Sémantique et TDL

Projet

Compilation du langage $\mu C^{\#}$

Contents

1 But du projet

Il s'agit ici d'écrire un compilateur pour le langage $\mu C^{\#}$ 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 changements.

$\mu C^{\#}$ est une extension objet du langage μC, 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 $\mu C^{\#}$ de μC qui s'inspire du langage $C^{\#}$ de Microsoft.

2 Compilateur μC

Parmi les concepts présentés par μC, on peut citer

- Quelques types de base (int, char, void), le type struct 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.
- Quelques opérations arithmétiques et booléennes
- Le transtypage (cast).
- La possibilité d'inclure du code TAM 'inline'.
- Les chaines de caractères vues comme des (char *). Optionnel.

Voici un exemple de programme μC (sans intérêt particulier, syntaxiquement correct, mais sans doute sémantiquement incorrect).

Listing 1: Exemple de programme μC

```
// NB. Ce programme ne sert qu'a illustrer les concepts de mC
// et ne calcule rien de particulier !
// nom de type (doit commencer par une majuscule)
typedef int Entier ;

// variable globale
Entier x;

// nom de type donne a un struct
typedef struct {
  int x;
  int y;
} Point ;

// variable globale
Point p;


// fonction illustrant la manipulation de pointeurs.
int foo(int n, int m){
    // declaration de variable locale sans initialisation
    int xxx;
    xxx = 100;
    // declaration avec initialisation
    int a = n + m;
    // declaration d'un pointeur et allocation de la zone pointee
    int *m = (int *)malloc(1);
    a = *m +666;
    // declaration d'un pointeur sur pointeur
    int ** k = (int **)malloc(1) ;
    *k = m;
    // allocation
    *k = (int *)malloc(1) ;
    ** k = 12;
    int b = **k + 9999;
    // cast
    m = (int *)malloc(1);
    return m;
}

// fonction illustrant les expressions arithmetiques
```

```
int bar(int a, int b){
   struct { int x; int y; } *pt;
   *pt.x = 3;
   pt->y = 3;
   int u = 301;
   int v = 401;
   int result;
   result = foo(33*a - b > (-55*u/22%11/v));
   return result;
}

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

// Assembleur inline dans une fonction.
// malloc en TAM :
// params : Taille : entier (pre : >0)
// retour : entier (post : une adresse dans le tas)
int * malloc(int taille){
  int adr;
  // appel a l'allocation TAM
  asm {
    LOAD %taille     ; acces au parametre taille
    SUBR Malloc      ; allocation
    STORE %adr       ; resultat dans adr
  }
  return adr;
}

// fonction renvoyant un pointeur utilisant la fonction precedente
int * pointeur(int x){
  // declaration et allocation
  int *p = malloc(x);
  // appel fonction foo
  *p = foo(999, 1);
  // retourner le pointeur
  return p;
  }


// Assembleur inline dans une fonction
// log : afficher un message et une valeur
void log(char *message, int valeur){
  int x = 12;
  asm {
    LOAD %message        ; acces au premier parametre
    SUBR Sout
    LOAD %valeur         ; acces au deuxieme parametre
    LOAD %x              ; acces a la variable x
    SUBR IAdd
    SUBR IOUT
    SUBR LN
    }
}

// point d'entree du programme.
int main(){
    int a = 3;
```

```
        char c = 'a';
        int y = 999;
        // appel fonction
106     char c = 'a';
        int y = 999;
        int res = foo2 (y, (int) c);
        // appel fonction bar
        log("Hello, world ! ", bar(98,99));
111     // instruction conditionnelle
        if (y ==0){
           log("alors",y);}
        else {
           log("sinon", y);
116     }
        return 0;
    }
```

