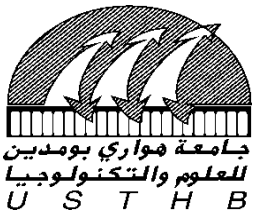
REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université des Sciences et de la Technologie Houari Boumediène



Faculté d'Informatique

Rapport de projet de Compilation

Filière : « Security Informatique »

Effectué à

Mme BELHADI.H

Sous le thème

Réalisation d'un mini compilateur

Réalisé par

G3 ATMANE Chakib 222231497907

G1 AMLIK Salim 212233230712

G1 LABRAOUI Aymen 222231410117

G3 LAIFAOUI Abderraouf 222231361814

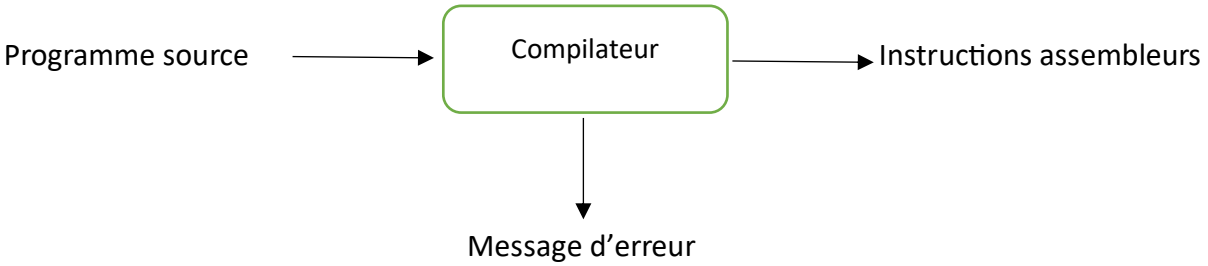
G3 SOULMIYA Ilyes 222235059211

Année Universitaire : 2024/2025

Thème : Réalisation d'un mini compilateur

Introduction :

Un compilateur est un logiciel particulier qui traduit un programme dans un langage de haut niveau en instructions exécutables. C'est donc l'instrument fondamental à la base de toute réalisation informatique.

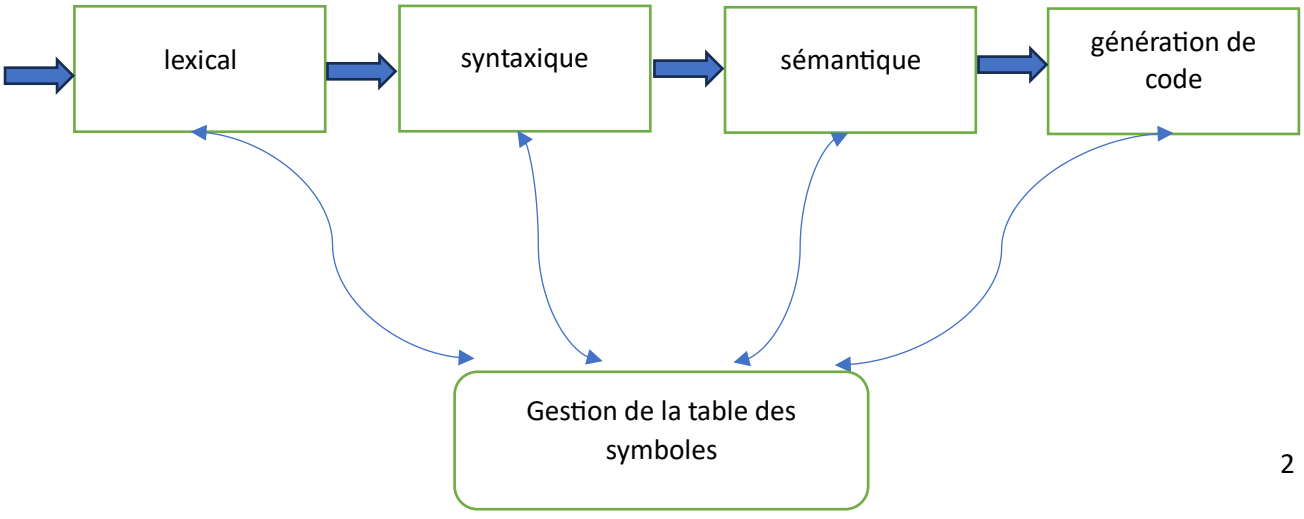


L'objectif de ce Projet est de réaliser un système informatique complet. On a réaliser un compilateur qui traduit un langage source en un langage cible : jeu d'instructions assembleur orientées registre

Un compilateur travaille généralement en plusieurs étapes successives, appelées les **phases de compilation qui sont** :

- Phase d'analyse:
    1. Analyse lexicale
    2. Analyse syntaxique
    3. Analyse sémantique
- Phase de production:(génération du code intermédiaire et optimisation)
- Une phase parallèle qui concerne la gestion des tables des symboles et la gestion des erreurs.

