**RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE**

**Ministère de l’Enseignement Supérieur et de la Recherche Scientifique**

**École nationale supérieure d'informatique (ESI ex. INI)**



**Projet compilation**

**2ème Année Cycle Supérieur (2CS)**

**2024-2025**

**Option : Systèmes Informatiques et Logiciels ( SIL )**

| **Livrable 04/Partie 02: Analyse Syntaxique, Généralisation avec Bison** |
| --- |

**Réalisé par**

* [**Mahdia Toubal**](mailto:km_toubal@esi.dz)
* [**Dina Keddour**](mailto:kd_keddour@esi.dz)
* [**LOUNI IMENE**](mailto:li_louni@esi.dz)
* [**Guitoun Djihen**](mailto:d_guitoun@esi.dz)

**Groupe: SIL2**

**Année : 2024/2025**

**Table de matieres**

[1. Introduction 3](#_9gnnoaq80s5s)

[2. Objectifs spécifiques de l’analyse syntaxique 3](#_a7ux7ykdg7mh)

[3. Détails d’Implémentation 4](#_jn884xpgabra)

[4. Structure du Fichier Bison 5](#_lcms1daiqno2)

[4.1 Déclarations 5](#_596wxd7qkj6y)

[4.1.1 Interface avec Flex et définition des tokens (Terminaux) 5](#_wevmprx1tx8n)

[4.1.2. Gestion de la priorité et de l’associativité 6](#_dclsv8hjoqd1)

[5 Vue d’ensemble des règles de production 6](#_voo5vq5vj7lt)

[5.1 Déclaration de fonctions et procédures : FuncsDeclar et FuncDeclar 7](#_rp7umuxq1khk)

[5.2 Règle Inst : le cœur des instructions 8](#_sxcxq5380o75)

[5.3 Exemples de boucles 8](#_7jxtp175az8)

[5.4 Structures if/else 9](#_12atjcyium0a)

[5.5 Conditions et expressions 9](#_r5sahkp8vs75)

[6. Gestion des erreurs et messages explicites 9](#_va73c39y8ovv)

[7. Résultats et observations 10](#_aksaor3mu4zw)

[8.Prochaines étapes 10](#_kqw3yqkvbea)

# 1. Introduction

La construction de l’analyseur syntaxique est une phase essentielle dans la construction d’un compilateur. Son rôle principal est de valider la structure syntaxique des programmes sources en se basant sur une grammaire définie. Dans le cadre de notre projet de compilation nous avons conçu un mini-langage appelé Flamingo. Après avoir terminé l’analyse lexicale avec Flex, nous abordons la phase d’analyse syntaxique à l’aide de Bison (Yacc). Le but de ce document est de présenter la structure de notre analyseur syntaxique, de justifier nos choix de conception.

Nous avons structuré le rapport de manière à mettre en avant le travail pratique réalisé plutôt que de reprendre en détail les concepts théoriques (LALR(1), les définitions.. etc.). Nous espérons ainsi fournir une vision claire et concrète de l’état d’avancement de notre compilateur Flamingo.

# 2. Objectifs spécifiques de l’analyse syntaxique

Le module d’analyse syntaxique que nous avons développé doit :

* Reconnaître la structure d’un programme Flamingo valide, incluant :
  + L’utilisation optionnelle de pack pour importer des bibliothèques ou ressources.
  + La déclaration de multiples fonctions (ou procédures).
  + La présence obligatoire d’un bloc Main(Hello\_Flamingo\_Developer) { … }.
* Garantir la cohérence des instructions : déclarations, affectations, appels de fonctions, etc.
* Gérer la priorité des opérateurs arithmétiques (+, -, \*, //, %, ^), logiques (&&, ||, !) et relationnels (<, <=, ==, !=, etc.).
* Proposer un mécanisme d’erreur permettant de signaler la ligne et le caractère où se situe une éventuelle erreur de syntaxe.

En remplissant ces objectifs, nous nous assurons que chaque programme saisissable en Flamingo, avant d’atteindre les phases sémantiques et de génération de code, respecte bien les structures syntaxiques de base.

