Sémantique et TDL

Projet

Compilation du langage μOC

1 But du projet

Il s'agit ici d'écrire un compilateur pour le langage μOC dont la grammaire est donnée en annexe. Ce compilateur devra engendrer du code pour la machine virtuelle TAM et sera conçu dans l'idée d'engendrer du code pour d'autres assembleurs avec le minimum de changement.

μOC est une extension objet du langage μC, lui-même sous-ensemble du langage C.

Le projet comporte donc deux parties :

1. Ecriture d'un compilateur pour μC.
2. Ecriture d'un compilateur pour l'extension objet de μC qui se rapproche, par la syntaxe, du langage Objective-C (http://fr.wikipedia.org/wiki/Objective-C) .

2 μC

Parmi les concepts présentés par μC, on peut citer

- Les types sont quelques types de base (int, char) et le type Pointeur. Le pointeur null est noté NULL.
- Pas de type booléen (comme en C : l'entier 0 représente le faux, les autres le vrai).
- La définition de fonctions, éventuellement récursives.
- La possibilité d'inclure du code TAM 'inline'.
- Quelques opérations arithmétiques et booléennes
- Le cast.

Voici un exemple de programme μC (sans intérêt particulier).

Listing 1: Exemple de programme μC

```
// assembleur inline en dehors d'une fonction
asm {
   CALL (LB) _main   ; appel au point d'entree du programme
   HALT              ; arret de la machine
   }

int foo(int n){
    // declaration sans initialisation
    int xxx;
    xxx = 100;
    // declaration avec initialisation
    int a = n+999;
    // declaration d'un pointeur
    int *m = malloc(1);
    a = *m +666;
    // declaration d'un pointeur sur pointeur
    int ** k = malloc(1) ;
    *k = m;
    ** k = 12;
    int b = **k + 9999;
    // cast
    m = (int *)malloc(1);
```

```
        return m;
    }

26  // fonction renvoyant un pointeur
    int * ref(int x){
      int *p = malloc(x);
      return p;
      }
31
    // fonction illustrant les expressions arithmethiques
    int bar(int a, int b){
       int u = 301;
       int v = 401;
36     int result;
       result = foo(33*a - b > (-55*u/22%11/v));
       return result;
    }

41  // Assembleur inline dans une fonction
    void log(char *message, int valeur){
      int x = 12;
      asm {
        LOAD %message          ; acces au premier parametre
46      SUBR Sout              ; afficher message
        LOAD %valeur           ; acces au deuxieme parametre
        LOADL 1
        SUBR IAdd
        STORE %x               ; ecrire valeur + 1 dans x
51      LOAD %x                ; acces a la variable x
        SUBR IOUT              ; afficher x
        SUBR LN
        RETURN (2) 0
        }
56  }

    // fonction illustrant la manipulation de pointeurs.
    // point d'entree du programme.
    int main(){
61      log("Hello, world ! ", bar(98,99));
        char c = 'a';
        int y = 999;
        // instruction conditionnelle
        if (y ==0){
66         log("alors",y);}
        else {
           log("sinon", y);
        }
        return y;
71  }
```

3 μOC

μOC étant une extension de μC, il est donc possible d'écrire en μOC un programme μC.

Bien que la syntaxe de μOC soit inspirée de celle de Objective-C, certains concepts de μOC sont plus proches de Java que d'Objective-C.

3.1 Principaux concepts de μOC

Parmi les concepts présentés par μOC (en plus de ceux de μC) , on peut citer

- La classe NSObject, parente de toutes les classes.
- La définition d'une classe et du type associé.
- L'héritage et le sous-typage associé.
- L'accès aux attributs et méthodes d'instance.
- L'accès aux attributs et méthodes de classe.
- La possibilité d'appeler une fonction μC dans une instruction d'une méthode de μOC.
- L'appel de méthodes par liaison tardive

Voici un exemple de programme μOC.

