



---

# Projet Compilation -Cube-

---

2ème Année Cycle Supérieur  
(2CS) 2023-2024

Option : Systèmes Informatiques et Logiciels ( SIL )

## Livrable Final

Analyse sémantique

### Réalisé par

- MEDJAHDJI Islem
- MOUSSAOUI AbdElmouncif
- HABOUCHE khaled Abderrahmène
- KENNICHE AbdErrazak

### Encadré par

MR ABDMEZIEM Mohamed  
Riyadh

Groupe:

SIL1

---

# Table de matiere

---

<b>Table de matiere</b>	<b>1</b>
<b>Chapitre 1</b>	<b>2</b>
Introduction	2
1. Rappel des phases précédents	2
1.1. Définition du langage	2
1.1.1. Les commentaires	2
1.1.2. Les declarations	2
1.1.6. Les instructions de base	3
1.1.8. Les regles d'associativite	4
1.2. L'analyse lexicale	4
1.3. L'analyse syntaxique	5
2. Approche adoptée pour l'analyse sémantique	6
3. La structure de quadruplets	6
4. L'implémentation du quadruplet	7
<b>Chapitre 2</b>	<b>8</b>
1. Injection des routines sémantiques associées aux règles dans le fichier BISON	8
1.1. Les routines sémantiques pour les condition If-Else	8
1.2. Les routines sémantiques pour le boucle for	9
2. L'utilisation de la table des symboles	9
2.1. La compatibilité des types de variables	10
2.2. L'appel à une variable inexistante	10
2.3. L'affectation à une constante	10
2.4. Programme de test	10
• Résultat du programme de test	12
<b>Chapitre 3</b>	<b>13</b>
Conclusion	13

# Chapitre 1

---

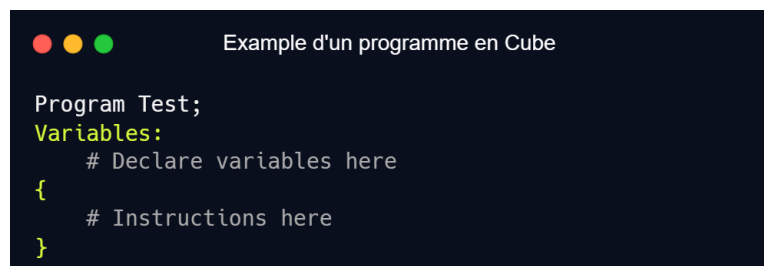
## Introduction

Après avoir accompli les étapes d'analyse lexicale et syntaxique, nous entamons maintenant la phase centrale de l'analyse sémantique dans le cadre de notre projet de compilation. Alors, nous allons implémenter une grammaire sémantique pour vérifier la cohérence sémantique de notre programme source avec le langage. Notre objectif principal est de collecter des informations sur les types et de les enregistrer dans la table des symboles en vue de les utiliser ultérieurement lors de la génération du code intermédiaire.

Pour ce faire, nous allons aborder deux aspects cruciaux de l'analyse sémantique : la vérification des types, où le compilateur s'assure que chaque opérateur a des opérandes correspondantes, et la création de la liste des quadruplets qui représente le programme en entrée. Dans ce rapport, nous détaillerons l'approche que nous avons adoptée pour développer cette analyse sémantique, ainsi que les résultats obtenus et les défis rencontrés lors de sa mise en œuvre. Cette étape revêt une importance cruciale pour assurer la qualité et la fiabilité de notre programme source, ainsi que pour garantir une génération de code adéquate.

## 1. Rappel des phases précédents

### 1.1. Définition du langage

A screenshot of a code editor with a dark background. At the top, there are three colored circles (red, yellow, green) and the text "Exemple d'un programme en Cube". The code is as follows:

```
Program Test;  
Variables:  
    # Declare variables here  
{  
    # Instructions here  
}
```

#### 1.1.1 Commentaires

```
# This is a comment in Cube
```

## 1.1.2 Declarations

### 1.1.2.1 Les types

Type	Description	Plages de valeurs
Integer	Représente les nombres entiers.	[ -2147483648 , 2147483647 ]
Real	Représente les nombres réels.	[ -3.4*10 <sup>-38</sup> , 3.4*10 <sup>38</sup> ]
Bool	Représente les valeurs de vérité	True ou False
Char	Représente un seul caractère.	Un caractère ASCII valide
Text	Représente une chaîne de caractères	

