KOOC - TP 1ère Partie

Sommaire

1) Etape 1

Objectif

Vous devez fournir un programme nommé 'kooc' qui va à partir d'un fichier source KOOC générer un programme en C. La grammaire du C à utiliser est celle fournie dans les précédents TP nommé "cnorm.cwp".

Chaîne de production

fichier source KOOC *.kc
fichier header KOOC *.kh

L'ensemble des scripts fournis correspond à la convention établie au début du module KOOC, TP 1 codeworker.

Vous pouvez avoir besoin d'utiliser des fichiers intermédiaires pendant votre génération. Dans ce cas, ils doivent avoir une extension commençant par 'k'.

Exemple d'utilisation:

```
#export PATH=$PATH:/le/chemin/ou/vous/avez/mis/kooc
#export KOOC_PATH=/le/chemin/ou/vous/avez/mis/kooc
#ls
test.kc
#kooc test.kc
#ls
test.kc test.c
#gcc test.c -o test
#./test
```

2) @import

Synopsis

```
@import "votre_fichier.kh"
```

Description

@import est une version évoluée de **#include** sauf que celui ci gère la multiple inclusion. Il permet aussi de ne charger qu'un module d'un fichier.

Exemple

```
------ test_import.kh
#define MAXBUF          4200  // on doit appliquer cpp sur le code avant

typedef struct          test_s
{
      int         size;
      char        st[MAXBUF];
}                 test_t;

static inline test_t    *test_new(char          *str)
{
      test_t            *test;

      if (!str)
            return (0);
      test = calloc(1, sizeof (test_t));
      test->size = strlen(str) > MAXBUF ? MAXBUF : strlen(str);
      memcpy(test->st, str, test->size);
      return (test);
}

static inline void      test_print(test_t *test)
{
      printf("%s", test->st);
}

+------ muf.kh
@import     "test_import.kh"
@module nia
{
  int a = 42;
}

@module bla
{
  int a = 666;
}

@module blu
{
  int a = 999;
}
```

```
------ test_import.kc

@import     "test_import.kh"
@import     "test_import.kh" /* faite attention au commentaire inutile */
@import     "test_import.kh" // oui je sais c en C++ mais comme GCC le
prend :)
@import     "muf.kh"

int               main(int ac, char **av)
{
      test_t            *test;
      char        my_buf[MAXBUF];
      char        fmt[13];// pourquoi 13!!peut pas avoir plus 10 milliards

      test = test_new("KOOC rulez\n");
      test_print(test);
      free(test);
      printf(«blu %d\n», @!(int)[blu.a]);
      printf("Tape un truc (c-D pour finir):");
      sprintf(fmt, "%%%ds", MAXBUF);
      scanf(fmt, my_buf);
      test = test_new(my_buf);
      test_print(test);
      free(test);
}
```

3) @module { ... }

Synopsis

```
@module Nom_du_module
{
....
....ici plein de declaration en C
....
}
```

Description

@module marque le début des déclarations des modules écrits en KOOC. Ceci veut dire que TOUS les symboles (fonctions et variables, sauf exception précisée) sont "décorés" en d'autre terme "manglés" Le plus souvent @module est mis dans un '.kh'. On doit importer ce fichier '.kh' dans tous les programmes qui vont utiliser les fonctions ou les variables du module. Notamment dans le fichier '.kc' qui apporte l'implémentation de ce module.

Attention, ce fichier '.kh' contient les déclarations des variables mais ces variables ne doivent pas être créées dans tous les fichiers qui incluent ce fichier.

Exemple

```
--------- test.kh
#include <stdlib.h>     // on doit appliquer cpp sur le code avant

@module           Test
{

      int   toto = 10; /* toto est une variable globale au module Test
                          et doit donc etre initialisé */
      double      toto = 0.2;
      int   toto();
      void  toto();
      void  toto(int);
      void  toto(const char);
      void  toto(const char* const);
}
```

Quelques précisions :

Comment dois-je gérer le mot-clé 'static' ?