Notre but était de réaliser un mini compilateur qui



Outils et Technologies :

Analyse lexicale :

Pour la réalisation de l'analyseur lexical on a choisi **Flex** (fast lexical analyser generator) l'outil Flex est un générateur d'analyseurs lexicaux. Il accepte en entrée des unités lexicales sous formes d'expressions régulières et produit un programme écrit en langage C qui, une fois compilé, reconnaît ces unités lexicales. L'exécutable obtenu lit le texte d'entrée caractère par caractère jusqu'à l'identification de plus long préfixe du texte source qui concorde avec l'une des expressions régulières.

Après l'installation de Flex on a commencé à lire la documentation de cet outil a fin de savoir sa structure, sa syntaxe..., et au mem temp à pratiquer un petit peu sur des exercice simple par exemple " Écrire un programme Flex permettant de dire si une chaîne en entrée est un nom ", ou on a trouvé que nous devons faire un petit rappelle sur les expressions régulières ce qui était régler juste avec un coup d'œil sur le chapitre de l'année passé.

Exemples : (des expressions réguliers utilisé dans notre mini projet)

[0-9]+ expression régulière décrit un entier non signé

[0-9]* "."[0-9]+

[A-Z][a-z0-9]{0,7} expression régulière décrit un identificateur

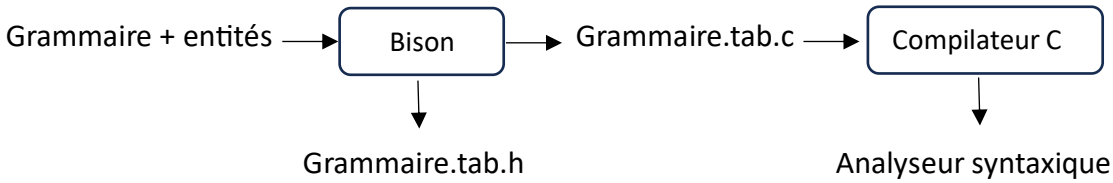
Les éléments lexicaux sont reconnus :

- Les identifiants (qui commence par une lettre ou un tiret du bas)
- Les entiers
- Les booléens "true" et "false"
- Les chaînes de caractères (entre guillemets " ")
- Les commentaires (entre //)
- On a la possibilité aussi d'analyser les mot clés suivant : "VAR_GLOBAL", "DECLARATION", "INSTRUCTION", "TYPE", " INTEGER", " FLOAT", " CHAR", "CONST", "IF","ELSE","FOR","READ","WRITE" .
- Le point virgule : ";" .
- Tout ce qui est calcul : "+", "-", "*", "/" .
- L'affectation avec : "=" .
- Operateurs binaires de comparaison : "!=" ,"==" ,"<=" ,">=" .
- Opérateurs logiques : "&&" ,"||" ,"!" .
- Les séparateurs (parenthèse et accolade) : "{", "}" ,"(" , ")" .
- Incrémentation : "++".
- Décrémentation : "--".

Analyse syntaxique :

Dans un deuxième temps on va réaliser l'analyseur syntaxique, Le rôle de l'analyseur syntaxique est de vérifier la syntaxe du programme source. Il reçoit une suite d'unités lexicales fournie par l'analyseur lexical et doit vérifier que cette suite peut être engendrée par la grammaire du langage, pour cette tache on va utiliser **Bison**.

Bison (version GNU de YACC) est un générateur d'analyseurs syntaxiques. Il accepte en entrée la description d'un langage sous forme d'une grammaire et produit un programme écriten C qui, une fois compilé, reconnaît les mots appartenant au langage engendré par la grammaire en entrée.



Il faut savoir que l'outil Bison permet de générer des analyseurs syntaxique avec des grammaire LALR(1)

L'entré de Bison est un fichier source avec extension. y qui contient trois section principales:

Déclaration : dans cette section on a défini les symboles utilisés dans la grammaire, les

types de données et les options de Bison

Règles de grammaire : la grammaire elle même est définie dans cette section c'est la que on

Spécifie les règles de production de notre grammaire qui reconnaitre notre langage spécifique

Action C : pour chaque règle de production on a associé un bloc de code c qui sera exécuté lors que cette règle est appliquée .

.