### 1.1.2.2 Declaration des variables simple :

Format	Exemple
<b>TYPE</b> variable_name ;	Integer amount;

## 1.1.3 Les instructions de base :

### 1.1.3.1 Affectation :

Format	Exemple
variable_name = value ;	x = 10;

### 1.1.3.2 Condition :

Format	Exemple
If ( condition ) { # instructions if true } else { # instructions if false }	If ( A > 10 ){ A = A + 1; } else { A = A - 1; }

### 1.1.3.3 Boucles :

Format	Example
<pre>While ( condition ) {     # instructions to repeat }</pre>	<pre>While( i &lt; 10 ){     Output(i); }</pre>

### 1.1.3.4 Entree/Sorties :

Instruction	Example
<pre>Input(var_name);</pre>	<pre>Input(x);</pre>
<pre>Output(value);</pre>	<pre>Output("Hello World!");</pre>

## 1.1.4 Les règles d'associativité:

Ce passage explique les priorités et les règles d'associativité dans l'évaluation des expressions mathématiques ou logiques.

- Parenthèses** : Les opérations à l'intérieur des parenthèses ont la priorité la plus élevée et sont évaluées en premier.
- Not** : L'opérateur logique "not" (négation logique) est appliqué ensuite.
- And** : L'opérateur logique "and" (conjonction logique) a une priorité inférieure à "not" et est évalué après.
- Or** : L'opérateur logique "or" (disjonction logique) est appliqué après "and".
- Multiplication, Division, Modulo** : Ces opérations ont une priorité supérieure à l'addition et à la soustraction.
- Addition, Soustraction** : Les opérations d'addition et de soustraction ont une priorité inférieure à la multiplication, à la division et au modulo.
- Comparaisons** : Les opérations de comparaison (comme <, >, <=, >=, ==, <>) sont effectuées en dernier, ayant la priorité la plus basse.

En respectant ces règles, l'ordre d'exécution des opérations est déterminé dans une expression complexe, assurant une évaluation correcte et cohérente.

## 1.2. L'analyse lexicale:

On a utilisé les outils **Flex**

- Les variables globales : yytext , yyleng
- Les fonctions : yyterminate , yywrap , yyerror , yymore

Au cours de l'analyse **syntaxique**, une modification significative a été apportée en choisissant de ne pas intégrer les tableaux et les enregistrements.

À partir de cette étape, nous avons obtenu les tokens nécessaires pour les prochaines phases (syntaxique et sémantique) et procédé à l'extraction de code. De plus, en, nous avons mis en place la détection des cas particuliers. En conséquence, notre programme d'analyse sera en mesure de déterminer les identifiants corrects et d'assurer une précision accrue des délimiteurs et des mots clés, qui constituent des éléments cruciaux dans l'analyse lexicale.

## 1.3. L'analyse syntaxique

Dans cette étape d'analyse syntaxique on a étudié la syntaxe du programme et vérifier la grammaire , afin d'assurer qu'il respecte les règles de construction d'un langage de programmation.

De plus , on a réalisé la table de symbole , en déclarant sa structure et les différentes fonctions et procédures pour sa gestion ainsi que leur implémentation que vous trouverez dans le fichier "symboles\_table.c" .

D'un autre côté, on a précisé la grammaire de notre langage avec les différentes sections de déclarations et règles de productions ainsi que les règles d'associativité et de priorité , suivi d'un programme de test .

Finalement on a traité les cas d'erreur qui peuvent être générés , et le format du message d'erreur est le suivant :

```
abdou@abdou-VirtualBox:~/Downloads/cube/cube-main$ ./script.sh input.cube
File 'input.cube', line 11, character 6 : syntax error, unexpected IDENTIFIER
```

tel que : Les erreurs qui peuvent être détectées dans cette phase:

- Des erreurs à cause de non respect de grammaire ou syntaxe
- De type délimiteurs: les fin d'instruction
- Les séparateurs manquant ou au plus
- Manque des mots clé comme ou des parties de définitions comme l'absence d'une condition d'une boucle, ou manque des paramètres après un séparateur dans une signature d'une méthode

et vers la fin vous trouverez des exemples de programmes de tests fonctionnels et des tests contenant des erreurs .

