الجـمهوريـة الجزائريـة الديمـقراطيـة الشـعبيـة
ⵜⴻⵣⴰⵢⵉⵔ ⵜⴰⵣⵣⴰⵢⵔⵜ ⵜⴰⵎⴰⵡⴰⵙⵜ ⵜⴰⵖⵔⴰⵎⵜ
République Algérienne Démocratique et Populaire
وزارة التعليـم العـالي والبـحث العلمـي
ⴰⵢⵎⴻⵔ ⵏ ⵜⵙⴻⵏⵜⵉⵏ ⵜⵉⵏⴻⵔ ⴷ ⵜⵓⴷⴻ ⵜⵙⵙⵓⵏ
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE

المدرسة الوطنية العليا للإعلام الآلي
ⵜⴰⵙⴷⴰⵡⵜ ⵜⴰⵎⵙⵙⵓⵔⵜ ⵜⵉⵏⴻⵔ ⵏ ⵜⵓⵙⵙⵏⵉⵡⵉⵏ
École nationale Supérieure d'Informatique

2ème année Cycle Supérieur (2CS)
Option : Systèmes Informatiques et Logiciels (SIL1)
Equipe : 06

Thème :

PROJET DE COMPILATION

Rapport final

Réalisé par :
- Bounab Chaima
- Djerfi Fatma
- Mekircha Rafika Houda
- Melliti Abdelmalek

Encadré par :
M. ABDMEZIEM Riyadh

Promotion 2025-2026

Table des matières

Liste des tableaux

Chapitre

1 Introduction

Au cur de l'informatique moderne, la compilation représente le pont indispensable entre l'abstraction des langages de programmation de haut niveau et la rigueur de l'exécution machine. Concevoir un compilateur ne se limite pas à une simple traduction de code ; c'est un processus complexe de transformation qui nécessite une compréhension profonde des structures linguistiques, de la logique formelle et de la gestion optimisée des ressources mémoires.

Dans le cadre de la formation d'ingénieur au sein de l'École Supérieure d'Informatique (ESI), ce mini-projet de compilation propose de mettre en pratique ces concepts théoriques à travers la réalisation d'un compilateur complet pour un langage personnalisé. L'objectif est de maîtriser chaque étape de la chaîne de compilation, depuis la reconnaissance des unités lexicales jusqu'à la génération d'un code intermédiaire sous forme de quadruplets.

Ce projet se distingue par une double approche : d'une part, l'implémentation manuelle d'algorithmes fondamentaux (automates et tables d'analyse) pour en saisir la mécanique interne, et d'autre part, l'utilisation d'outils industriels standards tels que **FLEX** et **BISON** pour traiter la grammaire globale du langage.

Afin de détailler la conception et la réalisation de ce système, le présent rapport s'articule autour des axes suivants :

— **Le premier chapitre** présente la **définition du langage cible**. Il décrit la syntaxe générale, les types de données (simples et structurés), les instructions de contrôle (boucles, conditions) ainsi que les règles de priorité et d'associativité des opérateurs qui régissent le langage.

— **Le deuxième chapitre** est dédié à **l'analyse lexicale et syntaxique**. Il détaille la conception des automates finis déterministes (AFD) et des tables d'analyse. Ce chapitre explique comment l'outil *FLEX* est utilisé pour la segmentation en unités lexicales et comment *BISON* assure l'analyse syntaxique ascendante (LALR) pour construire l'arbre syntaxique du programme.

— **Le troisième chapitre** traite de **l'analyse sémantique et de la gestion des erreurs**. Il expose la structure de la **Table des Symboles**, essentielle pour la vérification des types et la portée des variables. Ce chapitre présente également les routines sémantiques et le mécanisme de signalement des erreurs (lexicales, syntaxiques et sémantiques) permettant une localisation précise dans le code source.

— **Le quatrième chapitre** porte sur la **génération du code intermédiaire**. Il décrit la transformation de l'arbre syntaxique en une suite de **quadruplets**, offrant une représentation abstraite et efficace du programme, prête pour d'éventuelles optimisations futures.