Coordination entre Flex et Bison :

- La fonction yylex()

La fonction **yylex()** est une composante centrale de l'analyse lexicale dans un compilateur construit avec des outils comme **Lex** ou **Flex** cette fonction est responsable de lire le flux d'entrée (souvent du code source) et de le découper en unités lexicales ou tokens. Ces tokens sont ensuite transmis à l'analyseur syntaxique (généré par Bison) pour la phase d'analyse syntaxique.

- La variable yylval :

Cette variable est l'une des outils de base de communication entre Flex et Bison au cours de l'analyse. Donc on a pensé à affecter correctement **yylval** lors de l'analyse lexical

exemple :

```
"\""[^\"]*"\""        { yylval.sval = strdup(yytext); return STRING_LITERAL; }
```

Analyse sémantique :

L'analyse sémantique en liaison étroite avec l'analyse syntaxique permet de donner un sens à ce qui a été reconnu dans cette phase. Elle permet par exemple de

- ✓ Vérifier si un identificateur a été bien déclarer.
- ✓ La compatibilité de types dans une expression.
- ✓ Utilisation correcte dune étiquette.
- ✓ Vérification des structures complexes.

L'outil Bison permet de réaliser l'analyse sémantique en parallèle avec l'analyse syntaxique à laide de routine sémantique appelée à chaque réduction d'un terminal.

Revenons à notre mini compilateur ou on a :

- La fonction semantic_analysis() :

Cette fonction valide et gère la déclaration des variables et des constantes. Elle garantit que :

- Les symboles (variables ou constantes) ne sont pas redéfinis.
- Le type déclaré et les propriétés (par exemple, constante ou variable) sont correctement enregistrés dans la table des symboles.

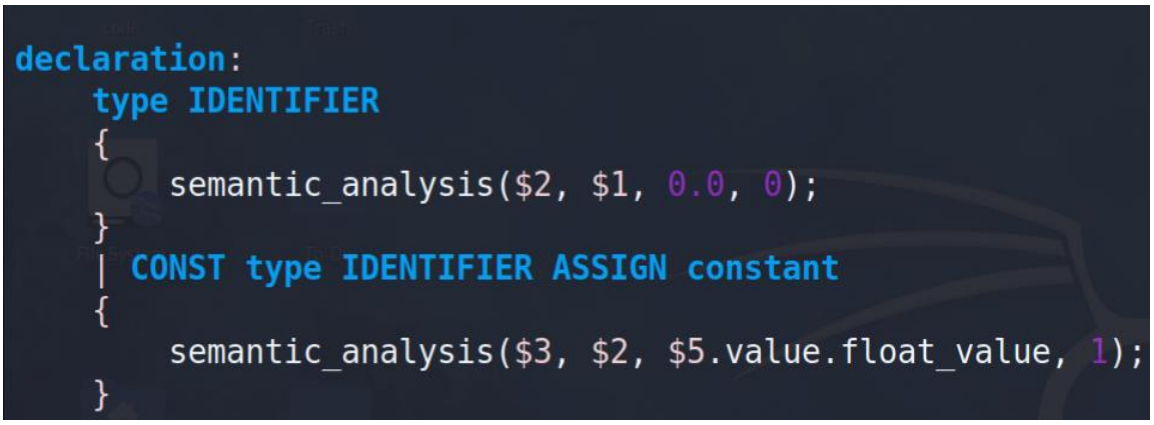
Fonctionnement :

Vérification de la redéfinition :

- Si le symbole existe déjà, la fonction vérifie s'il y a un conflit (par exemple, redéfinir une constante comme variable ou inversement).
- En cas de conflit, elle affiche une erreur et arrête l'exécution.

Insertion du symbole :

- Si le symbole est une constante, la fonction tente de l'insérer dans la table des symboles via insert_constant.
- Si le symbole est une variable, il est inséré via insert_symbol.
- Si l'insertion échoue, la fonction signale une erreur et arrête l'exécution.



```
declaration:
    type IDENTIFIER
    {
        semantic_analysis($2, $1, 0.0, 0);
    }
    | CONST type IDENTIFIER ASSIGN constant
    {
        semantic_analysis($3, $2, $5.value.float_value, 1);
    }
```

- La fonction analyze_assignment:

Cette fonction valide et traite les affectations aux variables. Elle garantit que :

- La variable à laquelle une valeur est assignée est déclarée.
- Les constantes ne sont pas réaffectées.
- Le type de la valeur assignée correspond au type de la variable.

Fonctionnement :

Vérification de l'existence de la variable :

- La fonction recherche la variable dans la table des symboles. Si elle n'est pas trouvée, une erreur est signalée.

Vérification de l'affectation à une constante :

- Si la variable est une constante, la fonction s'assure qu'aucune affectation ne lui est faite. Sinon, une erreur est signalée.

Vérification de l'incompatibilité de types :

- La fonction compare le type de la valeur assignée avec le type de la variable. S'ils ne correspondent pas, une erreur est signalée.

