Lefebvre Marie, Lesaffre Rémi
Baron Louis-Philippe, Parejo Pagador Pelayo

Kooc

Kind of Objective C

Préface:

Ce document présente les spécificités de notre langage KOOC. Ce langage est écrit dans le cadre du module B5 - Programmation Orientée Objet Avancée de la troisième année à EPITECH à des fins pédagogiques. Ce document est une documentation de conception destinée aux développeurs du langage.

Nous remercions Martin Chaine de nous avoir suivis et aidés tout au long de ce projet. Nous remercions Lionel Auroux pour les outils et les cours qu'il nous a fournis.

Sommaire :

Préface: 2
I Présentation générale et utilisation 4
Coté utilisateur 4
Côté développeur 5
II Conflits de symboles et module 6
La décoration de symbole ou Mangling 6
Le Module 7
L'Import 8
L'Implémentation 9
Appels Kooc 10
III Classes et instances 12
La Classe 12
Member 12
Allocateurs 12
Implémentation 13
Appel Kooc 13
Utilisation 13

I Présentation générale et utilisation

Coté utilisateur

a. *Introduction*

Le Kooc est un préprocesseur C réalisé en Python3 dont le but est d'ajouter au langage C, des fonctionnalités connues des langages orientés objet. La philosophie du langage veut qu'il y ai par défaut une compatibilité totale avec le C. La syntaxe du Kooc a donc été étudiée afin de ne pas entrer en collision avec la syntaxe du C. Par convention, un fichier source portera l'extension .kc et le header, l'extension .kh.

b. *Utilisation*

La compilation des sources se fera selon l'exemple suivant:

```
$> ls
test.kc test.kh
$> kooc test.kc
$> ls
test.c test.kc test.kh
$> kooc test.kh
$> ls
test.c test.h test.kc test.kh
$> gcc test.c -o my_binary_name
$> ls
my_binary_name test.c test.h test.kc test.kh
```

Que l'on pourra généraliser par :

```
$> ls
mod1.kc mod1.kh mod2.kc mod2.kh mod3.kc mod3.kh
$> for file in `ls`; do kooc $file; done
```

Notons que pour utiliser le kooc comme dans l'exemple qui suit, il sera nécessaire d'ajouter à la variable d'environnement PATH de son shell, le chemin complet vers le Kooc.

Exemple bash:

```
$> echo "export PATH=$PATH:/path/to/kooc_folder/" >> ~/.bashrc
```

Côté développeur

a. Python

Le compilateur Kooc est réalisé en python3. Notre code est divisé en divers modules ayant chacun un but distinct afin d'assurer sa modularité. En effet, chaque module possède une autogestion indépendante des autres modules. La maintenance et les modifications de comportements s'effectueront à une échelle micro sans risque de modification du comportement macro. Par exemple, le mangling peut être utilisé dans l'implémentation des différents éléments du langage Kooc (comme les classes, les modules...) sans que ceux-ci n'aient connaissance des spécificités du mangling.

Le "monkey patching" permet quant à lui de modifier le comportement d'une classe sans toucher à sa déclaration. Cela sert notamment à écrire une classe sur plusieurs fichiers et de regrouper les méthodes similaires de différentes classes dans un seul fichier. On peut ainsi modifier le comportement de Cnorm.