En conclusion, ce travail constitue une synthèse des techniques de traitement des langages, démontrant la capacité à transformer une grammaire théorique en un outil logiciel fonctionnel et robuste.

Chapitre

2 Présentation du langage QueryLang

2.1 Introduction

QueryLang est un langage impératif simple, conçu pour allier la puissance des langages procéduraux classiques à la lisibilité intuitive de la syntaxe SQL. Sa structure est pensée pour être proche du langage naturel, facilitant ainsi le développement et la maintenance du code source.

2.2 Structure et Syntaxe de base

Un programme QueryLang est strictement délimité par des balises de début et de fin. La structure suit une logique séquentielle rigoureuse.

Élément	Syntaxe / Exemple
Début du programme	`BEGIN PROGRAM [Nom_Programme]`
Fin du programme	`END PROGRAM;`
Commentaire simple	`- Texte du commentaire`
Commentaire multi-ligne	`/* Texte sur plusieurs lignes */`

TABLE 2.1 – Structure générale et commentaires

2.3 Système de Typage

QueryLang propose des types de données variés pour répondre aux besoins de calcul et de structuration de l'information.

2.3.1 Types de variables simples

Type	Syntaxe Déclaration	Description
INTEGER	`SET x INTEGER = 10;`	Nombres entiers
STRING	`SET s STRING = 'Texte';`	Chaînes de caractères
FLOAT	`SET f FLOAT = 12.5;`	Nombres réels
BOOLEAN	`SET b BOOLEAN = true;`	Valeurs logiques

TABLE 2.2 – Types de données primitifs

2.3.2 Structures de données complexes

Structure	Syntaxe d'exemple
Enregistrement	`CREATE RECORD Personne (nom STRING, age INTEGER);`
Tableau	`SET notes ARRAY[INTEGER, 5] = {12, 15, 10, 8, 18};`
Dictionnaire	`SET dict DICTIONARY<STRING, INTEGER>;`

TABLE 2.3 – Variables structurées

2.4 Opérateurs du Langage

2.4.1 Opérateurs Arithmétiques et de Comparaison

Op.	Arithmétique	Op.	Comparaison
+	Addition	=	Égalité
-	Soustraction	<>	Différent
*	Multiplication	>	Supérieur strict
/	Division	<	Inférieur strict
%	Modulo	>= / <=	Bornes (Sup/Inf ou égal)

TABLE 2.4 – Opérateurs arithmétiques et relationnels

2.4.2 Opérateurs Logiques

Opérateur	Fonction Logique
`AND`	Et logique (conjonction)
`OR`	Ou logique (disjonction)
`NOT`	Négation (inversement)

TABLE 2.5 – Opérateurs logiques

2.5 Priorités et Associativité

Ordre	Catégorie d'opérateurs	Associativité
1	Parenthèses `()`	Interne
2	`NOT`	Droite → Gauche
3	Multiplication, Division, Modulo (`*`, `/`, `%`)	Gauche → Droite
4	Addition, Soustraction (`+`, `-`)	Gauche → Droite
5	Comparaisons (`=`, `<>`, `<`, `>`, etc.)	Gauche → Droite
6	Logiques (`AND`, `OR`)	Gauche → Droite
7	Affectation (`=`)	Droite → Gauche

TABLE 2.6 – Hiérarchie des opérations

2.6 Instructions et Structures de Contrôle

2.6.1 Affectation et Entrées/Sorties

Action	Syntaxe / Exemple
Affectation	`x = x + 1;`
Affichage	`PRINT 'Premier nombre: ';`
Lecture	`INPUT a;`

TABLE 2.7 – Instructions de base

2.6.2 Structures Conditionnelles

2.6.2.1 Condition simple (WHEN)

La structure `WHEN` permet d'exécuter un bloc de code si une condition booléenne est vérifiée.

Listing 2.1 – Exemple de condition simple

```
WHEN x > 10 THEN
    PRINT 'Grand';
OTHERWISE
    PRINT 'Petit';
END WHEN;
```

2.6.2.2 Condition multiple (CASE)

Le `CASE` évalue plusieurs branches à l'aide du mot-clé `WHENCASE`.

