bloc `@class` mais n'étant pas précédée de `@member` sera décorée comme s'elle était dans un bloc `@module`.
Du côté de la conception, on considèrera qu'une classe sera déclarée comme une structure. Cependant, il faudra quand même décorer les variables et fonctions afin de conserver la possibilité de les surcharger.
Ainsi, dans notre conception :

```
@class A { }
```

Est équivalent à :

```
typedef struct _class_A_ {} A;
```

b. *Allocateur, initialiseur et destructeur*

La fonction `alloc` alloue la mémoire nécessaire pour stocker toutes les informations de la classe correspondante (variables, ...). S'il y a besoin, on utilise la fonction `malloc` pour les pointeurs déclarés dans la classe. La fonction `alloc` est une fonction non-membre et ne sera pas accessible à l'utilisateur (puisqu'elle sera en réalité appelée seulement lors de l'appel d'un `new`). D'un point de vue conceptuel, on retourne une allocation mémoire de la taille de notre classe.

La fonction `delete` libère la mémoire qui a été allouée au préalable pour ensuite appeler la fonction `clean`, équivalente au destructeur si celui a été déclaré. Si la fonction `clean` n'a pas été déclarée, on la déclare nous-même mais celle-ci restera vide.
La fonction `new` permet d'appeler la fonction `alloc` avant la fonction `init` qui correspond au constructeur. Ainsi, on crée un cas particulier dans la traduction pour appeler `alloc`, puis `init` dans notre code.
Conceptuellement parlant, tout cela peut se traduire à travers l'exemple suivant :

```
@class          Example
{
  Example*      alloc()
  {
    return (malloc(sizeof(Example)));
  }

  @member void delete(Example* self)
  {
    free(self);
    clean(self);
  }

  Example*      new()
  {
    Example*    self = alloc();
    init(self);
  }

  Example*      new(int n)
  {
    Example*    self = alloc();
    init(self, n);
    return (self);
  }
```

c. *Quelques précisions supplémentaires*

Les variables et les fonctions membres d'une classe verront apparaître la lettre « M » au début de leur décoration. Les fonctions déclarées en tant que membre via `@member` se verront ajouter un nouvel argument, placé en première position, qui sera un pointeur vers l'instance courante de la classe, que l'on nommera '`self` '. Pour ce faire, on interceptera la déclaration et, grâce à la « patchLib », nous apporterons les modifications adéquates. De plus, ne pouvant pas implémenter de fonctions dans une structure (la syntaxe du C l'interdit), celles-ci le seront dans un `@implementation` et seront décorées de la même façon qu'une fonction ayant été déclarée dans un `@module`.

d. *[] ;*

La traduction (terme que l'on emploie pour désigner le passage d'une variable/fonction déclarée entre crochets à son état décoré) sera traitée ici comme dans `@module`, puisqu'il doit y avoir compatibilité. Pour implémenter la traduction des fonctions, on vérifie que l'on connait un `@module` ou un `@class` qui a le même nom que le premier terme dans `[];` et si c'est le cas, la traduction implémentée précédemment pour le `@module` s'occupe de tout. Sinon, on choisit d'envoyer l'objet en premier argument à la fonction. Concernant la traduction des variables, notre solution consiste à enlever les crochets et à décorer, par la suite, la variable (`nomInstance.variableDecoree`) puis on se contente d'exécuter la même démarche qu'à l'intérieur du `@module` pour récupérer la variable correspondante (gestion des types en cas de surcharge). Concernant la traduction au sein d'un `@implementation`, notre conception devrait faire en sorte que cela soit automatique grâce à ce que l'on a implémenté précédemment (via la surcharge de `primary_expression`).