# 3. Détails d’Implémentation

* Organisation des fichiers :
  + **Flamingo.l** (analyseur lexical Flex)
  + **Flamingo.y** (analyseur syntaxique Bison)
  + **.c/.h** pour la table des symboles, la gestion des quadruples futurs, etc.
* Processus de compilation :
  + **flex Flamingo.l -> lex.yy.c**
  + **bison -d Flamingo.y -> Flamingo.tab.c / Flamingo.tab.h**
  + Compilation finale avec **gcc.**
* Structures de données : Pour représenter la structure hiérarchique du programme, nous avons conçu un **arbre d’analyse syntaxique** (AST). Comme un exemple on prend une partie de l’arbre sur le contenu de la fonction principale Main:
* ProgrammeFlamingo
* └── CorMain
* ├── MAIN (Hello\_Flamingo\_Developer)
* ├── ACCOLADEOUVRANTE
* ├── aumoinsInst
* │ ├── Inst (Declar : int x)
* │ ├── Inst (Affec : x = 10)
* │ ├── Inst (CorWhile)
* │ │ ├── WHILE
* │ │ ├── Condition (x > 0)
* │ │ ├── REPEAT
* │ │ ├── ACCOLADEOUVRANTE
* │ │ │ └── aumoinsInst
* │ │ │ └── Inst (Affec : x = x - 1)
* │ │ └── ACCOLADEFERMANTE
* │ ├── Inst (CorIf)
* │ │ ├── IF
* │ │ ├── Condition (x == 0)
* │ │ ├── ACCOLADEOUVRANTE
* │ │ │ └── aumoinsInst
* │ │ │ └── Inst (WriteMethod : write(x))
* │ │ └── ACCOLADEFERMANTE
* └── ACCOLADEFERMANTE

# 4. Structure du Fichier Bison

Sachant qu’un fichier Bison est divisé en trois sections :

1. Déclarations : Définition des tokens, règles de précédence, et types de données.
2. Règles de Grammaire : Description des relations entre terminaux et non-terminaux.
3. Code C : Fonctions auxiliaires pour la gestion des erreurs et l’intégration.

## 4.1 Déclarations

### 4.1.1 Interface avec Flex et définition des tokens (Terminaux)

Notre fichier Flex (Flamingo.l) produit des tokens à partir du code source. Chaque token est déclaré dans Bison par %token. Nous avons :

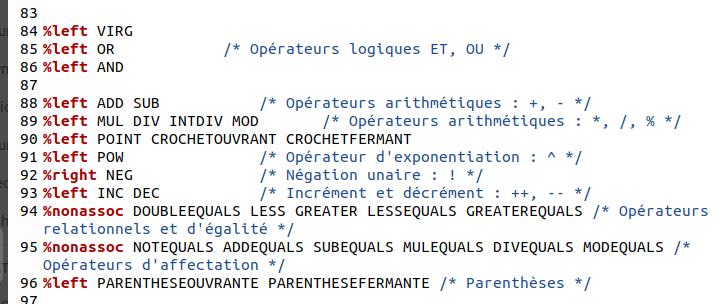
* Des mots-clés tels que IF, ELSE, WHILE, FOR, FROM, TO, READ, WRITE, etc.
* Des types de base INTTYPE, FLOATTYPE, BOOLTYPE, STRINGTYPE.
* Des opérateurs (ADD, SUB, MUL, DIV, MOD, POW, etc.) et symboles (PARENTHESEOUVRANTE, ENDINST, etc.).

Lorsqu’un lexème correspond à un mot-clé, Flex invoque return IF; (par exemple), et ainsi de suite. Pour les nombres, Flex retourne INT ou FLOAT selon qu’il détecte un point décimal. De même pour les booléens (TRUE/FALSE) qui renvoient BOOL.

Nous avons fait attention à l’ordre des règles dans le fichier .l pour éviter que des patterns génériques n' interceptent des patterns plus spécifiques.

### 4.1.2. Gestion de la priorité et de l’associativité

Pour résoudre les conflits entre opérateurs et imposer un ordre logique, nous avons défini la précédence via des directives Bison, par exemple :



Ce dispositif nous a permis d’éliminer la plupart des conflits shift/reduce. Si des ambiguïtés persistaient, nous avons ajusté la grammaire ou affiné la précédente.