Listing 2.2 – Exemple de structure CASE

```
CASE
    CASEWHEN operation = '+' THEN resultat = a + b;
    CASEWHEN operation = '-' THEN resultat = a - b;
    ELSE PRINT 'Operation invalide';
END CASE;
```

2.6.3 Boucles et Itérations

2.6.3.1 Boucle WHILE

La boucle se répète tant que la condition reste vraie.

```
LOOP WHEN x < 100
    x = x + 1;
END LOOP;
```

2.6.3.2 Boucle FOR

L'itération s'effectue sur une plage de valeurs définie.

```
LOOP ITERATE i FROM 0 TO 10
    PRINT i;
END LOOP;
```

2.7 Conclusion

Ce chapitre a posé les fondements du langage **QueryLang**. Ces spécifications servent de base pour le développement de l'analyseur lexical et syntaxique détaillés dans la suite de ce rapport.

Chapitre

3 Analyse Lexicale

3.1 Introduction

L'analyse lexicale constitue la première phase de notre compilateur QueryLang. Son rôle est de transformer le flux de caractères du code source en une séquence de tokens (unités lexicales), tout en détectant les erreurs lexicales éventuelles. Nous avons adopté une approche pédagogique en deux temps : d'abord une implémentation manuelle des transformations théoriques (expressions régulières $\rightarrow$ AFN $\rightarrow$ AFD) sur un sous-ensemble du langage, puis une généralisation avec l'outil FLEX pour l'ensemble du langage.

3.2 Approche 1 : Implémentation Manuelle des Transformations

Cette première approche vise à mettre en pratique les concepts théoriques vus en cours. Nous avons sélectionné trois entités lexicales fondamentales pour illustrer le processus complet de transformation.

3.2.1 Sélection du Sous-ensemble

Pour cette implémentation, nous avons choisi de traiter :

- **Identificateurs** : Pattern `[a-zA-Z_][a-zA-Z0-9_]*`
- **Entiers** : Pattern `[0-9]+`
- **Réels** : Pattern `[0-9]+\.[0-9]+`

Ce choix n'est pas anodin. Les identificateurs combinent plusieurs constructions (union, concaténation, étoile de Kleene), les entiers illustrent l'opérateur plus, et les réels montrent la gestion de séquences complexes. Ensemble, ils couvrent les principales opérations des expressions régulières.

3.2.2 Construction de Thompson (Regex $\rightarrow$ AFN)

La construction de Thompson est un algorithme qui transforme une expression régulière en un automate fini non-déterministe (AFN) équivalent. Nous l'avons implémentée en C dans le fichier `thompson.c`.

a) Principe général

L'algorithme procède par construction récursive. Chaque opération de l'expression régulière est traduite par une construction élémentaire d'AFN, puis ces fragments sont combinés selon les règles de Thompson.

Cette fonction crée l'AFN le plus simple : deux états reliés par une transition étiquetée par le caractère `c`. C'est le cas de base de la construction récursive.

b) Plages de caractères

Pour gérer efficacement les classes de caractères comme `[a-z]`, nous avons implémenté une fonction dédiée :

```
AFN thompsonRange(char start_char, char end_char) {
    AFN afn;
    initAFN(&afn);

    int start = newState(&afn);
    int final = newState(&afn);

    afn.start_state = start;
    afn.final_states[0] = final;
    afn.final_count = 1;

    for (char c = start_char; c <= end_char; c++) {
        addTransition(&afn, start, final, c);
    }

    return afn;
}
```

Listing 3.1 – Construction pour une plage de caractères

Au lieu de créer un AFN séparé pour chaque lettre puis de les unir, cette fonction crée directement plusieurs transitions parallèles entre l'état initial et l'état final. C'est une optimisation qui simplifie la structure résultante.