4. Héritage / @virtual / Object

a. *Description*

L'héritage est une notion rajoutée dans la programmation orientée objet et spécifique à celle-ci. Elle permet de créer des relations père/fils entre deux classes, tout en sachant que la classe fils héritera des variables et fonctions de la classe père, et qu'elle pourra les utiliser ; et cela, à la chaine. Pour cette troisième étape, il nous est demandé d'implémenter, entre autre, l'héritage simple à notre `@class` (la cascade d'héritage est donc à gérer).
De plus, à l'intérieur de cette notion d'héritage, il existe la notion du `virtual`. Le rajout de ce mot clé devant une fonction permet aux classes filles de redéfinir la fonction par *over-riding*. Nous devons implémenter un simili du `virtual` avec notre `@virtual`. Il faut préciser que toute fonction `virtual` à l'intérieur d'une classe est déclarée membre automatiquement.

Il nous est aussi demander d'implémenter une sorte de « super classe », dont toutes les classes hériteraient. Il s'agit de la classe `Object`.
Nous allons donc distinguer deux cas ici, celui de l'héritage simple, à travers les notions d'agrégation d'interfaces et de vtables, et celui du `@virtual` avec une variante des vtables.

b. La classe Object

La classe `Object` est une « super classe » dont toutes les méthodes seront redéfinies dans toutes les classes déclarées. Pour donner une explication précise, reprenons l'exemple donné dans le cours :

```
@class Object
{
  @member const char      *nameInterface(); // nom interface
  Object                  *alloc(); // allocateur
  Object                  *new(); // constructeur par défaut
  @member
  {
    void                  init(); // initialiseur par défaut
    @virtual void         clean(); // nettoyeur par défaut
    void                  delete(); // destructeur par défaut

    @virtual int          isKindOf(const char *);
    @virtual int          isKindOf(Object *);
    @virtual int          isInstanceOf(const char *);
    @virtual int          isInstanceOf(Object *);
  }
}
```

Nous voyons donc qu'il y a des fonctions `@member`, et des `@virtual`. Une explication s'impose. Pour l'héritage des fonctions `@member`, le fils pourra appeler les fonctions du père. Ce qui permet par exemple, dans le cas de `init`, qu'à l'intérieur du constructeur d'une classe fille, nous puissions faire un `init` de la classe père, et utiliser ses fonctions/variables. Ces fonctions seront donc passées à toutes les classes qui en hériteront.
Pour les fonctions en `@virtual`, nous rentrons dans le cas du polymorphisme. Ce qui signifie que ces fonctions devront être redéfinies dans la classe fille si on veut les utiliser. Dans le cas de la class `Object`, les fonctions `clean`, `isKindOf` et `isInstanceOf` seront obligatoirement définies.

c. Héritage, variables et fonctions

En ce qui concerne l'héritage, toutes les pistes de réflexions pour le mettre en œuvre ont convergées vers une seule solution. C'est la seule qui nous a paru viable, et qui s'inspirait des informations données dans les cours et le TP 3.
Pour l'héritage simple, nous pouvons distinguer deux traitements à faire. Un pour les variables, et un autre pour les fonctions. Pour les variables, nous utiliserons la technique dite de l'agrégation d'interfaces. Cette agrégation prend la forme d'une structure comprenant la liste des variables, préalablement manglées, de la classe. Ensuite, pour chaque classe qui va

hériter de celle-ci, nous inclurons en plus les nouvelles variables rajoutées dans la classe fille. Voici un exemple concret :

```
@class Pere
{
     @member
     {
       int      a;
       float    a;
     }
}

@class Fils : Pere
{
     @member
     {
       int      a;
       float    a;
     }
}
```

```
/* Donnera dans le code C généré : */

typedef struct _class_Pere_
{
     int        M_4Pere_variable_signed_int_1a;
     float      M_4Fils_variable_float_1a;
} Pere;

typedef struct _class_Fils_
{
     Int        M_4Pere_variable_signed_int_1a;
     float      M_4Pere_variable_float_1a;
     int        M_4Fils_variable_signed_int_1a;
     float      M_4Fils_variable_float_1a;
} Fils;
```

Ensuite, pour l'héritage des fonctions, il faut introduire la notion de vtable (tables virtuelles). Que sont les vtables ?
Tout simplement une structure contenant l'ensemble des fonctions d'une classe, répertoriées sous forme de pointeurs. En effet, une structure en C ne peut contenir de déclarations fonctions, syntaxe oblige : les pointeurs sur fonctions sont donc ici obligatoires.