2.1 Grammaires de 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 étudiée en cours et en TD.

Listing 2: Grammaire de μC, non LL

```
2   -- Grammaires de MC                                    --
    PROGRAMME -> ENTITES
    ENTITES ->
    ENTITES -> ENTITE ENTITES
7   -- definition d'un nom de type (commence par une majuscule)
    ENTITE -> typedef TYPE identc ;
    -- definition d'une variable globale ou d'une fonction
    -- (commence par une minuscule)
    -- Variable
12  ENTITE ->  TYPE ident ;
    -- Fonction
    ENTITE ->  TYPE ident ( PARFS ) BLOC
    -- parametres de fonctions
    PARFS ->
17  PARFS ->  PARF PARFSX
    PARFSX ->
    PARFSX ->  , PARF PARFSX
    PARF ->  TYPE ident
    -- les types (de base, noms, struct et pointeurs)
22  TYPE -> STYPE  PTRS
    -- type de base
    -- des * pour definir un pointeur
    PTRS ->
    PTRS -> * PTRS
27  -- types de base
    STYPE-> void
    STYPE-> int
    STYPE-> char
    -- nom de type (commence par une majuscule)
32  STYPE -> identc
```

```
-- struct
STYPE -> struct { CHAMPS )
CHAMPS ->
-- un nom de champ commence par une minuscule
CHAMPS -> CHAMP CHAMPS
CHAMP -> TYPE ident ;
-- corps de fonction et bloc d'instructions
BLOC ->  { INSTS }
-- instructions
INSTS ->
INSTS ->  INST INSTS
-- declaration de variable locale avec ou sans init
INST ->  TYPE ident AFFX ;
-- instruction expression (affectation et appel de procedure)
INST -> E ;
-- bloc d'instructions
INST ->  BLOC
-- conditionnelle
INST ->   if ( E ) BLOC
INST ->   if ( E ) BLOC else BLOC
-- retour de fonction
INST ->  return  E ;

F ->  ( E ) FX
F ->  ( TYPE )  F
F -> * F
F -> ident FX
FX ->
-- acces champ
FX ->   .  ident FX
-- acces champ pointeur
FX ->   ->  ident FX
-- appel de sous-programme
FX ->   ( ES ) FX
-- arguments appel de sous-programme
ES ->
ES -> E ESX
ESX ->
ESX -> , E ESX
-------------------------------------------------------------------------
-- E = expression (y compris 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 = R
E -> A
-- relation
A -> R OPREL R
A -> R
OPREL -> <
OPREL -> >
OPREL -> <=
OPREL -> >=
OPREL -> ==
OPREL -> !=
-- additions ...
R -> R OPADD T
R -> T
```

```
OPADD -> +
OPADD -> -
OPADD -> ||
-- multiplication, ...
T -> T OPMUL F
T -> F
OPMUL -> *
OPMUL -> /
OPMUL -> %
OPMUL -> &&
-- expressions de base
-- Constante entiere
F -> entier
-- Constante chaine
F -> chaine
-- Constante caractere
F -> caractere
-- expression unaire
F -> OPUN F
OPUN -> +
OPUN -> -
OPUN -> !
-- pointeur null
F -> NULL
-- expression parenthesee
F -> ( E )
F -> ( TYPE ) F
-- acces variable
F -> ident
-- acces champ struct
F -> F . ident
-- acces champ pointeur sur struct
F -> F -> ident
-- appel de sous-programme
F -> F ( ES )
---- acces zone pointee
F -> * F
-- liste d'expressions, par ex. arguments appel de fonction
ES ->
ES -> E ESX
ESX ->
ESX -> , E ESX

---------- inline asm ----------
-- peut etre utile pour implanter des sous-programmes
-- directement en assembleur TAM.
ENTITE -> asm ASM
INST -> asm ASM
```