## 2. Approche adoptée pour l'analyse sémantique

L'approche que nous allons utiliser dans cette partie du projet pour la génération du code source intermédiaire est celle de quadruplets qui a été détaillée en cours. Elle consiste à générer une liste de quadruplets à partir des informations recueillies lors de l'analyse lexicale, syntaxique et sémantique. Les quadruplets sont des instructions formées par un opérateur, un opérande gauche, un opérande droit et un résultat.

Cette approche présente deux avantages principaux :

- Elle permet de vérifier la cohérence sémantique du programme source, en s'assurant que les opérateurs ont les bons types d'opérandes et que les instructions sont sensées.
- Elle facilite les étapes ultérieures de la compilation, car les informations nécessaires à la génération de code sont déjà incluses dans les quadruplets.

Pour générer les quadruplets, on utilise les informations recueillies lors de l'analyse syntaxique pour identifier les opérations à effectuer et les opérandes correspondants. On utilise également les informations de la table des symboles pour vérifier les types des opérandes et pour générer les quadruplets correspondants. Une fois que les quadruplets sont générés, ils peuvent être utilisés pour générer du code intermédiaire ou du code machine.

## 3. La structure de quadruplets

Un quadruplet est représenté par un enregistrement de 5 champs comme celui:

```
typedef struct Quadruplet
{
    char op[10];
    char arg1[256];
    char arg2[256];
    char result[256];
    int num;
    struct Quadruplet *next;
} Quadruplet;
```

Pour la gestion des quadruplets, on a opté pour l'utilisation d'une liste linéaire chaînée qui représente la table des symboles.

Grâce à la table des symboles, le compilateur peut vérifier l'existence d'une variable ou la correspondance des types entre les opérateurs. Ces vérifications sont effectuées lors de la

phase d'analyse sémantique.

## 4. L'implémentation du quadruplet

```
typedef struct Quadruplet
{ ...
} Quadruplet;
Quadruplet *create_quadruplet(char *op, char *arg1, char *arg2, char *result)
{ ...
}
Quadruplet *insert_quadruplet(Quadruplet **q, char *op, char *arg1, char *arg2, char *result)
{ ...
}
void update_quadruplet_result(Quadruplet *q, char *result)
{ ...
}
void update_quadruplet_arg1(Quadruplet *q, char *arg1)
{ ...
}
void update_quadruplet_arg2(Quadruplet *q, char *arg2)
{ ...
}
Quadruplet *getLastQuad(Quadruplet *q)
{ ...
}
void print_quadruplets(Quadruplet *q)
{ ...
}
void free_quadruplets(Quadruplet *q)
{ ...
}
void save_quadruplets(Quadruplet *q, char *filename)
{ ...
}
```

- et on a utiliser une pile pour la gestion des adresses retours en cas des blocs imbriqués



```

typedef struct Stack
{
    Quadruplet *q;
    struct Stack *next;
} Stack;

Stack *create_stack(Quadruplet *q)
{
    ...
}

void push(Stack **s, Quadruplet *q)
{
    ...
}

Quadruplet *pop(Stack **s)
{
    ...
}

```

## Chapitre 2

---

### 1. Injection des routines sémantiques associées aux règles dans le fichier BISON

Dans Bison, les routines sémantiques associées aux règles sont des fonctions ou des sections de code qui sont exécutées lorsque les règles de grammaire correspondent à une entrée.

Pour injecter les routines sémantiques associées aux règles dans un fichier Bison, on a utilisé les actions sémantiques. Les actions sémantiques sont des sections de code pour les inclure directement dans les règles de grammaire.

#### 1.1. Les routines sémantiques pour les condition If-Else

Les premières routines sémantiques qu'on a choisies pour générer des quadruplets sont les routines des instructions "if-else" car les conditions sont nécessaires pour ces instructions. Les quadruplets peuvent être utilisés pour stocker les informations nécessaires pour exécuter ces instructions.