Ainsi, en plus de l'agrégation des variables, nous pouvons rajouter une agrégation des fonctions via la vtable de la classe.
Reprenons l'exemple au-dessus pour bien expliciter ce que nous allons faire. Rajoutons une fonction `print()` aux classes `Pere` et `Fils`. Comme décrit dans la partie concernant la « super classe » `Object`, les fonctions issues de cette dernière devront être redéfinies. Nous ne le faisons pas ici dans un souci de clarté.

```
@class Pere
{
  @member
  {
    int        a;
    float      a;
    void  print();
  }
}

@class Fils : Pere
{
  @member
  {
    int        a;
    float      a;
    void  print();
  }
}

/* Donnera dans le code C généré : */

typedef struct _class_Pere_
{
  /* les fonctions issues d'Object */
  typedef struct _vtable_Pere_
  {
    void (*V_4Pere_fonction_void_print_param_void)(struct
_class_Pere_*);
  };

  Int     M_4Pere_variable_signed_int_1a;
  float   M_4Pere_variable_float_1a;
} Pere;

typedef struct _class_Fils_
{
  /* les fonctions issues d'Object */
  typedef struct _vtable_Fils_
  {
    struct _vtable_Pere_  *vtP;
    void (*V_4Pere_fonction_void_print_param_void)(struct
_class_Fils_*);
  };

  int     M_4Pere_variable_signed_int_1a;
  float   M_4Pere_variable_float_1a;
  int     M_4Fils_variable_signed_int_1a;
  float   M_4Fils_variable_float_1a;
} Fils;
```

A cet héritage, il nous a été demandé de rajouter le mot clé `super` au moment de la traduction pour justement accéder aux variables/fonctions de la classe père. Ce mot clef ne s'applique pas pour le `@virtual`, uniquement pour l'héritage simple. Voici un exemple

d'accès à une variable d'une classe père à partir d'une classe fils à travers la fonction membre `print` du fils :

```
void Fils::print(void)
{
  [super.a] = 42;
}

/* Revient à faire : */

void Fils::print(void)
{
  M_4Pere_variable_signed_int_1a = 42;
}
```

d. @virtual

Le `@virtual` permet de rajouter le polymorphisme à l'héritage. Il ne s'applique qu'aux fonctions. Pour l'utiliser, il faut mettre le mot clé `@virtual` devant la fonction, qui devient alors « virtuelle ». Cette fonction sera toujours héritée dans une classe fille, mais pour l'utiliser, il faudra qu'elle soit redéfinie. Ce qui implique donc que d'une classe fille, on ne puisse pas appeler une fonction `virtual` père avec le `super`. Rappelons aussi qu'une fonction `@virtual` est automatiquement considérée comme une fonction `@member`.
Il faut donc ici bien différencier les fonctions héritées normalement et celles héritées à travers un `@virtual`. Dans l'héritage normal, nous pouvons rappeler toutes les fonctions définies dans les classes pères : elles apparaitront toutes dans la vtable de la classe fille. Pour le `@virtual`, cette fonction n'apparait qu'une seule fois dans la vtable fille, et ce sera la version redéfinie.
En reprenant l'exemple donné pour l'héritage dans le code C des exemples ci-dessus, il n'y aurait plus, dans la vtable, la fonction père ET la fonction fils, mais seulement la fonction fils redéfinie. Précisons aussi qu'une fonction `virtual` n'est pas forcément redéfinie. Sinon, il faudrait redéfinir les fonctions `@virtual` de la super classe `Object` du père dans toutes les classes fils. Cela deviendrait difficile et lourd à gérer.