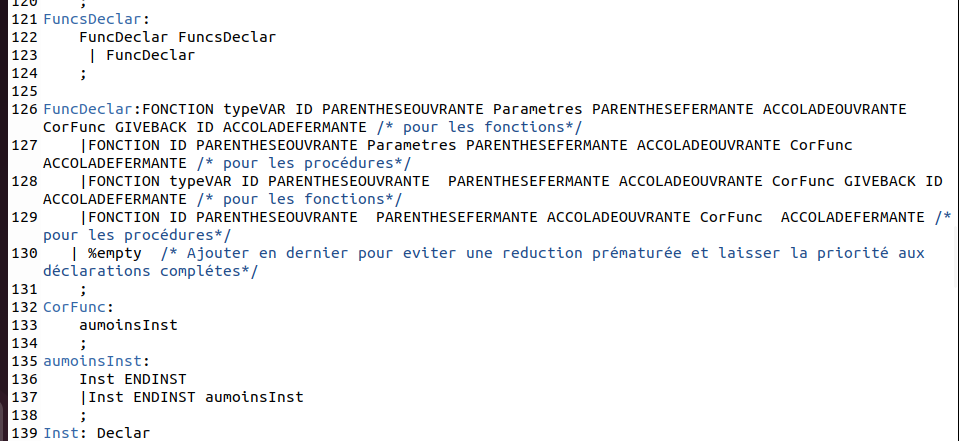
# 5 Vue d’ensemble des règles de production

Dans un fichier Bison, la section des règles de production (après les directives %token, %start, et les déclarations de précédence) décrit la structure syntaxique de notre langage. C’est ici que vous définissez les non-terminaux et la façon dont ils s’enchaînent pour former un programme valide.

Nous avons plusieurs non-terminaux majeurs :

1. **ProgrammeFlamingo :** règle de plus haut niveau, décrivant la forme complète d’un programme. C’est Le point d’entrée de l’analyse (spécifié par %start ProgrammeFlamingo).
2. **Importationlib** : gestion des pack "nomBibliothèque" ;; multiples ou absents.
3. **FuncsDeclar et FuncDeclar :** ensemble de règles dédiées à la déclaration de fonctions et de procédures.
4. **CorMain :** le bloc principal Main(Hello\_Flamingo\_Developer) { ... }.
5. **CorFunc** : Contenu principal d’une fonction ou procédure, c’est-à-dire l’enchaînement d’instructions.
6. **Inst :** instructions génériques (déclaration, affectation, structure if, while, for, etc.).
7. **Expression, Condition, et règles associées :** prise en charge de toutes les combinaisons arithmétiques et booléennes.

### 5.1 Déclaration de fonctions et procédures : FuncsDeclar et FuncDeclar



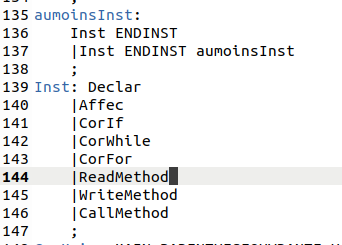
Ici, FuncsDeclar exprime la possibilité d’avoir plusieurs déclarations de fonction successives :

* FuncDeclar FuncsDeclar : une déclaration suivie d’autres déclarations,
* FuncDeclar : une seule déclaration (condition d’arrêt de la récursion).
* typeVAR ID : si on mentionne un type (ex. int, float), la fonction est censée retourner une valeur via GIVEBACK.
* Parametres : liste des paramètres (par ex. int x, float y).
* CorFunc : bloc d’instructions interne, commun entre fonction et procédure
* GIVEBACK ID : indique la variable (ou l’identifiant) renvoyée par la fonction.
* La factorisation de la grammaire se voit dans la réutilisation de CorFunc : que ce soit une fonction ou une procédure, on emploie le même bloc d’instructions. CorFunc renvoie simplement vers aumoinsInst, qui exprime “au moins une instruction” ou plusieurs.
* Inst ENDINST : une instruction suivie de ;; (ou autre symbole que vous utilisez pour terminer les instructions).
* Inst ENDINST aumoinsInst : une instruction + ;; + la possibilité d’enchaîner d’autres instructions.

De cette manière, que nous soyons dans le corps d’une fonction, d’une procédure ou même dans Main, on réutilise les mêmes règles pour “enchaîner” des instructions.