Listing 2: Exemple de programme μOC

```
// assembleur inline en dehors d'une fonction
asm {
   CALL (LB) _main   ; appel au point d'entree du programme
   HALT              ; arret de la machine
   }

// la classe mere de toutes les classes
@class NSObject{
}
+(id)init {}
@end

// un point
@class Point2D : NSObject{
    int x;
    int y;
}
+(id)init {
  x=0;
  y=0
}
-(int)x {
    return x;
}
-(int)y {
    return y;
}
-(void)x:(int)a y:(int)b{
    x = a;
    y = b;
}
@end

// un point colore
@class Point2DCol:Point2D{
    int col;
}
+(id)init {
  [super init];
  col=0;
}
-(int) col{
  return col;
}
-(void) x:(int)a y:(int)b col:(int)c{
    [self x:a y:b];
```

```
        col = 99;
    }
@end

int main()
{
    // creation d'un point colore
    Point2DCol *org = [Point2DCol init];
    // compatibilite de id avec tout objet
    id object = org;
    // appel de methode sur un objet
    [org x:0 y:0 col:4];
    // appel accesseur
    int c = [org col];
    // appel de fonction MC
    log(@"PointColore = ",[org x], [org y], [org col]);
}
```

NB. Ces concepts sont le minimum à réaliser.

NB2. Toutes les classes à compiler seront dans un même fichier.

NB3. Dans le fichier à compiler, les classes seront supposées définies avant d'être utilisées (pas de référence en avant).

NB4. Dans une classe, les attributs et les méthodes seront supposés définis avant d'être utilisés (pas de référence en avant).

NB5. La redéfinition d'une méthode dans une sous-classe doit être possible, mais la surcharge d'une méthode n'est pas à prendre en compte.

3.2 Système de types

- Le type BOOL est le type des booléens (avec vrai = YES et faux = NO).
- μOC ajoute donc à μC le type associé à une classe. Ce type sera donc défini de manière interne par un pointeur sur un 'struct' bien que le type 'struct' n'existe pas en MC.
- Une instance de la classe K ne peut être définie que comme pointeur sur le type associé à la classe K :

```
K  obj ;     // incorrect
K *obj ;     // correct
```

  Attention : la grammaire autorise la déclaration incorrecte, il faudra donc faire ce contrôle sémantiquement.

- μOC définit de plus un type 'id'. Il s'agit d'un type anonyme, compatible avec tout type associé à une classe. Ce type est utilisé quand on ne veut pas spécifier le type d'une expression (par exemple comme type de retour). Joue un peu le rôle de Object en Java. Attention, ce type est déjà un pointeur, ne pas mettre l'$*$ pour déclarer un objet de type 'id'.

```
id *obj ;     // incorrect
id  obj ;     // correct
```

3.3 Différences avec Java

- L'objet courant s'appelle 'self' et non 'this', mais son rôle est le même.
- L'objet 'super' fait référence au self de la surclasse.
- Le signe + devant un attribut (une méthode) indique que c'est un attribut (une méthode) de classe. Le signe $-$, que c'est un attribut (une méthode) d'instance.
- L'objet 'null' de Java est noté 'nil' en μOC et est compatible avec le pointeur NULL de μC.

- En μOC un objet d'une classe K est créé par l'appel à la méthode de classe 'init' définie dans la classe K (au lieu de 'new' et appel d'un constructeur en Java).

```
// Déclaration et initalisation d'un point p.
 Point2D *p = [Point2D init] ;
```

- Le nom d'une méthode est composé de sélecteurs :

  Par exemple dans la classe Point2D

```
// constructeur
+(id)init {
  if ([super init] = nil) ...
  }
// mise à jour des attributs d'un Point2D
-(void) x:(int)a y:(int) b{
   x = a;
   y = b;
  }
  // Accesseur de l'attribut x
-(int) x {
    return x;
}
```

  définit une méthode de classe 'init' sans paramètre, une méthode d'instance à 2 paramètres 'a' et 'b', de types int, associés aux sélecteurs 'x:' et 'y:' et une méthode d'instance (accesseur) 'x' sans paramètre.

  L'accès à une méthode ne se fait donc pas par une notation 'pointée', mais par un envoi de message à un receveur R (objet ou classe) par l'intermédiaire d'un ou plusieurs sélecteurs S_i et leurs arguments A_i:

$$[R\ S_1\ A_1\ \ldots\ S_n\ A_n]$$

```
// Déclaration et initalisation d'un point p.
 Point2D *p = [Point2D init] ;
 // maj de p
 [p x:42 y:23];

 int z = [p x] +1;
 P
 // On peut composer les appels
  int w = [[p x:42 y:23] x] +1;
```

  Attention, si une méthode est définie par plusieurs sélecteurs, on ne peut l'appeler qu'avec tous ses sélecteurs et dans l'ordre précisé à la définition.

```
   int w = [[p x:42 ] x];           // incorrect
   int w = [[p y:23 x:42 ] x];   // incorrect
```

4 Moyens et conseils

Le compilateur sera écrit en utilisant Java et le générateur de compilateur EGG étudié en TD et TP. L'utilisation d'Eclipse doit permettre de gagner du temps, mais n'est pas obligatoire.