Listing 3: Grammaire de μC, LL

```
-------------------------------------------------
-- Grammaires de MC                            --
-------------------------------------------------
PROGRAMME -> ENTITES
ENTITES ->
ENTITES -> ENTITE ENTITES
-- definition d'un nom de type (commence par une majuscule)
ENTITE -> typedef TYPE identc ;
-- definition d'une variable globale ou d'une fonction
-- (commence par une minuscule)
ENTITE -> TYPE ident DECL
```

```
-- Variable
DECL -> ;
DECL -> FONCTION
-- Fonction
FONCTION ->  ( PARFS ) BLOC
-- parametres de fonctions
PARFS ->
PARFS ->  PARF PARFSX
PARFSX ->
PARFSX ->  , PARF PARFSX
PARF ->  TYPE ident
-- les types (de base, noms, struct et pointeurs)
TYPE -> STYPE  PTRS
-- type de base
-- des * pour definir un pointeur
PTRS ->
PTRS -> * PTRS
-- types de base
STYPE-> void
STYPE-> int
STYPE-> char
-- nom de type (commence par une majuscule)
STYPE -> identc
-- struct
STYPE -> struct { CHAMPS )
CHAMPS ->
-- un nom de champ commence par une minuscule
CHAMPS -> CHAMP CHAMPS
CHAMP -> TYPE ident ;
-- corps de fonction et bloc d'instructions
BLOC ->  { INSTS }
-- instructions
INSTS ->
INSTS ->  INST INSTS
-- declaration de variable locale avec ou sans init
INST ->  TYPE ident AFFX ;
-- instruction expression (affectation et appel de procedure)
INST -> E ;
-- bloc d'instructions
INST ->  BLOC
-- conditionnelle
INST ->   if ( E ) BLOC SIX
SIX ->    else BLOC
SIX ->
-- retour de fonction
INST ->  return  E ;

-- les expressions ----------
-------------------------------------------------------------------------
-- E = expression (y compris 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 -> =  A
AFFX ->
-- relation
A ->   R AX
```

```
AX -> OPREL R
AX ->
-- operateurs relationnels
OPREL -> <
OPREL -> >
OPREL -> <=
OPREL -> >=
OPREL -> ==
OPREL -> !=
R -> T RX
-- additions ...
RX -> OPADD T RX
RX ->
-- operateurs additifs
OPADD -> +
OPADD -> -
OPADD -> ||
-- multiplication, ...
T -> F TX
TX -> OPMUL F TX
TX ->
-- operateurs multiplicatifs
OPMUL -> *
OPMUL -> /
OPMUL -> %
OPMUL -> &&
-- expressions de base
-- Constante entiere
F -> entier
-- Constante chaine
F -> chaine
-- Constante caractere
F -> caractere
-- expression unaire
F -> OPUN F
-- operateurs unaires
OPUN -> +
OPUN -> -
OPUN -> !
-- pointeur NULL
F -> NULL
-- expression parenthesee
F -> ( E ) FX
F -> ( TYPE ) F
F -> * F
F -> ident FX
FX ->
-- acces champ
FX -> . ident FX
-- acces champ pointeur
FX -> -> ident FX
-- appel de sous-programme
FX -> ( ES ) FX
-- arguments appel de sous-programme
ES ->
ES -> E ESX
ESX ->
ESX -> , E ESX

---------- inline asm ----------
-- peut etre utile pour implanter des sous-programmes
```

```
-- directement en assembleur TAM.
ENTITE ->    asm ASM
INST ->   asm ASM
```

3 Compilateur $\mu C^{\#}$

$\mu C^{\#}$ étant une extension de μC, il est donc possible d'écrire en $\mu C^{\#}$ un programme μC.

3.1 Principaux concepts de $\mu C^{\#}$

Parmi les concepts présentés par $\mu C^{\#}$ (en plus de ceux de μC) , on peut citer

- La classe 'Object', parente de toutes les classes (cette classe est appelée 'object' dans le C# de Microsoft).
- La définition d'une classe et du type associé. Une classe peut-être définie dans un 'namespace' (proche d'un 'package' Java).
- L'accès à un 'namespace' par le mot-clef 'using' (proche du 'import' de Java).
- L'héritage et le sous-typage associé.
- L'accès aux attributs et méthodes d'instance.
- La possibilité d'appeler une fonction μC dans une instruction d'une méthode de $\mu C^{\#}$.
- L'appel de méthodes par liaison tardive.
- Le mode de passage des paramètres est semblable à celui de ADA. Par défaut, un paramètre est une donnée (= IN), avec le mot-clef 'ref' c'est un donnée/résultat (= IN OUT) , avec le mot-clef 'out' c'est un résultat (= OUT).

Voici un exemple de programme $\mu C^{\#}$.