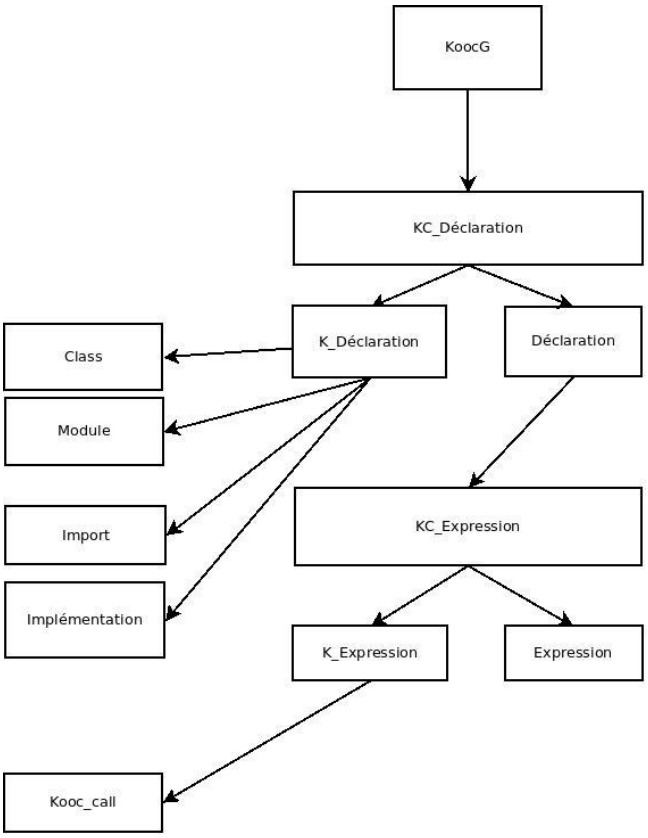
b. Pyrser, Cnorm

Pyrser est une librairie de lexer/parser. Cnorm, héritant de Pyrser, est une spécification de celui-ci pour le langage C. Son but est de construire un arbre syntaxique contenant les tokens de ce langage.

Le parseur Kooc a donc été réalisé via ces deux outils en complétant la Backus-NaurForm (BNF) de Cnorm avec celle du Kooc.

Afin de coupler le Kooc au C, nous avons décidé d'implémenter une BNF fusionnant les spécificités Kooc et C. Ces règles de BNF seront toutes préfixées de "KC_".

Elle est schématisée comme suit :



II Conflits de symboles et module

La décoration de symbole ou Mangling

Le mangling est une technique utilisée par bon nombre de compilateurs afin de résoudre les problèmes liés au besoin d'assigner un identifiant unique (symbole) à chaque entité. Elle consiste à décorer l'identifiant en y ajoutant sa description. Le nom ainsi généré sera bien propre à sa fonction et lui sera unique.

Le Kooc utilisera le mangling afin de contourner les limitations du C. La surcharge de fonctions et de variables sera implémentée grâce au mangling. En effet, deux variables de types différents auront d'apparence le même nom pour le développeur, mais un identifiant distinct pour le compilateur.

Cette propriété sera par la suite beaucoup utilisée par les directives @class et @module. (cf. les chapitres @Class et @Module)

Syntaxe:

Notre mangling s'inspire du mangling fait par g++. Le symbole modifié contiendra toujours le nom original déterminé par le développeur. On viendra alors y agrémenter tout attribut permettant de le distinguer d'un autre. Évidemment, rien n'empêche un développeur connaissant cette syntaxe d'accéder au symbole.

Noms:

L'encodage des noms de classe, de module ou de fonction sera préfixé au symbole par
la taille du nom. De plus, on ajoutera respectivement 'K' ou 'M' si on est dans le cas d'une classe ou d'un module.
Ainsi 'Class MyName' deviendra 'K6MyName'.

Types Primitifs, Pointeurs et Tableaux:

Ils seront encodés à partir de la première lettre de leur type. Ainsi char, int, short, long,
float, double seront encodés respectivement c, i, s, l, f, d. Exception faite pour les long long et les signed char qui deviendront x et Sc respectivement. Pour toutes les variables unsigned, nous préfixerons U au caractère qui les définit.
Les pointeurs ajouteront le préfixe P avant le type et les tableaux, le préfixe A. Dans le cas d'un tableau de pointeur, P sera avant le A.
Ainsi 'int* myVar' deviendra 'Pi5myVar' et 'int* myTab[]' deviendra 'PTi5myVar'

Qualifieurs:

Ils seront dotés de la première lettre de leur nom suivi d'un '_' et seront placés à droite de l'objet qu'ils qualifient, excepté dans le cas des fonctions (cf. Mangling des fonctions).
Ainsi 'const int * myPtr' deviendra 'PiC_5myPtr'.

Fonctions:

Elles commenceront par l'encodage du nom de la classe, suivit de '__', l'encodage du
qualifieur, celui du nom de la méthode et celui de ses paramètres.
Les paramètres seront encodés de la même manière que les types primitifs sauf que leur nom ne sera pas représenté.
Ainsi, 'Class::MyMethod(char, int) const' deviendra 'K5Class__C_8MyMethodci'.