- Voici un aperçu de ce type de routine

```
condition: BeginCondition OPEN_CURLY_BRACE Body CLOSE_CURLY_BRACE ElseCondition
BeginCondition: IF OPEN_PARENTHESIS expression CLOSE_PARENTHESIS {
    if(strcmp($3.type,"Bool")==0){
        Quadruplet *newQuad = insert_quadruplet(&Quad, "BZ", "", $3.value, "");
        push(&stack, newQuad);
    }else{
        printf("File '%s', line %d: Type mismatch \n", file, yylineno);
        YYERROR;
    }
}
ElseCondition: {
    Quadruplet *poppedQuad = pop(&stack);
    Quadruplet *lastQuad = getLastQuad(Quad);
    char *num = (char*)malloc(sizeof(char)*10);
    sprintf(num, "%d", lastQuad->num + 1);
    update_quadruplet_arg1(poppedQuad, num);
} | BeginElse Body CLOSE_CURLY_BRACE {
    Quadruplet *poppedQuad = pop(&stack);
    Quadruplet *lastQuad = getLastQuad(Quad);
    char *num = (char*)malloc(sizeof(char)*10);
    sprintf(num, "%d", lastQuad->num + 1);
    update_quadruplet_arg1(poppedQuad, num);
}
BeginElse: ELSE OPEN_CURLY_BRACE {
    Quadruplet *poppedQuad = pop(&stack);
    Quadruplet *newQuad = insert_quadruplet(&Quad, "BR", "", "", "");
    char *num = (char*)malloc(sizeof(char)*10);
    sprintf(num, "%d", newQuad->num + 1);
    update_quadruplet_arg1(poppedQuad, num);
    push(&stack, newQuad);
}
```

## 1.2. Les routines sémantiques pour le boucle while

On a également utilisé des routines sémantiques pour générer des quadruplets pour les instructions de boucle "while" car ces instructions nécessitent également des conditions pour fonctionner.

- Voici un exemple de comment générer des quadruplets pour une instruction de boucle "while" en utilisant des routines sémantiques:

```
While: BeginWhile Body CLOSE_CURLY_BRACE {
    // ICI on est apres le bloc d'instructions du while
    Quadruplet *beginWhileQuad = pop(&stack); // c'est l'adr de debut while car c'est la derniere
    Quadruplet *conditionQuad = pop(&stack); // l'adr de condition

    char*num = (char*)malloc(sizeof(char)*10);
    sprintf(num, "%d", conditionQuad->num + 1);
    // on insert un quadruplet pour se brancher vers la condition du while
    Quadruplet *newQuad = insert_quadruplet(&Quad, "BR", num, "", "");

    char*num2 = (char*)malloc(sizeof(char)*10);
    sprintf(num2, "%d", newQuad->num + 1);
    // mis à jour de l'adr du branchement vers la fin (le prochain bloc d'instructions) crée dans debut while
    update_quadruplet_arg1(beginWhileQuad, num2);
}
BeginWhile: WhileCondition expression CLOSE_PARENTHESIS OPEN_CURLY_BRACE {
    // ICI on est apres la condition du while
    if(strcmp($2.type, "Bool")==0){
        Quadruplet *newQuad = insert_quadruplet(&Quad, "BZ", "END_WHILE", $2.value, "");
        // on sauvegarde l'adr de la condition pour pouvoir se brancher vers elle apres le bloc d'instructions
        push(&stack, newQuad);
    }else{
        printf("File '%s', line %d: Type mismatch \n", file, yylineno);
        YYERROR;
    }
}
WhileCondition: WHILE OPEN_PARENTHESIS {
    // ICI on est avant la condition du while
    push(&stack, getLastQuad(Quad)); // sauvegarde le quad du debut de la condition
}
%%
```

## 2. L'utilisation de la table des symboles

Pendant cette étape, nous allons principalement conserver la même structure de la table des symboles précédemment définie pour effectuer les vérifications nécessaires, telles que la compatibilité des types de variables, l'appel à une variable qui n'existe pas.