Listing 4: Exemple de programme $\mu C^{\#}$

```
// NB. Ce programme ne sert qu'Ã  illustrer les concepts de mCS
// et ne calcule rien de particulier !

// Le namespace System est en C# predefini et contient les classes
// et les types de base

namespace System {
  // la classe mere de toutes les classes.
  class Object {
  // attributs
  // methodes
  }
}

// import de System
using System;

// on peut definir des entites mC dans un programme mCS
typedef struct {
  int x;
  int y;
  } Point;

int estRouge(int c){
  return c == 42;
  }

// les modes de passages de parametres en mCS sont plus riches qu'en mC
```

```
29  void swap(ref int a, ref int b){
      int tmp = a;
      a = b;
      b = tmp;
      }
34
    namespace Geometry {
      typedef struct {
        int x;
        int y;
39      } Point;
      // sous-namespace : Geometry.A
      namespace A {
        // un point : le nom complet de cette classe est Geometry.A.Point2D
        class Point2D : Object{
44        // attributs
          int x;
          int y;
          // Constructeur
          public Point2D(int a, int b){
49          x = a;
            y = b;
            }

          // methodes
54        public int getX() {
            return x;
            }
          public int getY() {
            return this.y;
59          }
          public void afficher(){
            // ...
            }
          } // classe Point2D
64      }
        // un autre sous-namespace
        namespace B {
          // un point colore sous-classe de Point2D.
          // le nom complet de cette classe est Geometry.B.Point2DCol
69        public class Point2DCol:Geometry.A.Point2D{
            // attributs
            int col;
            // Constructeur avec appel du constructeur de Point2D
            public Point2DCol(int a, int b, int c) : base(a,b){
74            col = c;
              }
            // methode
            public int getCol() {
              return col;
79            }
            private bool isRed(){
              return estRouge(col) != 0;
              }
            // redefinition
84          public void afficher(){
              // appel a la methode de la classe parente.
              base.afficher();
            // ...
            }
89
```

```
        } // classe Point2DCol
      }

      // suite du namespace A
94    namespace A {
        // ...
      }

    } // namespace Geometry
99
  using Geometry;

  // point d'entree du programme
  int main(){
104   Point2D p2d = new Point2D(3,4);
      Point p ;
      p.x = p2d.x +1;
      p.y = p2d.y +2;
      swap(p.x,p.y);
109   p2d = p;
      bool b = ((Point2DCol) p2d).isRed();
      Point2DCol pc = null;
      pc = p; // incorrect
      p  = pc ; // correct
114
      return 0;
  }
```

NB. Ces concepts sont le minimum à réaliser. Si vous décidez de ne pas traiter certains concepts plus secondaires de $\mu C^{\#}$ vous le préciserez dans le rapport. Si un concept n'a pas une sémantique très claire, vous pouvez proposer votre propre interprétation de ce concept.

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

NB3. Dans le fichier à compiler, un 'namespace' sera supposé défini avant d'être utilisé (pas de référence en avant).

NB4. Dans le fichier à compiler, une classe sera supposée définie avant d'être utilisée (pas de référence en avant).

NB5. Dans une classe, les attributs et les méthodes seront supposés définis avant d'être utilisés (pas de référence en avant).

NB6. 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 de $\mu C^{\#}$

- Le type 'bool' est le type des booléens.
- $\mu C^{\#}$ ajoute à μC le type associé à une classe. Ce type sera donc défini de manière interne comme un pointeur sur un 'struct'.

3.3 Différences entre $\mu C^{\#}$ et Java

- L'objet courant s'appelle 'this' comme en Java.
- L'objet 'base' fait référence au this de la surclasse (proche de 'super' en Java). On peut s'en servir pour appeler une méthode de la surclasse, ou le constructeur de la surclasse.
- Le signe + devant un attribut (ou une méthode) indique que c'est un attribut (ou une méthode) de classe. Le signe −, que c'est un attribut (ou une méthode) d'instance.
- L'objet 'null' de $\mu C^{\#}$ est compatible avec le pointeur NULL de μC.

4 Grammaires

La grammaire de $\mu C^{\#}$ 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 étudiée en cours et en TD.

