## Licence d'informatique - 3ème année

SAM Nensy, SIMONNOT Florent

# Compilateur du langage TPC

**13 avril 2019**

## Présentation

Ce projet consiste en la programmation d'un compilateur permettant de valider ou non un programme TPC et de la traduire en langage assembleur en cas de succès.

Le langage TPC est un sous-ensemble du langage C. D'ailleurs, nous avons modifié notre grammaire afin de rajouter des éléments de langage comme le C.

## Objectifs

1. **Compilateur:** le premier objectif est de créer un programme qui compile un programme TPC. Il doit vérifier la déclaration ou non des variables, constantes, fonctions utilisés. Il doit aussi typer les expressions. (Rendu 1)
2. **Traducteur:** Le programme doit aussi traduire le programme TPC source en langage assembleur nasm.

## Specifications

Il n'était pas demandé de réalisé une documentation pour ce premier rendu. Comme nous sommes allés un peu plus loin dans ce qui était demandé pour ce rendu, nous avons décidé d'expliquer nos choix et notamment le contenu rajouté.

Le compilateur ne gère pas encore les tableaux.

# Développement

1. Variables, constantes, macros

   La première phase de travaille consista à créer des structures permettant de sauvegarder des informations sur une variable, constante ou macro.

   Nous avons donc déclaré dans le fichier decl.h ces structures.

```
typedef struct {
      char name[MAXNAME];      //Name
      int type;                //Type
      int address;             //Adress
      int size;                //0 if not array
}STentry;
```

```
typedef struct {
      char name[MAXNAME]; //Name
      int type;           //Type
      int value;          //Value
      float valueFloat;   //Value if a float
}CONSTentry;
```

```
typedef struct {
      char name[MAXNAME]; //Name
      int type;   //Type
      int value;  //Value
      float valueFloat; //Value if a float
}MACROentry;
```

   Les structures constantes et macros comportent une variable pour la valeur entier et une variable pour la valeur flottante.

   Nous avons également créé une structure permettant de sauvegarder toutes les macros.

```
typedef struct{
```

```
   MACROentry *Mtable; /* table des macro constantes */
   int Msize;
   int Mmax;
}Mtable;
```

On y retrouve un tableau de type macro ainsi que la taille max du tableau et sa taille courante.

Afin de sauvegarder les variables et constantes nous avons utilisé une autre structure.

```
typedef struct cel {
      STentry *STtable;       /* table des variables */
      CONSTentry *Ctable;     /* table des constantes */
      int STmax;              /* taille max de la table des variables */
      int STsize;             /* taille courante de la table des variables
*/
      int Cmax;               /* taille max de la table des constantes */
      int Csize;              /* taille courante de la table des constantes
*/
      int current_stack_address; /* adresse de la variable dans la pile */
      struct cel *next;       /* prochaine table dans la pile */
} STStackCel, *STStack;
```

Cette structure comporte donc une table pour les variables et une pour les constantes.

On y ajoute la taille maximum et courante pour ces deux tables. Nous avons utilisé une variable pour préciser l'adresse de la variable ajoutée (cette adresse étant un multiple de 8). Puis nous avons ajouté un pointeur vers une même structure. Notre pile est donc en théorie une liste chaînée. Le premier bloc servira aux variables et constantes "globales". Puis s'il y a dans le fichier source des fonctions on ajoutera alors un bloc pour chaque fonction. Il faudra a priori supprimer les blocs au fur et à mesure mais nous le ferons au prochain rendu.

Ce système d'implémentation permet donc de pouvoir coder des variables globales et locales portants le même nom sans avoir d'erreurs.

Après avoir écrit ces structures, il nous a fallu coder les fonctions d'ajout de variables, constantes etc…

Les fonctions d'ajout sont écrites avec le même principe que ce soit une variable, une constante ou une macro (même si on utilise pas la même table).

```
int addVar(const char name[], int type, int is_parameter);
int addConst(const char name[], int type, int value, float valueFloat);
int addMacro(const char name[], int type, int value, float valueFloat);
```

Les fonctions prennent en paramètres le nom, le type un entier pour les variables qui précise si la variable est un paramètre d'une fonction, une valeur et valeur décimale pour les constantes et macros. Ces fonctions renvoient 1 en cas de réussite, 0 si l'ajout échoue.

A chaque ajout on vérifie par une fonction si une variable du même nom existe déjà. Ensuite, si la variable peut être ajouté on vérifie que la table n'est pas pleine (on réalloue la mémoire nécessaire dans le cas contraire).

On peut ensuite rechercher le premier indice de la table disponible et y ajouter les informations nécéssaire.

Après avoir gérer l'ajout de variables, constantes et macros nous avons implémenté des fonctions permettant de vérifier si une variable (resp. constante, macros) existe et si oui qui renvoie son type.

```
int lookup(const char name[], int is_tab);
```

Cette fonction prend en paramètre le nom de la variable (resp. constante , macro) recherché et un entier permettant de préciser si c'est un tableau. Cette fonction parcourt toutes les tables et renvoie son type ou -1 si aucune variable (constante ou macro) portant ce nom n'existe.

