

Rapport de projet Sémantique et Traduction des langages

Damien Hostettler, Simon Maurel, Qi Chen et Vicky Dincher

17 Juin 2016

Table des matières

1	Construction de la table des symboles	2
1.1	Contenu et hiérarchie	2
1.2	Gestion des variables globales	3
1.3	Ajouts pour le $\mu C\#$	3
2	Préconditions et gestions des erreurs de type	3
2.1	Opérateurs et compatibilité de types	3
2.2	Types particuliers	4
3	Fonctions et leurs surcharges	4
4	Génération de code	4
4.1	Opérateurs	4
4.2	Appel de fonctions	4
5	Limitations et extensions	5
5.1	Préconditions pour le μC	5
5.2	Traitement du $\mu C\#$	5
6	Fichiers fournis	5

Introduction

Le but de ce projet a été de réaliser un compilateur pour les langages μC et $\mu C\#$. Ce compilateur doit vérifier les erreurs détectables lors de la compilation (erreurs de types, variable non définies ...) et doit générer la traduction du programme compilé en langage **TAM**.

La réalisation de ce compilateur passe par la gestion de la table des symboles, des erreurs de type, ainsi que la génération de code

1 Construction de la table des symboles

La table des symboles doit contenir toutes les informations sur ce qui est déclaré dans le programme (variables, types, fonction) sauf leur valeur en temps réel.

Une table des symboles est une liste d'élément de type INFO que l'on peut repérer par leur nom (nom de variable par exemple).

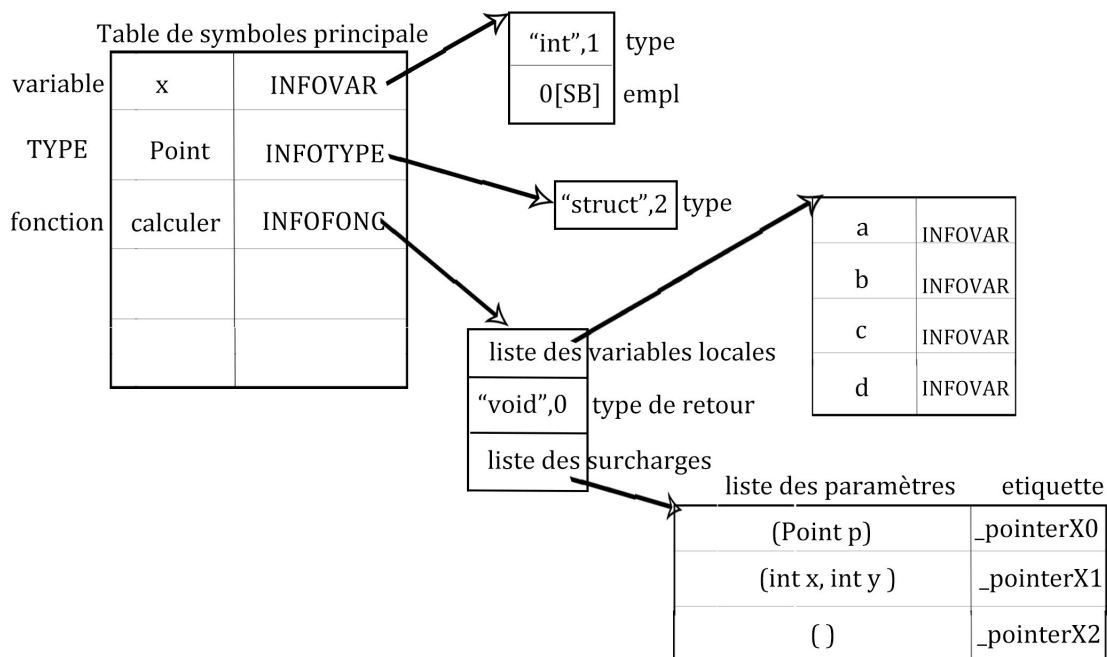
1.1 Contenu et hiérarchie

Nous avons donc modélisé notre table comme un *HashMap* $\langle String, INFOVAR \rangle$. Ce sont les différents couples (Nom des variables (fonctions ...), informations liées).