Le Module

a. *Défauts du C*

Le C présente le problème de ne pas être assez modulaire. Nous devons donc commencer par ajouter la notion de bibliothèques de fonctions. Ainsi nous éviterons la collision des noms de symboles entre différentes unités de compilation.

b. *Encapsulation de données*

Les langages-objets ont depuis longtemps incorporé cette notion. Son utilisation permet de regrouper une série de déclarations de variables et de fonctions au sein d'un même espace nommé afin d'aider le développeur à organiser son code.

Le langage Kooc introduit donc la directive @module pour palier à ce problème son utilisation est semblable à un namespace où la surcharge de fonctions et variables de même nom est possible.

c. *Utilisation*

myHeader.kh:

```
@module myFirstModule
{
 int value;
 float buffer;
 void shoot();
 void shoot(int);
}
@module mySecondModule
{
 int value;
 float buffer;
 void shoot();
 void shoot(int);
}
```

myHeader.h retranscrit :

```
extern int M13myFirstModule__i5value;
extern float M13myFirstModule__f6buffer;
extern void M13myFirstModule__v5shoot();
extern void M13myFirstModule__v5shooti(int);
extern int M14mySecondModule__i5value;
extern float M14mySecondModule__f6buffer;
extern void M14mySecondModule__v5shoot();
extern void M14mySecondModule__v5shooti(int);
```

Le contenu du module est transformé en déclaration externe. De plus, grâce à la décoration (mangling) des variables et fonctions, la surcharge des variables et des fonctions est alors possible.

On notera que la directive @implementation sera nécessaire afin d'implémenter une fonction (cf. Chapitre sur L'Implémentation)

L'Import

a. *Présentation*

L'importation est le phénomène d'inclure un fichier header dans un fichier source. Cela permet d'éviter des réplications de code en les faisant automatiser par le compilateur. En C, cette directive se nomme #include.

Le langage Kooc introduit sa propre directive d'inclusion nommée @import. Elle permet, tout en gardant la compatibilité avec le C, d' automatiser la protection contre la multiple inclusion et d'importer des fichiers pour le langage Kooc.

b. *Utilisation*

myHeader.kh:

```
@module myModule
{
 int myVar;
 void myFunc(void);
}
```

Code source myFile.kc:

```
@import "myheader.kh"
@import "myheader.kh"
```

Code C retranscrit myFile.c:

```
#ifndef ~USER_PATH~_MYHEADER_H_
# define ~USER_PATH~_MYHEADER_H_
extern int M8myModule__i5myVar;
extern void M8myModule__v6myFuncv(void);

#endif /* ~USER_PATH~_MYHEADER_H_ */
```

Les headers seront gérés en deux temps:

Dans un premier temps, les .kh sont retranscrits dans un .h ne contenant que les définitions de fonctions et des références externes aux variables globales.

Dans un deuxième temps, le contenu du .h est copié dans le .c. On ajoute "#numero_ligne nom_fichier_header.kh flags". En cas d'import infini (ex : quand un fichier s'importe lui-même), l'import est empêché. Par sécurité on ajoute les macros de protection contre la double inclusion autour du code copié.

Un module peut être défini en plusieurs fois, en utilisant la directive @module à plusieurs reprises dans différents fichiers header. A chaque import les différentes parties sont fusionnées pour donner un unique module dans le fichier source. Cela permet à l'utilisateur de n'inclure, dans fichier source, que la partie qu'il va utiliser. Pour obtenir le module complet, tous les imports nécessaires doivent être effectués.