Comme vues précédemment les variables et fonctions statiques sont locales à un modules. Elles ne seront pas appelées de l'extérieur, vous n'aurez pas de "décoration" à y apporter.

Comment dois-je gérer le mot clé 'const' ?

Une variable **'const'** est **'read only'**, vous ne pouvez pas traiter un appel KOOC qui tente d'écrire dessus. Attention même si **'int toto;'** et **'const int toto;'** ont 2 signatures différentes, ils ne peuvent apparaître en même temps dans le même module sans générer d'erreur. Sinon vous ne pourriez pas traiter **x = [Test.toto];**

Comment dois-je gérer le mot clé 'inline' ?

Le mot clé **'inline'** induit un comportement particulier du compilateur suivant les options de compilations.

Les fonctions 'inlinées' peuvent dans un cas être de vraie fonction et dans l'autre considérés comme des macros.

Vous devez interdire l'utilisation du mot-clé **inline** dans un bloc **@module**.

Une utilisation réelle d'une fonction 'inlinée' est généralement associer au mot-clé **'static'**. Ce genre de fonction se retrouve le plus souvent dans des fichiers headers.

Dans un cas concret de programmation en KOOC, vous pourrez utiliser des fonctions **'static inline'** mais dans des fichiers headers et hors de tout bloc @module.

Par contre, des appels de type KOOC peuvent se trouver dans ce genre de fonction.

Comment dois-je gérer les typedef, struct, union, enum définie dans un module ?

Il n'est pas nécessaire de les décorer. Ainsi :

```
@module Gun
{
      struct blam {int c; double d;};

      void    shoot(struct blam&);
}

@implementation Gun
{
      void shoot(struct blam &s)
      {printf("blam %d %f\n", s.c, s.d);}
}

int main()
{
      struct blam u = {.d=4.3,.c=12};
      [Gun shoot :&u]
}
```

Toutefois, par convention d'écriture, on préférera éviter de les mettre dans le block d'un @module car leur effet restant identique au C, ce type d'écriture peut induire des erreurs d'interprétation.

Comment gérer les ambiguités ?

Attention dans certaine utilisation, il peut y avoir ambigüité à cause d'un nombre trop important de surcharge de fonction. Donc on autorise le cast sur expression KOOC mais de manière particulière. Pour cela on utilise la syntaxe **@!()** car il ne s'agit pas réellement d'un cast (conversion) mais plus d'une information de type particulière pour le langage KOOC.

Par exemple :

```
@module Test
{
      int         id = 42;
      char        *id = "quarante deux";
}

@implementation  Test {} /* empty */

int main(int ac, char **av)
{
      printf("%s %d\n", @!(char*)[Test.id], @!(int)[Test.id]);
}
```

4) @implementation { ... }

Synopsis

```
@implementation Nom_du_module
{
....
....ici plein de code en C
....
}
```

Description

@implementation marque le début de l'implémentation d'un module. Ce mot-clé marque aussi le début de toutes les créations de variables associées au module (exigence #3). Les symboles externes dans un tel bloc sont évidement à décorer dans le fichier '.c' généré.

Exemple

```
---------- test.kc
/*
  test de :
      @import,
      @module,
      @implementation
*/
@import "test.kh"

@implementation Test
{
/*
ceci est l'implementation
*/

int         toto()
{
  printf("je suis la fonction toto qui retourne un int\n");

  return (42);
}


void  toto()
{
  printf("je suis la fonction toto de base\n");
}

void  toto(int n)
{
      printf("je suis la fonction toto qui prend 1 parametre=%d\n", n);
}

void  toto(const char c)
{
```

```
      printf("je suis la fonction toto qui prend 1 parametre char=%c\n", c);
}

void  toto(const char* const s)
{
      printf("je suis la fonction toto qui prend 1 parametre char=%s\n", s);
}

}
```

5) [];

Synopsis en BNF

A rajoutez en tant que *primary_expression*

```
primary_expression ::=
... /* les regles existantes  */ ...
| appel_KOOC
;

appel_KOOC ::= '[' identifiant_module
      [identifiant_fonction list_parametre]
      | identifiant_variable
      ']'
;

list_parametre ::= [':' assignment_expression ]*
;

identifiant_module ::= identifier
;

identifiant_fonction ::= identifier
;

identifiant_variable ::= '.' identifiant
;
```

Description

[]; permet de faire un appel à une fonction ou une variable d'un module KOOC.