Chaque partie du projet sera bien sûr basée sur :

- Une gestion de la table des symboles permettant de conserver des informations sur les fonctions, les variables locales (type, adresse, ...), les paramètres, sur les classes (héritage, types et sous-types, ...), les attributs (type, ...), les méthodes (signature, ...), etc. La gestion de cette TDS devra être votre premier travail car tout dépend d'elle que ce soit pour le typage ou la génération de code. Il est important de bien prendre en compte tous les besoins lors de sa conception car il sera difficile de revenir en arrière si vos choix s'avèrent peu pertinents.
- Le contrôle des types. Pour μOC, on réfléchira particulièrement au traitement du sous-typage et son rapport avec l'héritage.
- La génération de code. TAM est un assembleur simple pour la génération, mais il est demandé d'être le plus générique possible pour éventuellement traiter d'autres cibles. L'appel de méthode par liaison tardive est LA difficulté du projet.

Vous avez toute liberté pour l'organisation du travail dans le groupe, mais n'oubliez pas que pour atteindre votre objectif dans les délais, vous devez travailler en étroite collaboration, surtout au début pour la conception de la TDS. N'hésitez pas à nous poser des questions, (en séance de suivi de projet ou par mail) si vous avez des doutes sur votre conception.

5 Dates, Remise

Le projet a commencé ...

- Votre trinôme doit être constitué avant le lundi 14 Avril 8h. Vous me préviendrez de la composition du trinôme par mail à **marcel.gandriau@enseeiht.fr**
- Les sources (projet Eclipse et version 'make'), la documentation (TAM, EGG) et le présent sujet sont sur Moodle. Il y a un seul fichier MOC.egg pour μC et μOC mais la syntaxe de l'extension μMOC y est initialement commentée.
- La première partie du projet est à rendre pour le Vendredi 16 Mai 2014 18h. Une archive de votre projet (projet Eclipse ou version 'make'), sera envoyée par mail à **marcel.gandriau@enseeiht.fr** Cette partie sera rapidement testée pendant une séance de TP.
- La deuxième partie est à rendre pour le 4 Juin 2014 16h. Une archive de votre projet (projet Eclipse ou version 'make'), sera envoyée par mail à **marcel.gandriau@enseeiht.fr**

  Chaque archive contiendra :

  - Les sources de votre projet ainsi que les fichiers de test.
  - Un document (au format pdf uniquement) expliquant vos choix et limitations (ou extensions) dans le traitement. Ce document ne sera pas long, mais le plus précis possible (schémas) pour nous permettre de comprendre et juger votre travail.

NB. Vous avez bien sûr intérêt à commencer la deuxième partie dès que possible.
Bon courage à tous.

6 Grammaires

6.1 μC

La grammaire de μC est donnée sous deux formes :

- Une version récursive à gauche et non factorisée qui se prête mieux à la réflexion.
- Une version LL(2) qui est la seule acceptée par EGG.

Pour la transformation de la sémantique associée à l'élimination de la récursivité à gauche, et à la factorisation, vous pouvez exploiter la transformation systématique de la sémantique étudiée en cours et en TD.