### 5.2 Règle Inst : le cœur des instructions

Le non-terminal Inst regroupe toutes les formes d’instructions que le langage *Flamingo* reconnaît :



* Declar : déclarations de variables, constantes, tableaux, structures simples.
* Affec : affectation (ex. x = y, tableau[i] = 3, etc.).
* CorIf : branchement conditionnel if/else.
* CorWhile : boucle while.
* CorFor : boucle for.
* ReadMethod / WriteMethod : opérations d’entrée/sortie (lire ou afficher).
* CallMethod : appel de fonction.

Toutes ces règles sont développées ensuite de manière plus spécifique (ex. CorIf pour if/else, CorWhile pour while, etc.).

### 5.3 Exemples de boucles

Boucle while

* La production CorWhile définit la syntaxe while (condition) repeat { ... }.
* On y trouve un non-terminal Condition (expression booléenne) et un bloc d’instructions CorFunc.

Boucle for

* La production CorFor gère la syntaxe for (id from borne\_debut to borne\_fin) { ... }, avec différentes variantes (les bornes peuvent être soit des entiers, soit des identifiants).
* Le bloc répété est également assuré par CorFunc.

### 5.4 Structures if/else

* CorIf et Alternate résolvent le problème du “dangling else” en distinguant clairement les cas if, if-else et else if.
* CorIf définit le if (...) { ... }; Alternate prend en charge else { ... } ou else if.

### 5.5 Conditions et expressions

* La règle Condition impose souvent l’usage de parenthèses pour les expressions logiques (opérateurs &&, ||, négation !).
* La règle expression définit la hiérarchie des opérations arithmétiques, tandis que terme gère les littéraux, identifiants et appels de fonction.

# 6. Gestion des erreurs et messages explicites

Afin d’améliorer l’expérience de débogage :

Nous conservons les numéros de ligne via **yylineno** et l’index **termeCourant.**

Dans Flex, toute séquence non reconnue déclenche un message d’erreur lexical et l’affichage de la ligne incriminée avec un caret (^) qui pointe la position du caractère illégal.

Dans Bison, la fonction yyerror(const char \*s) affiche un message sous la forme :

**File "input.txt", line 4, character 56: syntax error** ou tout autre texte explicite.

Nous avons choisi, pour l’instant, de **terminer** l’analyse **dès la première erreur rencontrée.** Néanmoins, il serait possible d’implémenter des règles de récupération d’erreurs (error recovery) afin de détecter plusieurs erreurs au fil de l’analyse.

# 7. Résultats et observations

Taux de réussite : Sur l’ensemble de nos tests positifs (programmes corrects), la compilation aboutit sans erreur.

Ergonomie : Les messages d’erreur syntaxique sont explicites, indiquant la ligne et le caractère fautif. Cela simplifie la correction pour l’utilisateur.

Performance : Le parseur s’exécute rapidement, même sur des programmes plus longs. Bison génère un code assez efficace.

Lisibilité de la grammaire : Nous avons structuré les règles pour qu’elles restent faciles à maintenir et à modifier.

En résumé, nous constatons que l’analyse syntaxique couvre la plupart des cas d’utilisation courants et gère correctement les combinaisons d’instructions et d’expressions.

**Note: Au cours de l’implémentation, nous avons rencontré plusieurs ambiguïtés qui nous ont conduits à modifier légèrement la déclaration du langage. Par exemple, nous avons supprimé certains opérateurs binaires jugés redondants, ajouté un mot-clé repeat pour clarifier la syntaxe de la boucle while et imposé un endinst (;;) après les structures while, for, if ou toute autre instruction. Ces ajustements se sont avérés nécessaires pour résoudre des conflits de grammaire et rendre le langage plus cohérent et compréhensible.**

# 8.Prochaines étapes

**Analyse sémantique :** vérification des types, du nombre et type des paramètres de fonctions, portée des variables.

**Génération de code intermédiaire (quadruples) :** pour traduire chaque instruction Flamingo en une suite d’opérations plus élémentaires, indépendantes de l’architecture machine.

En conclusion, notre analyseur syntaxique fonctionne désormais de manière satisfaisante et constitue une base solide pour la phase suivante du développement du compilateur Flamingo.