Listing 5: Grammaire de $\mu C^{\#}$, non-LL

```
-------------------------------------------------------------
-- Grammaires de MCS
-------------------------------------------------------------
PROGRAMME -> ENTITES
ENTITES ->
ENTITES -> ENTITE ENTITES
-- definition d'un nom de type (commence par une majuscule)
ENTITE -> typedef TYPE identc ;
-- definition d'une variable globale ou d'une fonction
-- (commence par une minuscule)
-- Variable
ENTITE ->  TYPE ident  ;
-- Fonction
ENTITE ->  TYPE ident ( PARFS ) BLOC
-- parametres de fonctions
PARFS ->
PARFS ->  PARF PARFSX
PARFSX ->
PARFSX ->  , PARF PARFSX
PARF ->  TYPE ident
-- les types (de base, noms, struct et pointeurs)
TYPE -> STYPE  PTRS
-- type de base
-- des * pour definir un pointeur
PTRS ->
PTRS -> * PTRS
-- types de base
STYPE-> void
STYPE-> int
STYPE-> char
-- nom de type (commence par une majuscule)
STYPE -> identc
-- struct
STYPE -> struct { CHAMPS )
CHAMPS ->
-- un nom de champ commence par une minuscule
CHAMPS -> CHAMP CHAMPS
CHAMP -> TYPE ident ;
-- corps de fonction et bloc d'instructions
BLOC ->  { INSTS }
-- instructions
INSTS ->
INSTS ->  INST INSTS
-- declaration de variable locale avec ou sans init
INST ->  TYPE ident AFFX ;
-- instruction expression (affectation et appel de procedure)
INST -> E ;
-- bloc d'instructions
INST ->  BLOC
```

```
-- conditionnelle
INST ->   if ( E ) BLOC
INST ->   if ( E ) BLOC else BLOC
-- retour de fonction
INST ->  return  E ;

F ->  ( E ) FX
F ->  ( TYPE )  F
F -> * F
F -> ident FX
FX ->
-- acces champ
FX ->   .  ident FX
-- acces champ pointeur
FX ->   ->  ident FX
-- appel de sous-programme
FX ->   ( ES ) FX
-- arguments appel de sous-programme
ES ->
ES -> E ESX
ESX ->
ESX -> , E ESX
-----------------------------------------------------------------------
-- E = expression (y compris 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 = R
E -> A
-- relation
A -> R OPREL R
A -> R
OPREL -> <
OPREL -> >
OPREL -> <=
OPREL -> >=
OPREL -> ==
OPREL -> !=
-- additions ...
R -> R OPADD T
R -> T
OPADD -> +
OPADD -> -
OPADD -> ||
-- multiplication, ...
T -> T OPMUL F
T -> F
OPMUL -> *
OPMUL -> /
OPMUL -> %
OPMUL -> &&
-- expressions de base
-- Constante entiere
F -> entier
-- Constante chaine
F -> chaine
-- Constante caractere
F -> caractere
```

```
    -- expression unaire
    F -> OPUN F
    OPUN -> +
    OPUN -> -
    OPUN -> !
    -- pointeur null
    F -> NULL
    -- expression parenthesee
    F -> ( E )
    F -> ( TYPE ) F
    -- acces variable
    F -> ident
    -- acces champ struct
    F -> F . ident
    -- acces champ pointeur sur struct
    F -> F -> ident
    -- appel de sous-programme
    F -> F ( ES )
    ---- acces zone pointee
    F -> * F
    -- liste d'expressions, par ex. arguments appel de fonction
    ES ->
    ES -> E ESX
    ESX ->
    ESX -> , E ESX

    ---------- inline asm ----------
    -- peut etre utile pour implanter des sous-programmes
    -- directement en assembleur TAM.
    ENTITE -> asm ASM
    INST -> asm ASM

    ---------- CS extension ----------
    TYPE -> bool
    -- definit un contenu d'un namespace
    ENTITE -> namespace identc { ENTITES )
    -- donne l acces au contenu du 'namespace'
    ENTITE -> using IDC ;
    IDC -> identc
    IDC -> IDC . identc
    -- definition d'une classe (peut etre en dehors d'un namespace)
    ENTITE -> ACCES class identc HERITAGE { DEFS )
    -- acces
    ACCES -> public
    ACCES -> private
    ACCES ->
    -- heritage
    HERITAGE ->
    HERITAGE -> : IDC
    -- membres d'une classe
    DEFS ->
    DEFS -> ACCES DEF DEFS
    -- attribut ou methode
    DEF -> TYPE ident DECL
    -- constructeur
    DEF -> identc FONCTION
    -- Mode de passage pour les fonctions et methodes : rien : D, ref : D/R, out : R
    PARF -> MODE TYPE ident
    MODE -> ref
    MODE -> out ;
    F -> true
```

```
F -> false
F -> this FX
F -> null
F -> new IDC ( ES )
```