Listing 3: Grammaire de μC, version non LL

```
-- Grammaire de MC
-- non factorisée et RG
Terminaux

separateur = "[\r\n\t ]+"
comm = "\/\/[^\n]*\n"
paro = "\("
parf = "\)"
aco = "\{"
acf = "\}"
virg = ","
pv = "\;"
affect = "="
si = "if"
sinon = "else"
void = "void"
asm = "asm"
int = "int"
char = "char"
retour = "return"
null = "NULL"
inf = "\<"
infeg = "\<="
sup = "\>"
supeg = "\>="
eg = "=="
neg = "\!="
plus = "\+"
moins = "\-"
ou = "\|\|"
mult = "\*"
div = "\/"
mod = "\%"
et = "\&\&"
non = "\!"
entier = "[0-9]+"
caractere = "\'[^\']\'"
chaine = "\"[^\"]*\""
ident = "[a-z][_0-9A-Za-z]*"

PROGRAMME -> ENTITES
ENTITES ->
ENTITES -> asm ASM ENTITES
ENTITES -> FONCTION ENTITES
--fonctions
FONCTION -> TYPE ident paro PARFS parf BLOC
-- parametres de fonctions
PARFS ->
```

```
PARFS -> PARF PARFSX
PARFSX ->
PARFSX -> virg PARF PARFSX
PARF -> TYPE ident
-- les types (simples et pointeurs)
TYPE -> STYPE REFS
REFS ->
REFS -> mult REFS
STYPE-> void
STYPE-> int
STYPE-> char
-- corps de fonction et bloc d'instructions
BLOC -> aco INSTS acf
-- instructions
INSTS ->
INSTS -> INST INSTS
-- declaration de variable locale avec ou sans init
INST -> TYPE ident AFFX pv
-- instruction expression
INST -> E pv
-- bloc d'instructions
INST -> BLOC
-- conditionnelle
INST -> si paro E parf BLOC SIX
SIX -> sinon BLOC
SIX ->
-- return
INST -> retour E pv
-- inline asm : ASM = instructions TAM
INST -> asm ASM
-- les expressions
-------------------------------------------------------------------
-- E = expression (y compr= l'affectation)
-- A = expression figurant dans une affectation
-- R = expression figurant dans une expresion relationnelle
-- T = expression figurant dans une expression additive (TERME)
-- F = expression figurant dans une expression multiplicative (FACTEUR)
-------------------------------------------------------------------
-- affectation
E -> A affect R
E -> A
-- relation
A -> R OPREL R
A -> R
OPREL -> inf
OPREL -> sup
OPREL -> infeg
OPREL -> supeg
OPREL -> eg
OPREL -> neg
-- additions ...
R -> R OPADD T
R -> T
OPADD -> plus
OPADD -> moins
OPADD -> ou
-- multiplication, ...
T -> T OPMUL F
T -> F
OPMUL -> mult
OPMUL -> div
```

```
    OPMUL -> mod
    OPMUL -> et
112 -- expressions de base
    -- Constante entiere
    F -> entier
    -- Constante chaine
    F -> chaine
117 -- Constante caractere
    F -> caractere
    -- expression unaire
    F -> OPUN F
    OPUN -> plus
122 OPUN -> moins
    OPUN -> non
    -- pointeur null
    F -> null
    -- expression parenthesee
127 F -> paro E parf
    F -> paro TYPE parf F
    -- appel de sous-programme
    F -> ident paro ARGS parf
    -- accès variable
132 F -> ident
    ---- acces zone pointee
    F -> mult F
    -- arguments appel de fonction
    ARGS ->
137 ARGS -> E ARGSX
    ARGSX ->
    ARGSX -> virg E ARGSX
```

Listing 4: Grammaire de μC, version LL

```
1  -- Grammaire de MC
   Terminaux

   separateur = "[\r\n\t ]+"
   comm = "\/\/[^\n]*\n"
6  paro = "\("
   parf = "\)"
   aco = "\{"
   acf = "\}"
   virg = ","
11 pv = "\;"
   affect = "="
   si = "if"
   sinon = "else"
   void = "void"
16 asm = "asm"
   int = "int"
   char = "char"
   retour = "return"
   null = "NULL"
21 inf = "\<"
   infeg = "\<="
   sup = "\>"
   supeg = "\>="
   eg = "=="
26 neg = "\!="
   plus = "\+"
   moins = "\-"
   ou = "\|\|"
```

```
mult =  "\*"
div =  "\/"
mod =  "\%"
et =  "\&\&"
non =  "\!"
entier =  "[0-9]+"
caractere =  "\'[^\']\'"
chaine =  "\"[^\"]*\""
ident =  "[a-z][_0-9A-Za-z]*"

PROGRAMME -> ENTITES
ENTITES ->
ENTITES ->  asm ASM ENTITES
ENTITES -> FONCTION ENTITES
--fonctions
FONCTION -> TYPE ident paro PARFS parf BLOC
-- parametres de fonctions
PARFS ->
PARFS ->  PARF PARFSX
PARFSX ->
PARFSX ->  virg PARF PARFSX
PARF -> TYPE ident
-- les types (simples et pointeurs)
TYPE -> STYPE REFS
REFS ->
REFS -> mult REFS
STYPE-> void
STYPE-> int
STYPE-> char
-- corps de fonction et bloc d'instructions
BLOC ->  aco INSTS acf
-- instructions
INSTS ->
INSTS ->  INST INSTS
-- declaration de variable locale avec ou sans init
INST -> TYPE ident AFFX pv
-- instruction expression
INST -> E pv
-- bloc d'instructions
INST -> BLOC
-- conditionnelle
INST ->  si paro E parf BLOC SIX
SIX ->   sinon BLOC
SIX ->
-- return
INST ->  retour  E pv
-- inline asm : ASM = instructions TAM
INST ->  asm  ASM
-- les expressions
-------------------------------------------------------------------------
-- E = expression (y compr= l'affectation)
-- A = expression figurant dans une affectation
-- R = expression figurant dans une expresion relationnelle
-- T = expression figurant dans une expression additive (TERME)
-- F = expression figurant dans une expression multiplicative (FACTEUR)
-------------------------------------------------------------------------
E ->   A AFFX
-- affectation
AFFX -> affect  A
AFFX ->
-- relation
```

```
A ->   R AX
OPREL -> inf
OPREL -> sup
OPREL -> infeg
OPREL -> supeg
OPREL -> eg
OPREL -> neg
AX -> OPREL R
AX ->
R ->  T RX
-- additions ...
RX ->   OPADD  T RX
OPADD -> plus
OPADD -> moins
OPADD -> ou

RX ->
T ->  F TX
-- multiplication, ...
TX ->   OPMUL  F TX
OPMUL -> mult
OPMUL -> div
OPMUL -> mod
OPMUL -> et
TX ->
-- expressions de base
-- Constante entiere
F -> entier
-- Constante chaine
F -> chaine
-- Constante caractere
F -> caractere
-- expression unaire
F ->  OPUN  F
OPUN -> plus
OPUN -> moins
OPUN -> non
-- pointeur null
F -> null
-- expression parenthesee
F ->  paro E parf
F ->  paro TYPE parf  F
-- appel de sous-programme
F ->  ident paro ARGS parf
-- accès variable
F ->  ident
---- acces zone pointee
F -> mult F
-- arguments appel de fonction
ARGS ->
ARGS -> E ARGSX
ARGSX ->
ARGSX -> virg  E ARGSX
```