## 2.1. La compatibilité des types de variables

La compatibilité des types de variables lors de la compilation est un processus qui vérifie que les variables utilisées dans un programme sont de types appropriés pour les opérations effectuées sur elles. Les erreurs de compatibilité de type peuvent également se produire lors de l'affectation de variables, de la déclaration de fonctions et de la définition de structures de données. Les compilateurs effectuent généralement ces vérifications pour garantir que le code est valide avant de le compiler en code machine exécutable, et c'est exactement ce qu'on a fait durant cette phase , voici un aperçu de cette vérification qu'on effectue ,en utilisant les différentes méthodes de la table de symbole :

```
abdou@abdou-VirtualBox:~/Downloads/cube/cube-main$ ./script.sh input.cube
File 'input.cube', line 11: Type mismatch
```

## 2.2. L'appel à une variable inexistante

L'appel à une variable inexistante lors de la compilation est une erreur qui se produit lorsqu'un programme tente d'utiliser une variable qui n'a pas été déclarée. La table des symboles est utilisée pour stocker des informations sur les variables utilisées dans un programme, comme leur nom (**nameToken** de la structure **Column**), leur type (**typeToken** ). Lors de la compilation, le compilateur parcourt le code source et utilise la table des symboles pour vérifier que toutes les variables utilisées dans le programme ont été déclarées et initialisées correctement sinon une erreur sera déclenchée comme suit :

```
abdou@abdou-VirtualBox:~/Downloads/cube/cube-main$ ./script.sh input.cube
File 'input.cube', line 12: x Variable not declared
```

### 2.3. Programme de test

Maintenant on va présenter un simple programme de test qu'on réaliser pour tester la fiabilité de différents vérification qu'on traitées

```
program cube;
Variables:
i : Integer;
r : Real;
b : Bool;
t : Text;
c : Char;
{

    i = 7;
    r = 5.13;

    while(i > r) {
        Output("Test while loop 1");
        while(b) {
            Output("Inside while loop 2");
            if(b) {
                Output("Inside If");
                i = 5;
            }
            else {
                Output("Inside Else");
                r = 3.14;
            }
        }
    }

    Output("End of program");
}
```

- **Résultat du programme de test**

```
abdou@abdou-VirtualBox:~/Downloads/cube/cube-main$ ./script.sh input.cube
File 'input.cube' compiled successfully
Quadruplets saved in 'quadruplets.txt'
Symbols table saved in 'symbols_table.txt'
```

- **Table des symboles**

1	typeToken : Integer - nameToken : i
2	typeToken : Real - nameToken : r
3	typeToken : Bool - nameToken : b
4	typeToken : Text - nameToken : t
5	typeToken : Char - nameToken : c

- **Les quadruplets**

```
quadruplets.txt
1 - (START , , , )
2 - (:= , 7 , , i)
3 - (:= , 5.130000 , , r)
4 - (> , i , r , T1)
5 - (BZ , 17 , T1 , )
6 - (output , "Test while loop 1" , , )
7 - (BZ , 16 , b , )
8 - (output , "Inside while loop 2" , , )
9 - (BZ , 13 , b , )
10 - (:= , 5 , , i)
11 - (output , "Inside If" , , )
12 - (BR , 15 , , )
13 - (output , "Inside Else" , , )
14 - (:= , 3.140000 , , r)
15 - (BR , 7 , , )
16 - (BR , 4 , , )
17 - (output , "End of program" , , )
18 - (END , , , )
```

## Chapitre 3

---

### Conclusion

En conclusion, Nous avons dû élaborer notre propre langage, en prenant en considération les exigences spécifiques de notre application, puis mettre en place une analyse lexicale, syntaxique et sémantique pour assurer la qualité du code généré.

En fin de compte, nous avons réussi à créer un exécutable capable de générer du code intermédiaire. Ce projet nous a offert une meilleure compréhension des différentes phases de la compilation et de la complexité inhérente à la création d'un compilateur.

En résumé, ce projet de compilation nous a offert l'opportunité d'approfondir notre compréhension des concepts théoriques liés à la compilation et de réaliser l'importance de chaque étape du processus de compilation pour produire un code de qualité et sécurisé.