Pour les constantes nous avons codé une fonction qui vérifie si le nom passé en paramètre concerne une constante. Cette fonction est utilisé lors des affectations afin de vérifier que l'on n'affecte pas une valeur à une constante (ce qui est impossible dans le langage). Elle renvoie 1 si c'est une constante et 0 sinon.

```
 int isConstante(const char name[]);
```

Pour finir, nous avons coder des fonctions permettant d'afficher les tables.

```
void displayTable();
void displayConst();
void displayMacro();
```

## 2. Le typage

Partie importante d'un compilateur et plus difficile que la gestion des variables nous avons travaillé sur la typage des expressions.

Nous avons commencé par vérifier le type des variables utilisés lors d'une lecture (READE et READC). Il faut une variable de type int pour READE et une variable de type char pour READC. Nous avons donc écrit une fonction qui vérifie si deux types sont égaux. Elle renvoie 1 si les types sont égaux et 0 sinon.

```
int check_types(int a, int b);
```

Pour vérifier le type de la variable dans READE et READC on appelle la fonction lookup afin d'avoir le type de la variable (ou même une erreur si la variable n'existe pas) et en deuxième paramètre nous utilisons la macro INTEGER ou CHAR selon le contexte.

Nous avons ensuite écrit une fonction afin de caster les types dans les affectations comme indiqué dans le sujet.

```
int cast_type(int a, int b, int flag_cast);
```

La fonction prend deux types et un entier correspondant au type du cast ou -1 si il n'y a pas de cast. Cette fonction renvoie le type de l'évaluation entre ces deux types.

S'il y a un cast explicite alors le type renvoyé est le type de a.
Si les deux types a et b ont la même valeur on renvoie le type de a.
Si a est de type long et b de type int alors on renvoie un type long.
Si a est de type int et b est de type char alors on renvoie un type int.
Dans tous les cas suivants, le compilateur emmetra un warning :

- a est un char et b est un int
- a est un int et b est un long
- a est un char et b est un long

Nous avons ensuite créer une fonction qui renvoie le type des règles où il y a des ADDSUB et DIVSTAR. Pour cela la fonction prend deux entiers correspondant au type des opérandes et renvoie le type le plus grand.
Si l'on a par exemple un type char + un type int alors le type de l'opération est int (Et inversement).
SI l'on a par exemple un type int + un type long alors le type de l'opération est long (Et inversement).

## 3. Les fonctions

Pour ce rendu nous sommes aller un peu plus loin en ajoutant les fonctions. Nous avons pour l'instant seulement créer la table des fonctions et gérer leur type et les retours.

```
typedef struct fun{
      char name[MAXNAME]; /*Name*/
      int *args; /*array of args*/
      int Nargs; /*number of args*/
      int MAXargs; /*number max of args*/
      int return_type; /*type of function*/
}FUNentry;

typedef struct ftable{
      FUNentry *Ftable;
      int Fsize;
      int Fmax;
}Ftable;
```

La première structure représente les informations sur une fonction. La deuxième est une tableau avec la taille max et la taille actuelle.

On a ensuite créer la fonction pour ajouter la fonction.

```
int addFun(const char name[], int type);
```

Cette fonction renvoie 1 si l'ajout est effectué. 0 sinon.

Nous avons également écrit la fonction pour vérifier si la fonction existe ou pas.

```
int lookupFunction(const char name[]);
```

Comme pour les variables, constantes ou macros cette fonction renvoie le type de retour de la fonction ou -1 si elle n'existe pas.

Pour finir nous avons implémenté la fonction qui ajoute un argument.

```
void addArg(char name[], int type);
```

La fonction prend le nom de la fonction en paramètre et le type de l'argument à ajouté. Dans cette fonction on recherche l'indice du tableau où se trouve la fonction et on ajoute le type de l'argument dans le tableau de ses arguments.

### 4. Modification de la grammaire

Nous avons modifié la grammaire afin de pouvoir déclarer des variables et les initialiser en même temps.

```
DeclVars: DeclVars TYPE Declarateurs ';'
      |  DeclVars TYPE DeclInitVars ';'
      |  DeclVars error Declarateur ';'
      |  DeclVars TYPE Declarateur error
      |
      ;

Declarateurs: Declarateurs ',' Declarateur
      |  Declarateurs ',' DeclInitVars
      |  Declarateur
      ;

Declarateur: IDENT
      |  IDENT DeclarateurTableau
      |  error DeclarateurTableau
      ;

DeclInitVars: DeclInitVars ',' DeclInitVar
    |  DeclInitVars ',' Declarateurs
    |  DeclInitVar
    ;

DeclInitVar:
      IDENT '=' DeclInit
```

```
 ;

DeclInit: NUM
    | ADDSUB NUM
    | FLOAT
    | ADDSUB FLOAT
    | CARACTERE
    ;
```

Nous avons été obligé de “mêler” les règles pour la déclaration sans initialisation et celle avec puis comme en C en TPC on pourrait écrire :

```
Int a = 3, b, c = 2;
```

On peut ainsi mélanger déclaration avec ou sans initialisation.

Nous avons également modifié la grammaire permet de caster explicitement une variable.

```
Exp :  LValue '=' Exp
     |  LValue '=' '(' TYPE ')' Exp
     |  LValue '=' NAME_DEFINE
     |  EB
     ;
```

La troisième ligne permet aussi d’affecter à une variable une macro constante.