6.2 μOC

Listing 5: Grammaire de μOC

```
-- Grammaire de MOC
-- *** Les règles de production spécifiques de MOC sont en fin de grammaire

--Terminaux

separateur = "[\r\n\t ]+"
comm = "\/\/[^\n]*\n"
paro = "\("
parf = "\)"
aco = "\{"
acf = "\}"
virg = ","
pv = "\;"
affect = "="
si = "if"
sinon = "else"
void = "void"
asm = "asm"
int = "int"
char = "char"
retour = "return"
null = "NULL"
inf = "\<"
infeg = "\<="
sup = "\>"
supeg = "\>="
eg = "=="
neg = "\!="
plus = "\+"
moins = "\-"
ou = "\|\|"
mult = "\*"
div = "\/"
mod = "\%"
et = "\&\&"
non = "\!"
entier = "[0-9]+"
caractere = "\'[^\']\'"
chaine = "\"[^\"]*\""
ident = "[a-z][_0-9A-Za-z]*"

-- MOC extension
dpts is "\:"
cro is "\["
crf is "\]"
id is "id"
classe is "@class"
fin is "@end"
self is "self"
bool is "BOOL"
super is "super"
yes is "YES"
no is "NO"
entier is "[0-9]+"
caractere is "\'[^\']\'"
chaine is "\"[^\"]*\""
ident is "[a-z][_0-9A-Za-z]*"
-- MOC
```

```
--ident de classe
identc is  "[A-Z][_0-9A-Za-z]*"
-- chaine
chaineo is  "@\"[^\"]*\""

:========== OBJC extension ==========
ENTITES -> IMPLEMENTATION ENTITES
-- definition d'une classe
IMPLEMENTATION ->  classe identc SUPER aco ATTRIBUTS acf METHODES fin
-- super classe
SUPER ->
SUPER -> dpts identc
ATTRIBUTS ->
ATTRIBUTS -> TYPE ident pv ATTRIBUTS
METHODES ->
METHODES -> METHODE  METHODES
METHODE -> QUAL PTYPE MPARFS BLOC
QUAL -> plus
QUAL -> moins
STYPE-> identc
STYPE-> bool
-- type "any"
TYPE -> id
-- type arg methode entre parentheses
PTYPE-> paro TYPE parf
-- parametres de methodes
MPARFS ->
MPARFS ->  MPARF MPARFS
MPARF ->  ident
MPARF ->  ident dpts PTYPE ident
-- object nil
F -> nil
--Constante 'yes'
F -> yes
-- Constante 'NO'
F -> no
F -> chaineo
F -> self
F -> super
F -> cro F MARGS crf
-- appel methode de classe
F -> cro identc MARGS crf
-- arguments appel de methode
MARGS ->
MARGS ->  MARG MARGS
MARG -> ident dpts  E
MARG -> ident
```