Il est évident que pour faire la différence entre un appel à '**int toto();**' ou '**void toto();**', il faut absolument considérer le type de retour dans la décoration.

Il faut aussi être capable de déterminer le type global de l'expression pour choisir correctement le bon candidat à l'appel.

Les appels KOOC sont à typage fort.

Toutefois dans un premier temps utilisez la syntaxe **@!()** pour noter le type d'une expression KOOC. Et dans un deuxième temps essayez de vous en passer.

Exemple avec @!

```
-------- main.kc
@import "test.kh"

void  main(int ac, char **av)
{
      int c;
      double d;

      c = @!(int)[Test.toto]; // acces a la variable de module int toto
      d = @!(double)[Test.toto]; // acces a la variable de module double toto
      @!(int)[Test.toto] = 12; // changement de la variable de module int toto
      @!(double)[Test.toto] = 1.2; // changement de la variable de module
double toto
      c = @!(int)[Test toto]; // appel fonction module int toto();
      @!(void)[Test toto];  // void toto();
      @!(void)[Test toto :(int)c]; // void toto(int);
      @!(void)[Test toto :(int)64]; // void toto(int);
      @!(void)[Test toto :(const char)'c']; // void toto(const char);
      @!(void)[Test toto :(const char*const)"64"]; // void toto(const char *
const);
}
```

Exemple sans @!

Dans l'exemple suivant on détermine les types des expressions par le traitement de l'AST.

```
-------- main.kc
@import "test.kh"

void  main(int ac, char **av)
{
      int c;
      double d;

      c = [Test.toto]; // acces a la variable int toto
      d = [Test.toto]; // acces a la variable double toto
      [Test.toto] = 12; // changement de la variable int toto
      [Test.toto] = 1.2; // changement de la variable double toto
      c = [Test toto]; // int toto();
      [Test toto];  // void toto();
      [Test toto :c]; // void toto(int);
      [Test toto :64]; // void toto(int);
      [Test toto :'c']; // void toto(const char);
      [Test toto :"64"]; // void toto(const char * const);
}
```

Dans l'exemple précédent, il est à noter que le type des constantes sont interprétées de la façon suivante :

12 est un 'signed int'
1.2 est un 'double'
'c' est un 'const char'
"muf" est un 'const char *'

6) Tips

Gestion des erreurs

Pour la gestion d'erreur, il est important que les erreurs KOOC soient gérées au niveau KOOC alors que les erreurs C sont traité par le C. Pour vous aider, utilisez les "line specifier" avec # comme dans l'exemple suivant dans votre code générer:

```
------ fichier monfichierkooc.c generer par votre kooc et contenant une erreur
int main;
#42 "monfichierkooc.kc"
++
------ EOF
      # kooc monfichierkooc.c
      # gcc monfichierkooc.c       // l'appel a gcc devra etre cache a la fin
      monfichierkooc.c:42: parse error before '++' token
```

«#numéro nom de fichier» est valide en C et permet de renseigner le compilateur sur le nom du fichier et le numéro de ligne à prendre en compte. Ici on trompe le compilateur en lui indiquant qu'on est à la ligne 42 du fichier monfichierkooc.kc alors que le fichier est généré et que le numéro de ligne est différent.

Votre arbre sera décoré avec ces informations de type donc attention pour le «symbol mangling».

Au sein de Codeworker reportez-vous au directive **#try #catch**.

Coercion de Type / cast

La coercion de type, c'est la possibilité de contraindre une expression à un type donné. Les types des variables sont clairement connus au moment des appels. Sur l'exemple de code sans **@!()** la coercion des types rentre en jeux.

VOUS DEVREZ UTILISER LE CNORM, LE CNORM2C ET LA PATCHLIB POUR LE PROJET.