On trouve ainsi plusieurs type d'informations (toutes héritées de la classe INFO) :

- Les INFOVAR liées aux variables. Elles contiennent simplement le type de la variable, et son emplacement dans la pile.
- Les INFOTYPE liées aux types créés avec *typedef*. Elles contiennent un type (celui créé).
- Les INFOFONC, liées aux fonctions. Elles contiennent le type de retour de la fonction, la liste des différentes possibilités de paramètres pouvant être utilisés avec cette fonction (surcharges), ainsi qu'une TDS fille de la TDS courante, contenant les informations sur les variables (ou types) locales à la fonction.

On crée donc une TDS fille à chaque nouvelle fonction, mais également lorsque l'on rentre dans un nouveau bloc. On obtient ainsi la hiérarchie suivante :



1.2 Gestion des variables globales

Le compilateur implémenté permet l'utilisation de variables globales (déclarées tout au début du programme). Pour ce faire, l'emplacement des variables globales au programme (qui ne sont pas déclarées à l'intérieur d'un bloc) se situera dans le registre **SB**, et les variables locales seront déclarées dans le registre **LB**.

1.3 Ajouts pour le $\mu C\#$

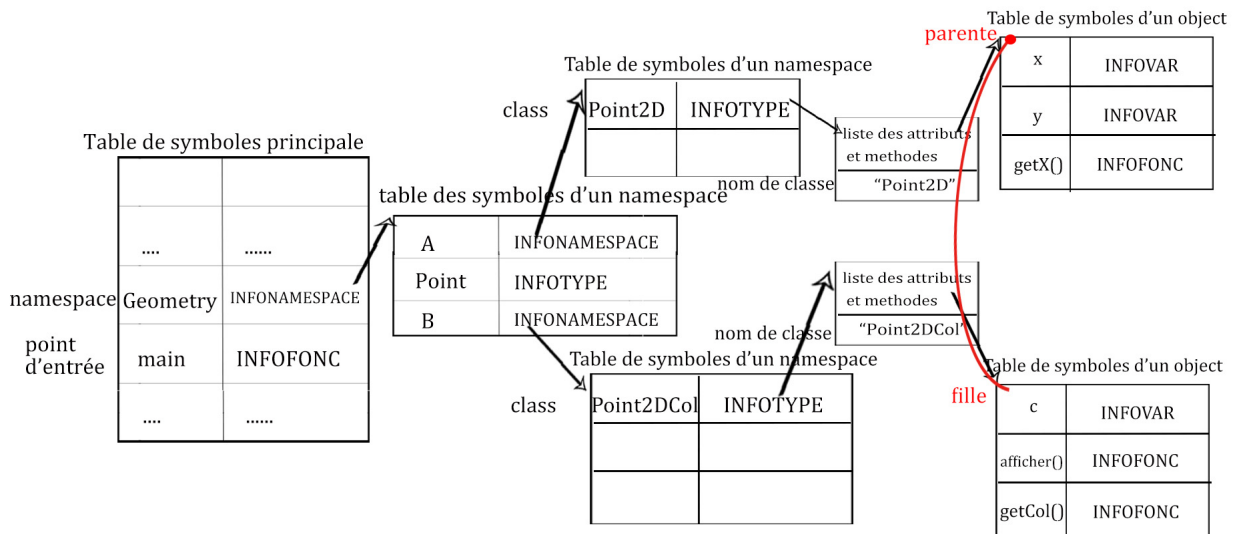
Les notions de *namespace* et de *classe* ont été ajoutées dans le langage $\mu C\#$. On aura ainsi des INFONAMESPACE qui peuvent être présents dans la TDS. Les namespaces ne sont pas gérés dans notre programme.

