Documentation Projet KOOC

Note : Ce document est un draft de la documentation finale, il représente juste en grandes lignes l'ossature de notre développement. Nous vous prions d'excuser les erreurs de frappe ou de non-sens à ce stade, et de vous attardez beaucoup plus sur le fond.

Etape 1

Etape 1 : @module, @implementation, @import, []

Partie 1:

But et objectif a atteindre:

Realiser un compilateur (un preprocesseur intelligent), qui a partir d'un fichier d'entree generera un fihier contenant du code compilable par le compilateur gcc.

1) Types de fichier d'entree et de sortie:
– Par defaut tout fichier passe au compilateur KOOC sera transforme en un fichier .c utilisable par gcc
– Les fichiers avec l'extension .kh sont transformes en .h
– Les fichiers .kc sont transformes en .c

2) Processus de compilation/preprocessing KOOC : Les fichiers d'entree sont passes au prealable au preprocesseur cpp pour les premieres resolutions avant

d'etre traites avec KOOC. Ceci peut impliquer plusieurs passes, le detail du processus sera decrit de bout en bout dans la suite.

3) Contenu d'un fichier d'entree :

– Expression de type C (Tout ce qui n'a pas ete specifie pour le KOOC)
– Les modules : @module
– Les implementations : @implementation
– Les directives d'inclusion : @import
– L'operateur [] permet d'accder a une variable ou a une fonction d'un module

4) Contraintes liees au fonctionnement du compilateur KOOC :
– Une gestion optimisee des doubles inclusions
– Gestion des colisions dans la definition des symboles
– Gestion des initialisations globales et des methodes d'access externes
– Tout ce qui est C reste intacte, et est laisse au soin du compilateur gcc dans le fichier genere en sortie
– Les fonctionnalites voulues pour le KOOC seront implementees grace a une technique de decoration de symboles, et cette technique de decoration ne doit pas affecter le compilateur applique au fichier genere(ie impacter ses routines de compilation). C'est a dire les symbole decore avec KOOC devrons etre resolus avec gcc sans aucun souci.
– Preservation de la semantique du C

5) Technique de mise en oeuvre:

– Outils utilises : Codeworker, cpp, cnorm, cnorm2c
– Definition d'une grammaire pour le parsing des fichiers
– Definition d'une technique de decoration de symboles
– Technique de gestion d'inclusion (@import)
– Technique de gestion des modules et des interfaces (@implementation)

6) Point important. Il faut bien determiner ce qui est a gerer en priorite entre la gestion des @import et l'appel a cpp. Ceci sea clarifie dans technique d'implemantation

Partie 2) La BNF

```
–       @import:
                ImportRule  ::=     [ -> #readText(“@import”) #readCString ]* ->
#empty
                ;

–       @module / @implementation:
ModuleImplementation ::=    [ #readText(“@module”)
            | #readText(“@implementation”) ]
            ReadBlock
                ;

                ReadBlock                   ::=     // Lecture d'un block ecrit en C
pouvant contenir la                                                 // syntaxe
KOOC [], encadre par des accolades.
                ;
–       []:
CrochetKooc         ::=     [ TypePrecision ]? '[' ModuleName [ Variable | Function ]
']'
;

TypePrecision ::=   #readText(“@!(”) CType ')'
;

                Ctype           ::=     //La regle du cnorm qui lit les types
                ;

                ModuleName      ::=     #readIdentifier
                ;

                Variable        ::=     #readIdentifier
                ;

                Function        ::=     #readIdentifier [ ':' Parameter ]*
                ;

                Parameter   ::=     Variable | CrochetKooc | Literal
```

;

Literal ::= //Tous les literals de base du C
;

Partie 3) Les mots cles

1) Le @import

On fait un premier passage sur le fichier passe en entre pour gerer le preprocessing et les import.

- a chaque reconnaissance du mot clef import on cree un fichier temporaire avec une extension ".ktmp", ce fichier aura comme contenu le contenu du fichier import.
– On fais le preprocessing sur ce fichier temporaire.
– Et on remplace le import par le code genere par le preprocesseur du fichier temporaire.
– On sauvegarde dans une variable le nom du fichier import pour ne pas faire de double inclusions
– On supprime les fichiers temporaire.

2) Le @module

– On fais un passage qui va nous permettre de voir ou se trouvent les @module
– on va transformer tous les @module en son equivalent en C (ils seront decores) dans un fichier ".h"
– check si les declarations sont toutes implementee apres avoir parse tout le fichier

3) Le @implementation

– On fais un passage qui va nous permettre de voir ou se trouvent les @implementation
– on va transformer tous les @implementation en son equivalent en C (ils

seront decores) dans un fichier .c

4) L'operateur []

– on fais une premiere passe pour check si le module existe
– seconde passe pour check si la variable ou la fonction existe
– verification du nombre de parametre
– verification de la coherence des types, si le type ne corespond pas a la signature, une erreur est remontee
– autre passe pour check les eventuelle surcharge , determination du meilleur type de retour. Si aucune surcharge ne correspond, erreur.