L'Implémentation

a. *Présentation*

Le langage Kooc introduit la directive @implementation afin de marquer le début de l'implémentation d'un module. Ce mot-clé marque aussi le début de toutes les créations de variables associées au module ainsi que leur assignation si elle avait été précisée dans le module.

b. *Utilisation*

myHeader.kh:

```
@module myModule
{
 int myVar = 42;
 void myFunc(void);
}
```

Fichier source myFile.kc:

```
@import "myHeader.kh"
@import "myHeader.kh"

@implementation myModule
{
 void myFunc(void)
 {
    printf("myFunction from myModule\n");
 }
}
```

Fichier généré myFile.c

```
#ifndef ~USER_PATH~_MYHEADER_H_
# define ~USER_PATH~_MYHEADER_H_
    extern int M8myModule__i5myVar;
    extern void M8myModule__v6myFuncv(void);
#endif /* ~USER_PATH~_MYHEADER_H_ */

int M8myModule__i5myVar = 42;
void M8myModule__v6myFuncv(void)
{
    printf("myFunction from myModule\n");
}
```

Appels Kooc

a. Utilité

Les Appels Kooc permettent de faire appel à une variable ou une fonction d'un module Kooc. Ils sont utilisés grâce à la syntaxe `[ ]`.
L'appel à un élément du module se fait alors grâce à un appel Kooc:

```
[MyModule.myvar]                    pour une variable
[MyModule func_name :p1 :p2 ...]    pour une fonction
```

Afin de régler d'éventuels problèmes de type, le Kooc introduit la syntaxe de typage explicite `@!(type)` pour le type global de l'expression et `(type)` pour les paramètres des fonctions.

b. Utilisation

myHeader.kh:

```
@module myModule
{
 int myVar = 42;
 void myFunc(int);
 int myFunc(void);
}
```

Fichier source myFile.kc:

```
#include <stdio.h>
@import "myHeader.kh"
@import "myHeader.kh"

@implementation myModule
{
 void myFunc(int var)
 {
    printf("myFunction from myModule with parameter %d\n", var);
 }
 int myFunc(void)
 {
    return [myModule.myVar];
 }
}
int main()
{
 int a;
 int b;
 @!(void)[myModule myFunc :(int)42];
 a = @!(int)[myModule.myVar];
 b = @!(int)[myModule myFunc];
 printf("a = %d\n", a);
 printf("b = %d\n", b);
 a == b;
 @!(int)[myModule.myVar] = 0;
 printf("%d\n", [myModule.myVar]);
}
```

Fichier généré myFile.c:

```
#include <stdio.h>
#ifndef ~USER_PATH~_MYHEADER_H_
# define ~USER_PATH~_MYHEADER_H_
        extern int M8myModule__i5myVar;
        extern void M8myModule__v6myFunci(int);
        extern int M8myModule__i6myFuncv();
```

```
#endif /* ~USER_PATH~_MYHEADER_H_ */
```

```
int M8myModule__i5myVar = 42;
void M8myModule__v6myFunci(int var)
{
    printf("myFunction from myModule with parameter %d\n", var);
}
int M8myModule__i6myFuncv()
{
    return M8myModule__i5myVar;
}
int main()
{
    int a;
    int b;
    M8myModule__v6myFunci(42);
    a = M8myModule__i5myVar;
    b = M8myModule__i6myFuncv();
    printf("a = %d\n", a);
    printf("b = %d\n", b);
    a == b;
    M8myModule__i5myVar = 0;
    printf("%d\n", M8myModule__i5myVar);
}
```

Nous avons implémenté une coercition (ou coercion en anglais) des types, c'est-à-dire une conversion implicite du type, pour les variables et fonctions non surchargées.

L'existence et la non-ambiguïté des symboles appelés sont vérifiées. Ainsi une erreur sera générée, si l'utilisateur essaie d'utiliser la coercition des types sur un symbole possédant plusieurs surcharges. Pour ce faire, chaque déclaration de symbole est référencée en interne par le précompilateur.

III Classes et instances

La Classe

a. *Défauts du Module*

Un module peut servir à définir un nouveau type de données et une bibliothèque de fonctions pour le manipuler. Cependant ce type est unique dans le programme. Nous devons trouver un moyen de l'allouer dynamiquement afin d'en avoir plusieurs.

b. *Type abstrait de données*

Le langage Kooc introduit donc la directive @class afin de supporter des types abstraits, c'est-à-dire des types avancés se comportant comme des types primitifs. Nous pouvons alors instancier des modules évolués par le biais de fonctions et données indépendantes ainsi que faire une allocation sur la pile ou sur le tas.