Afin de représenter une classe, nous avons créé le type OBJECT qui hérite de DTYPE. Ainsi, un OBJECT contient une TDS, qui est initialisée par :

- Une TDS vide si la classe déclarée est la classe mère (classe objet).
- Une TDS fille de la TDS de la classe mère, si la classe considérée est héritée.

La taille de notre classe est définie par la taille de tous ses attributs (comme dans le cas d'un STRUCT). Ainsi, on définit la taille à la fin de la définition d'une classe, avec une recherche de chaque attribut dans la TDS de la classe.

Un OBJECT possède également son héritage (de type OBJECT) qui sera null si une classe ne possède pas d'héritage.



2 Préconditions et gestions des erreurs de type

Pour la gestion des types et des erreurs de types, nous avons imposé certaines conditions, notamment pour la compatibilité des opérateurs. Dans la gestion des opérateurs, on prend en compte le type **bool**, pris en compte dans le langage $\mu C\#$.

2.1 Opérateurs et compatibilité de types

- Les opérateurs de comparaison $\leq, \geq, <, >, +$ (unaire), $-$ (unaire), $-$ (binaire), $*$, $/$, $\%$ ne sont compatibles qu'avec les types **int**.
- Les opérateurs $=$ et \neq sont compatibles avec les types **int** et **char**.
- L'opérateur $+$ est compatible avec les types **int**, **string** et **char**.
- Les opérateurs \vee, \wedge, not sont compatibles avec les opérateurs **int** et **bool**.

Ainsi, afin de pouvoir gérer les types lors des opérations sur des variables, nous avons créé le type Opérateur, qui affecte à chaque numéro d'opérateurs (numérotés de 1 à 15) une liste des noms des types admis.

On peut ainsi tester la compatibilité d'un type pour un opérateur dans la classe DTYPE. On peut donc effectuer une opération à condition que les deux opérandes aient le même type, et que ce type soit compatible avec celui de l'opérateur. Sinon, on renvoie une erreur.

2.2 Types particuliers

Les types particuliers comme les POINTEURS et les STRUCTS ont été gérés de la même manière qu'en TP.

3 Fonctions et leurs surcharges

Les fonctions ont été implémentées de manière à pouvoir les surcharger *i.e* : on peut appeler une fonction avec plusieurs listes de paramètres différentes, mais **le type de retour doit toujours rester le même**. Pour ce faire, nous avons implémenté les fonctions de la manière suivante :

Une INFOFONC contient une liste de SURCHARGES.

Une SURCHARGE est une liste d'INFOVAR ainsi que l'étiquette associée à la surcharge (ce sera dans notre cas *nom_fonctionXnum_surcharge*)

Ainsi, lors de la déclaration d'une fonction, on réalise les actions suivantes :

```
INFO i = TDS.RechercherGlobalement(nom_fonction)
Si i = null alors
    Créer une nouvelle INFOFONC, avec la surcharge correspondante aux
    paramètres déclarés associée au numéro X0
Sinon
    Vérifier que i est bien une INFOFONC
    Ajouter la surcharge des paramètres déclarés aux autres surcharges,
    associée à un nouveau numéro
Fin si
```

Ainsi, lors de l'appel, on génère une liste ordonnée de types (correspondante aux paramètres avec lesquels la fonction est appelée), et on la compare avec chacune des surcharges de la fonction (dont l'info est obtenue grâce au nom).

Si l'une des surcharges correspond, on appelle la fonction avec cette surcharge.

4 Génération de code

La génération de code a été gérée de la même façon qu'en TP, mais quelques ajouts ont été faits.

4.1 Opérateurs

Afin de réaliser les différentes opérations, nous avons créé la fonction *GenSubr* dans l'interface machine. Cette fonction prend un opérateur, et le type sur lequel il est appliqué, et envoie le code généré.

Par exemple, pour l'addition de deux entiers, le résultat sera "SUBR IAdd ".