Affectation de la valeur :

- Si toutes les vérifications sont réussies, la valeur est assignée à la variable dans la table des symboles.

```
instruction:
    IDENTIFIER ASSIGN expression
    {
        analyze_assignment($1, $3.type, $3.value.float_value);
    }
```

La table des symboles :

Les déférentes entités définies dans le programme sont stockées sans la table symbole

Dans Flex et Bison la table des symboles doit être programmée manuellement

C'est une table de structure bien définie contient des informations sur les entités

Program <nom_program>

Integer x ;

Const Char s;

Debut

X:=2*x

Fin

Nom entité	Code entité	Type	Constante (1/0)	Affectation (1/0)
nom program	idf	String	0	
x	idf	Integer	0	
s	idf	Char	1	

Dans notre travaille il existe deux fonction globale qui gère cette table :

- Iitialze_symbol_table(): Réinitialiser la table des symboles en la vidant (grâce à memset())

```
void initialize_symbol_table() {
    symbol_count = 0;
    memset(symbol_table, 0, sizeof(symbol_table));
}
```

- Insert_symbol(): Elle vérifie si un symbole portant le même nom existe déjà dans la table. Si c'est le cas, elle affiche un message d'erreur et retourne un échec, si le symbole valide et n'existe pas encore ,il est ajouté à la table à l'index symbol_count
- Insert_constant() : Ajoute une constante(variable avec valeur fixe)à la table des symboles.

```c
int insert_constant(const char *name, int type, float value) {
    if (!insert_symbol(name, type)) {
        return 0; // Return failure
    }

    symbol_table[symbol_count - 1].is_constant = 1;

    if (type == TYPE_INT) {
        symbol_table[symbol_count - 1].value.int_value = (int)value;
    } else if (type == TYPE_FLOAT) {
        symbol_table[symbol_count - 1].value.float_value = value;
    }

    return 1; // Return success
}
```

Génération de code intermédiaire :

La génération de code intermédiaire peut être considérée comme le point ou on passé de la partie analyse à la partie synthèse.

Les quadruplés :

Les quadruplets constituent une forme de code intermédiaire dans le processus de compilation. Ils permettent de représenter les opérations arithmétiques, logiques, ou d'affectation sous une forme standardisée, indépendante de la machine cible. Cette représentation simplifie les optimisations et la traduction vers le code machine final.

Notre grammaire appelle des fonctions pour des produire quadruplets ,ces fonctions clés incluent:

- generate_assgment() : Crée un quadruplet pour une affectation .
- evaluate_condition() : Gere les comparaisons logiques.

```c
if (!check_type($1, $4)) {
    yyerror("Type mismatch in assignment.");
} else {
    // Generate an assignment based on the type of the expression
    if ($4.type == TYPE_INT) {
        Value val = { .type = TYPE_INT, .value.intval = $4.value.intval };
        generate_assignment($2, val);
    } else if ($4.type == TYPE_FLOAT) {
        Value val = { .type = TYPE_INT, .value.intval = (int)$4.value.floatval };
        generate_assignment($2, val); // Cast float to int
    }
}
```

Conclusion :

La realisation d'un mini-compilateur nous a permis d'explorer les composantes essentielles du processus de compilation, allant de l'analyse lexicale et syntaxique à la validation sémantique et la génération de code intermédiaire. Grâce à l'utilisation d'outils tels que Flex et Bison, nous avons développé une chaîne d'analyse capable de reconnaître un langage source simplifié et de vérifier sa validité.

Ce projet a permis d'atteindre les objectifs suivants :

- **Analyse lexicale** : Mise en œuvre avec Flex pour identifier et segmenter les éléments clés tels que les identificateurs, les mots-clés et les opérateurs.
- **Analyse syntaxique** : Développée avec Bison pour garantir que l'entrée respecte la grammaire du langage, détectant ainsi les erreurs structurelles.
- **Analyse sémantique** : Intégration de routines pour valider les déclarations des identificateurs, la compatibilité des types et l'utilisation correcte des constructions du programme.
- **Génération de code intermédiaire** : Conception de quadruplets standardisés représentant les opérations et affectations dans un format indépendant de la machine.

Ce projet nous a offert une compréhension approfondie du fonctionnement des compilateurs, tout en soulignant l'importance de la coordination entre les différentes phases du processus. Bien que l'implémentation soit axée sur un compilateur basique, elle constitue une base solide pour explorer des fonctionnalités avancées telles que l'optimisation de code ou la génération complète de code machine.

En somme, ce travail nous a permis d’acquérir des compétences pratiques et de consolider nos connaissances théoriques en conception de compilateurs.