Listing 6: Grammaire de $\mu C^{\#}$, LL

```
-------------------------------------------------------
-- Grammaires de MC                                  --
-------------------------------------------------------
PROGRAMME -> ENTITES
ENTITES ->
ENTITES -> ENTITE ENTITES
-- definition d'un nom de type (commence par une majuscule)
ENTITE -> typedef TYPE identc ;
-- definition d'une variable globale ou d'une fonction
-- (commence par une minuscule)
ENTITE ->  TYPE ident DECL
-- Variable
DECL ->
DECL -> FONCTION
-- Fonction
FONCTION ->  ( PARFS ) BLOC
-- parametres de fonctions
PARFS ->
PARFS ->  PARF PARFSX
PARFSX ->
PARFSX ->  , PARF PARFSX
PARF ->  TYPE ident
-- les types (de base, noms, struct et pointeurs)
TYPE -> STYPE  PTRS
-- type de base
-- des * pour definir un pointeur
PTRS ->
PTRS -> * PTRS
-- types de base
STYPE-> void
STYPE-> int
STYPE-> char
-- nom de type (commence par une majuscule)
STYPE -> identc
-- struct
STYPE -> struct { CHAMPS )
CHAMPS ->
-- un nom de champ commence par une minuscule
CHAMPS -> CHAMP CHAMPS
CHAMP -> TYPE ident ;
-- corps de fonction et bloc d'instructions
BLOC ->  { INSTS }
-- instructions
INSTS ->
INSTS ->  INST INSTS
-- declaration de variable locale avec ou sans init
INST ->  TYPE ident AFFX ;
-- instruction expression (affectation et appel de procedure)
INST -> E ;
-- bloc d'instructions
INST ->  BLOC
-- conditionnelle
INST ->   if ( E ) BLOC SIX
SIX ->    else BLOC
SIX ->
```

```
-- retour de fonction
INST -> return E ;

-- les expressions ------------
-------------------------------------------------------------------------
-- E = expression (y compris 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 -> = A
AFFX ->
-- relation
A -> R AX
AX -> OPREL R
AX ->
-- operateurs relationnels
OPREL -> <
OPREL -> >
OPREL -> <=
OPREL -> >=
OPREL -> ==
OPREL -> !=
R -> T RX
-- additions ...
RX -> OPADD T RX
RX ->
-- operateurs additifs
OPADD -> +
OPADD -> -
OPADD -> ||
-- multiplication, ...
T -> F TX
TX -> OPMUL F TX
TX ->
-- operateurs multiplicatifs
OPMUL -> *
OPMUL -> /
OPMUL -> %
OPMUL -> &&
-- expressions de base
-- Constante entiere
F -> entier
-- Constante chaine
F -> chaine
-- Constante caractere
F -> caractere
-- expression unaire
F -> OPUN F
-- operateurs unaires
OPUN -> +
OPUN -> -
OPUN -> !
-- pointeur NULL
F -> NULL
-- expression parenthesee
F -> ( E ) FX
F -> ( TYPE ) F
```

```
    F -> * F
    F -> ident FX
119 FX ->
    -- acces champ
    FX ->   .  ident FX
    -- acces champ pointeur
    FX ->   ->  ident FX
124 -- appel de sous-programme
    FX ->   ( ES ) FX
    -- arguments appel de sous-programme
    ES ->
    ES -> E ESX
129 ESX ->
    ESX -> , E ESX

    ------------- inline asm ----------------
    -- peut etre utile pour implanter des sous-programmes
134 -- directement en assembleur TAM.
    ENTITE ->   asm ASM
    INST ->  asm ASM

    ------------------ CS extension ---------------------
139 TYPE -> bool
    -- definit un contenu d'un namespace
    ENTITE -> namespace identc { ENTITES )
    -- donne l acces au contenu du 'namespace'
    ENTITE -> using identc IDC ;
144 IDC ->
    IDC -> . identc IDC
    -- definition d'une classe (peut etre en dehors d'un namespace)
    ENTITE -> ACCES class identc HERITAGE { DEFS )
    -- acces
149 ACCES -> public
    ACCES -> private
    ACCES ->
    -- heritage
    HERITAGE ->
154 HERITAGE -> : identc IDC
    -- membres d'une classe
    DEFS ->
    DEFS ->  ACCES DEF DEFS
    -- attribut ou methode
159 DEF  ->  TYPE ident DECL
    -- constructeur
    DEF  ->  identc FONCTION
    -- Mode de passage pour les fonctions et methodes : rien : D, ref : D/R, out : R
    PARF ->  MODE TYPE ident
164 MODE -> ref
    MODE -> out ;
    F -> true
    F -> false
    F -> this FX
169 F -> null
    F -> new identc ( ES )
```

5 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, ...), les 'namespaces'.

  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 $\mu C^{\#}$, 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, (par mail, en séance de suivi) si vous avez des doutes sur votre conception.

6 Dates, Remise

Le projet a commencé ...

Les sources (projet Eclipse et version 'make'), la documentation (TAM, EGG) et le présent sujet sont sur Moodle.

La première partie du projet (μC)est à rendre pour le 20 Mai 2016 18h. Une archive de votre projet (projet Eclipse ou version 'make'), sera déposée dans la boite de dépôt Moodle 'Partie 1'

La deuxième partie ($\mu C^{\#}$)est à rendre pour le 10 Juin 2016 18h. Une archive de votre projet (projet Eclipse ou version 'make') sera déposée dans la boite de dépôt Moodle 'Projet Complet'.

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.

Bon courage à tous.