Member

La directive @member permet de définir les attributs et méthodes d'instance. Elle permet également de différencier les attributs et méthodes de classe de ceux d'instance.

Même si la directive @member n'est pas précisée devant une déclaration de fonction, si celle-ci prend en premier paramètre un pointeur sur la structure, elle est également considérée comme fonction membre, car elle permet bien de manipuler l'instance.

La somme des parties membres de la classe constitue alors une structure qui peut être allouée automatiquement sur la pile.

Allocateurs

Le type abstrait de données est caractérisé par son cycle de vie qui comprend les fonctions new, delete, init et clean.

New est un mot clef utilisé pour instancier un objet dans un appel KOOC (ex : "var = [MyClass new :param1 : param2]"). Il sera remplacé par le précompilateur en un appel à la fonction new correspondant aux paramètres de l'appel KOOC.

Une fonction *new* sera générée pour chaque fonction *init* définie par le développeur. Chaque fonction *new* fera un appel à la fonction *alloc* et un appel à la fonction d'initialisation *init* correspondant elle aussi aux paramètres de l'appel KOOC.

Delete est un mot-clef permettant de détruire une instance de classe. Elle correspond à l'appel d'une fonction de classe "~~delete_my_class~~(var)" qui libère tous les champs de l'instance (fonction non membre *clean*) et l'instance en elle-même. La fonction *clean*, si elle est définie par le développeur, permet de libérer en mémoire tous les champs d'une instance.

Si aucune fonction *init* n'est définie dans la classe, celle-ci sera générée par défaut; de même pour la fonction *clean*.

Implémentation

L'implémentation d'une classe et semblable à celle d'un module cependant, la classe rajoute la notation self afin d'accéder à l'instance courante d'une fonction membre (en plus du pointeur arg1 si présent)

Appel Kooc

Les appels Kooc d'une classe suivent la même syntaxe que ceux d'un module. Mais en plus, nous pouvons appeler des fonctions membres directement sur une instance. L'appel à la fonction de la classe sera effectué en fournissant automatiquement à la fonction le pointeur d'instance en premier paramètre.

En interne, l'existence des symboles est vérifiée comme pour les appels au module. De plus l'appel à un symbole de même nom et type présent dans une classe et un module de même nom, est interdit.

Utilisation

myHeader.kh:

```
@class Titor
{
 char * John;
 @member
 {
     char *dmail;
     char *get_dmail(void);
 }
 void send_dmail(Titor *this, char * mail);
 @member  int    year;
}
```

fichier source myFile.kc:

```
@import "class_test.kh"

@implementation Titor
{
 char *get_dmail(void)
 {
    return (“Hououin Kyouma”);
 }

 void send_dmail(Titor *this, char *mail)
 {
this->dmail = mail;
    printf("Send: %s\n", mail);
 }
}

int main()
{
 Titor t;
 [t.year] = 2036;
 printf("%d\n", [t.year]);
 [t send_dmail :"Watashi wa mad scientist !"];
 char * mail = [t get_dmail];
 printf("%s\n", mail);
}
```

fichier généré myFile.c:

```
#ifndef ~USER_PATH~_MYHEADER_H_
# define ~USER_PATH~_MYHEADER_H_
typedef struct K5Titor {
        char *~~mangling_member_class~~Pc5dmail;
        void *~~mangling_member_class~~Pv9get_dmailv(void);
        int ~~mangling_member_class~~i4year;
} Titor;

void ~~mangling_class~~v6deletePTitor(Titor *ptr)
    {
       free(ptr);
    }
void *~~mangling_class~~Pv3new()
    {
       return (malloc(sizeof (Titor)));
    }

#endif /* ~USER_PATH~_MYHEADER_H_ */

char *~~mangling_member_class~~Pc9get_dmailv(void)
{
        return ("Hououin Kyouma");
}
void ~~mangling_member_class~~v10send_dmailPTitorPc(Titor *this, char *mail)
{
        this->dmail = mail;
        printf("Send: %s\n", mail);
}
int main()
{
    Titor t;
    t.year = 2036;
    printf("%d\n", t.year);
    t.send_dmail("Watashi wa mad scientist !");
    char *mail = t.get_dmail();
    printf("%s\n", mail);
}
```