4.2 Appel de fonctions

Lorsqu'une fonction est appelée, on obtient l'étiquette correspondante à la fonction à appeler en comparant les paramètres d'appels, et les paramètres déclarés, comme expliqué précédemment. L'étiquette est donc unique pour chaque surcharge de la fonction, et on peut ainsi générer le code de l'appel à la fonction avec la méthode *genCall*.

5 Limitations et extensions

5.1 Préconditions pour le μC

Nous avons défini plusieurs préconditions dans le cadre de la compilation du μC :

- Les variables globales doivent être déclarées avant les fonctions.
- Une fonction peut être surchargée, mais doit toujours avoir le même type de retour.
- Une fonction ou un type doivent être déclarés avant d'être utilisés.
- On peut utiliser la récursivité des fonctions mais aucun message d'erreur ne sera affiché en cas d'une boucle infinie.
- Il faut implanter une fonction **main** sans argument avec un type de retour **void** . Cette fonction sera lancée en lançant le programme. Si l'utilisateur désire surcharger la fonction *main*, il faudra déclarer le *main* sans paramètre avant d'éventuelles autres fonction *main* (cela sera donc inutile).
- A la fin de l'exécution du programme (en TAM), seules les variables globales et les variables pointées resteront sauvegardées dans la pile, le reste sera effacé.

5.2 Traitement du $\mu C\#$

Nous avons longuement réfléchi à l'implémentation du $\mu C\#$, et nous avons mis en place une stratégie pour gérer l'aspect TDS lié à ce langage, mais nous avons éprouvé quelques difficultés à établir une stratégie fonctionnelle.

Par conséquent, le compilateur implémenté n'est pas opérationnel avec un programme écrit en $\mu C\#$. Le fichier *egg* ne contient pas d'erreurs, et le compilateur fonctionne sur un programme classique, mais le code TAM généré n'est pas correcte.

De plus, nous avons fixé les préconditions suivantes :

- Le constructeur doit retourner l'objet courant (**this**) pour obtenir l'objet créé.
- Lors de l'appel d'une méthode sur un objet, celui sur lequel on veut l'appliquer doit être le premier paramètre de la fonction (mais pas lors de la déclaration de cette fonction).

Ces difficultés sont liées à la différence notable entre le langage impératif μC et le langage objet $\mu C\#$, car beaucoup de chose étant théoriquement impossibles en objet (par exemple, déclarer une variable globale dans une classe) sont possibles en impératif.

Cela implique de devoir changer le traitement de certaines règles de productions définies lors de la définition du μC .

Si nous avions disposé de plus de temps, nous aurions instancié une classe comme un **POINTEUR** sur un **STRUCT**, et non comme un **STRUCT** (stratégie actuelle). Cela permettrait de pouvoir modifier un objet en le mettant simplement en paramètre.

6 Fichiers fournis

Notre projet contient les deux fichiers *.egg* suivants :

- Le fichier *MCS_local.egg* dans le dossier *BackUp* est propre au μC .
- Le fichier *MCS.egg* dans le dossier *src.mcs* est le fichier contenant les ajouts faits pour le $\mu C\#$.

Remarque : En théorie, le fichier *MCS.egg* peut être utilisé pour les codes de tests en μC (dans le dossier *tests*), mais dans le doute, nous vous avons fourni le fichier *MCS_local.egg* dans le cas où l'une de nos modifications aurait pu entraîner des erreurs de compilation au niveau du μC .

Conclusion

Ce projet nous a permis de nous familiariser avec la compilation de programme. Nous avons ainsi pu mieux appréhender la manière dont on peut compiler un programme (gestion de la table des symboles, erreurs de types), mais aussi la traduction d'un langage.

Ce projet a également mis en valeur la gestion de la mémoire lors de l'exécution d'un programme, ainsi que les différences intrinsèques entre les langages de type impératif et objet. Ainsi, cela pourra nous permettre de mieux optimiser nos futurs programmes, en prenant en compte l'allocation mémoire et la gestion de la pile.