**Dossier de Conception**

**KOOC**

ELKAIM\_R

AMSTUT\_A

LEDARA\_F

DOUZIE\_L

BARAOU\_C

SLAIM\_L Octobre 2015

# I - Sommaire

[I - Sommaire 2](#__RefHeading__22_1540618119)

[II - Buts du document 3](#__RefHeading__24_1540618119)

[III - Documents applicables et de reference 3](#__RefHeading__26_1540618119)

[IV - Contexte de la conception 4](#__RefHeading__28_1540618119)

[V - Description des tâches 5](#__RefHeading__30_1540618119)

[VI - Conclusion 10](#__RefHeading__32_1540618119)

# II - Buts du document

Ce document concerne la réalisation d'une surcouche au langage C au moyen de l'outil Pyrser et Cnorm. Cette surcouche servira à augmenter les possibilités du langage:

* Utilisation de modules
* Ajout d'un système de classes
* Héritage simple/virtuel
* Notions de public/privé
* Exceptions
* ...

# III - Documents applicables et de r éférence

## A) Documents applicables

* Licence publique générale GNU: [http://w](http://w/)[ww.gnu.org/licenses/gpl-3.0.fr.html](http://www.gnu.org/licenses/gpl-3.0.fr.html)

## B) Documents de référence

* Sujet du KOOC: <https://intra.epitech.eu/module/2015/B-PAV-475-1/PAR-5-1/acti-187352/project/file/14_KOOC_Sujet.pdf>, Lionel Auroux
* Documentation de Pyrser: <http://pythonhosted.org/pyrser/>
* Documentation de Cnorm: <http://pythonhosted.org/cnorm/>

# IV - Contexte de la conception

## A) Présentation de l’application

L'application a pour but d'apporter une surcouche au langage C. Elle s'occupe d'une pré-compilation du programme. Ces améliorations visent à apporter certaines fonctionnalités des langages objets dans le langage C.

En effet, la programmation orientée objet apporte de multiples avantages. L'encapsulation permise par le système de classes permet d'architecturer un programme beaucoup plus simplement qu'en programmation impérative. De plus, le programme final permettra l'implémentation d'interfaces, qui servent à décrire un ensemble de fonctionnalités pour un composant logiciel.

A la fin de son exécution, notre programme renverra du code C compilable de façon classique

## B) Principales exigences applicables

**1) Exigences techniques:**

Le programme intervenant pendant la période de compilation, il doit pouvoir faire remonter les erreurs (syntaxe, domaine de définition, …) de façon claire à l'utilisateur.

Le fichier C renvoyé par le KOOC devra toujours être valide et ne pas avoir besoin d'être modifiépar l'utilisateur avant d'être compilé en exécutable.

**2) Exigences du langage:**

Les ajouts faits au langage C doivent avoir une syntaxe la plus simple possible, tout en étant distinguable simplement par rapport au langage de base.

Les mots clés utilisés par le KOOC doivent être similaires à ceux utilisés dans la plupart des langages objets, pour qu'un utilisateur puisse rapidement s'adapter.

# V – Description des parsings

A) Décorateur de symboles

Le décorateur de symboles a pour but de créer un symbole unique pour chaque variable et fonction présentes dans un module.

Nous allons donc créer une classe «Mangler» qui s'en occupera.

Cette classe se base sur l'utilisation de la classe "Module" Décrite dans la partie D. suivant la convention suivante; elle se base sur le prototype de chaque fonction contenue dans un module ou une classe pour établir son nom une fois en C:

**["Func"/"Var"]'$'["Module"/"Class" '\_' module/class name]?'$'OriginalName'$'[[P]\* [U]?'$'ReturnType'\_'[[P]\* [U]?'$'argsN'\_']["$$$variadic$$$"]?**

de cette manière, une fonction avec le prototype:

int toto(char \*stuff, void \*something);

dans le module tata donnera:

Func$Module\_tata$toto$$int\_P$char\_P$void

de même, une variable 'int foo' dans le module 'bar' donnera:

Var$Module\_bar$foo$$int

à noter:

* en cas de collision à l'issue de la décoration, un nombre sera ajouté à la fin, jusqu’à ne trouver aucune collision.
* le nom de chaque type commence par un $.
* un P est ajouté au nom du type selon la profondeur du pointeur (un void \*\* correspondrait donc à PP$void).
* un U correspond à un entier non signé et se retrouve après le(s) P symbolisant les pointeurs.

B) Typeur:

Le typeur a pour but de déduire le type de certaines variables et fonctions (retour, paramètres), pour permettre la surcharge de variables et fonctions dans du code.

C) Import

Le but de la fonction import est de connaitre des types déclarés dans un fichier .kh.

Pendant le parsing, à chaque rencontre d'un mot clé "@import fichier", nous allons faire un appelle récursif à Kooc pour parser ce fichier et en récupérer l'ast.

Grâce à cet ast, nous allons pouvoir lister toutes les déclarations faites dans ce fichier pour les mémoriser. Ces déclarations sont alors stockées dans une liste appartenant au singleton "DeclKeeper". Cela nous permettra ensuite de vérifier si les fonctions, variables et type utilisé dans le fichier courant sont connu.

Nous ajoutons ensuite à l'ast courant un objet " Raw("#include fichier\n") " afin d'ajouter la directive "Include" dans le fichier C généré.

D) Module

Le module va servir a rassembler des entités logiques du code. Il regroupe un ensemble de fonctions, variables et classes.

Pendant le parsing, chaque module rencontré via le catch de « @module » sera représenté sous la forme d'un objet « module ».

Cet objet module va stocker dans un dictionnaire chaque déclaration faite avec comme clef le symbole manglé. Il sera stocker dans le singleton "DeclKeeper" dans un dictionnaire avec comme clef le nom du module.

E) Classe

La classe se comporte comme un module mais définit en plus un type abstrait ce qui veut dire qu'il est possible de l'instanciée et donc de lui donner des fonctions et variables via le mot clef « @member »

@member : @member est un pré-fixe sur les variables ou les fonctions dans une classe pour désigner ses membres.

À chaque rencontre de "@class" la classe catchée va être stockée dans un objet de la classe "Classe" qui possèdera un dictionnaire de ses attributs membres et un autre de ses attributs non membres. À la création de cet objet un typedef de structure sera généré. Cet objet sera stocké dans le singleton "DeclKeeper".

F) Implémentation

L'implémentation est le code des fonctions qui se trouve dans les modules ou les classes.

Lors de la rencontre d'un "@implémentation" nous allons créer un objet implémentation qui va contenir un dictionnaire liant le symbole décoré d'une implémentation à son corps.

Un flag dans l'objet Kooc est utilisé pour savoir si nous sommes dans une récursion ou non pour ne pas récupérer les implémentations dans un fichier importé.

# VI - Description des transformations

La transformation est le processus après le parsing qui consiste a

A)

# VII - Conclusion

## A) Contraintes de conception

Les symboles générés par le KOOC doivent être lisibles par des humains pour un soucis de clarté.

## B) Contraintes de développement

Ce programme doit obligatoirement être écrit en langage Python puisqu'il repose sur l'utilisation de la librairie «Pyrser» et de l'outil «Cnorm».