II) Definition du mangling

separateur : _x_

chaque type correspond a son nom.

fonction: M<nom du module>_x_F<type de retour>_x_N<nom de fonction>_x_P1<type parametre 1>..._x_PN<type parametre N>

variable: M<nom du module>_x_V<type de la variable>_x_N<nom de la variable>

du coup on interdit _x_ dans tous les identifiants des modules.

exemple :

```
Module Poulet
    {
        int dindon(int glouglou, int rute, char cocorico);
        int crabe;
    }
```

notre mangling sera comme ceci:

fonction : Mpoulet_x_Fint_x_Ndindon_x_P1int_x_P2int_x_P3int
variable : Mpoulet_x_Vint_x_Ncrabe

Etape 2

Partie 1: parties communes avec le @module :

La premiere partie sea de faire une passe pour check si il y a des definitions sans le @member, car ils seront traites de la meme manière que le @module.

On va appliquer la meme decoration, la generation de .kc et .kh seront pareil, on interdit les 'inline', les meme ignatures avec le mot cle 'const et static non decore.

Au final ce qui va diferrencier le @module de @class est la creation d' un type utilisateur.

Partie 2 : @member et le type utilisateur :

Dans notre premiere passe on va ajouter @class et @member dans notre parsing.

Pour chaque '@class' rencontre on va créer un type utilisateur correspondant au nom de ' l'@class' . Le type utilisateur ne sera pas decore , il s'agira d'un typedef struct.

Dans cette structure il y aura seulement les donnes membres pour chaques instance de l' '@class'.

La difference entre les fonctions du @module et du @ class est le premier parametre. Dans le @class l'instance est passee comme premier parametre.

Le mangling sera le meme , il n'y aura pas de changement on rajoute juste le premier parametre.

Lors de l'implementation dans le fichier '.kc' la syntaxe sera semblable a celle d'un appel de structure

Exemple :

.kh

```
@class jambon
{
    @member int cornichon ;
    @member void sauciflard(int nombreDeTranches);
    void pouletRoti(jambon *this,int painDeMie);
}
```

```
.kc
@implementation
{
@member Int cornichon = 666 ;
@member void Sauciflard(int nombreDeTranches)
    {
    for(int i = 0; i < nombreDeTranches; ++i )
    [self.cornichon ] --;
    }
    void PouletRoti(jambon *this,int painDeMie)
    {
        [this.cornichon] += painDeMie;
    }
}


.h
Typedef struct
{
    int cornichon;
}   jambon;


extern
Mjambon_x_Fvoid_x_Nsauciflard_x_P1jambonPT_x_P2int(jamb
on *self, int nombreDeTranches);
```

```
extern
Mjambon_x_Fvoid_x_NpouletRoti_x_P1jambonPT_x_P2int(jamb
on *this, int painDeMie);
```

.c

```
Mjambon_x_Fvoid_x_Nsauciflard_x_P1jambonPT_x_P2int(jamb
on *self, int nombreDeTranches)
{
for(int i = 0; i < nombreDeTranches; ++i )
    self->cornichon --;
}
```

```
Mjambon_x_Fvoid_x_NpouletRoti_x_P1jambonPT_x_P2int(jamb
on *this, int painDeMie)
{
    this->cornichon += painDeMie;
}
```

[*alloc] , [*init] et [*new]:

[*alloc]: pour chaque @class une function alloc est generee permettant d'allouer l'espace memoire necessaire , elle est manglee non membre.

[*init] : l'utilisateur va créer le [*init] initialiser le @class, il pourra donc surcharger l'init pour avoir plusieurs constructeur, le type de retour de init est obligatoirement un void, init est une fonction membre.

[*new] : combine [*alloc] et [*init] ceci va instancier le @class , il y a autant de new que de init, elles seront generees par kooc . Il va falloir

interdire a l'utilisateur la possibilite de créer une fonction qui s'appel 'new', new est non membre.

Un new par defaut sera genere pour les classes sans fonction membre init.

[*delete] et [*clean] :

clean : correspond au destructeur en c++, il est l'oppose de init, et c'est a l'utilisateur de la definir, clean est une fonction membre .

delete : le delete va appeler la fonction 'clean' si elle a été definie et desallouer l'instance de l'objet sur lequel il a été appele, il est creer par kooc et est une fonction membre. Une fonction delete est generee pour chaque surcharge de clean.

Si jamais le clean n'a pas été defini le delete genere ne fera pas appel a clean et l'instance va etre detruite.

Exemple :

```
.kc
Int main()
{
Jambon   jean;
Jambon * bon ;
```

```
[jean init :42] ;
[bon new :42] ; //Allocation et appel de init(int) ;
[jean clean] ;
[bon delete] ; //Desallocation et appel de clean() ;
}
```

.c

```
Int main()
{
Jambon  jean;
Jambon * bon ;

Mjambon_x_Fvoid_x_Ninit_x_P1jambonPT_x_P2int(&jean,
42);
bon = Mjambon_x_FjambonPT_x_Ninit_x_P1int(42);
Mjambon_x_Fvoid_x_Nclean_x_P1jambonPT(&jean);
Mjambon_x_Fvoid_x_Ndelete_x_P1jambonPT(bon);
}
```