c) Opérations composées

Les opérations de composition (union, concaténation, étoile) suivent les schémas classiques de Thompson :

```
AFN thompsonUnion(AFN afn1, AFN afn2) {
    AFN result;
    initAFN(&result);

    int new_start = newState(&result);
    int new_final = newState(&result);


    for (int i = 0; i < afn1.state_count; i++) {
        result.states[result.state_count++] = afn1.states[i];
    }
```

```
12
13
14
15      addTransition(&result, new_start, afn1.start_state, EPSILON);
16      addTransition(&result, new_start, afn2.start_state, EPSILON);
17
18
19      for (int i = 0; i < afn1.final_count; i++) {
20          addTransition(&result, afn1.final_states[i], new_final, EPSILON);
21      }
22
23
24      result.start_state = new_start;
25      result.final_states[0] = new_final;
26      result.final_count = 1;
27
28      return result;
29  }
```

Listing 3.2 – Union de deux AFN

La concaténation et l'étoile de Kleene suivent des principes similaires, en créant les epsilon-transitions appropriées pour relier les fragments d'AFN.

d) Exemple : Construction de l'AFN pour les identificateurs

La construction complète d'un identificateur illustre bien l'enchaînement des opérations :

3.2.2.1 Exemple : Construction de l'AFN pour les identificateurs

Code source : Construction de l'AFN pour les identificateurs

```
1  AFN letter = thompsonRange('a', 'z');
2  AFN upper = thompsonRange('A', 'Z');
3  AFN underscore = thompsonSymbol('_');
4
5  AFN first_part = thompsonUnion(letter, upper);
6  first_part = thompsonUnion(first_part, underscore);
7
8  AFN letter2 = thompsonRange('a', 'z');
9  AFN upper2 = thompsonRange('A', 'Z');
10 AFN digit = thompsonRange('0', '9');
11 AFN underscore2 = thompsonSymbol('_');
12
13 AFN second_part = thompsonUnion(letter2, upper2);
14 second_part = thompsonUnion(second_part, digit);
15 second_part = thompsonUnion(second_part, underscore2);
16 second_part = thompsonStar(second_part);
17
18 AFN final_afn = thompsonConcat(first_part, second_part);
19
20 AFD afd = subsetConstruction(&final_afn);
21
22 return afd;
```

Cette construction décompose clairement l'expression régulière en ses parties constituantes, puis les combine selon les règles de Thompson.

3.2.3 Construction par Sous-ensembles (AFN → AFD)

Une fois l'AFN construit, il faut le déterminiser pour obtenir un AFD exploitable efficacement par l'analyseur lexical. L'algorithme de construction par sous-ensembles (subset construction) réalise cette transformation.

a) Principe de l'algorithme

L'idée fondamentale est que chaque état de l'AFD correspond à un ensemble d'états de l'AFN. On part de l'epsilon-closure de l'état initial de l'AFN, puis on calcule itérativement les transitions possibles pour chaque symbole d'entrée.

b) Epsilon-closure

La fonction epsilon-closure calcule l'ensemble des états accessibles depuis un état donné en suivant uniquement des epsilon-transitions :

```
StateSet epsilonClosure(AFN* afn, StateSet states) {
    StateSet closure = states;
    bool added = true;

    while (added) {
        added = false;
        for (int i = 0; i < closure.count; i++) {
            int state = closure.states[i];

            // Chercher toutes les epsilon-transitions depuis cet état
            for (int j = 0; j < afn->trans_count; j++) {
                if (afn->transitions[j].from_state == state &&
                    afn->transitions[j].symbol == EPSILON) {

                    int target = afn->transitions[j].to_state;

                    // Ajouter l'état cible s'il n'est pas déjà présent
                    bool found = false;
                    for (int k = 0; k < closure.count; k++) {
                        if (closure.states[k] == target) {
                            found = true;
                            break;
                        }
                    }

                    if (!found && closure.count < MAX_STATES) {
                        closure.states[closure.count++] = target;
                        added = true;
                    }
                }
            }
        }
```

```
    }

    return closure;
}
```

Listing 3.3 – Calcul de l'epsilon-closure

L'algorithme itère jusqu'à ce qu'aucun nouvel état ne puisse être ajouté. C'est un point fixe classique.

c) Fonction move

La fonction move calcule l'ensemble des états accessibles depuis un ensemble d'états donné en consommant un symbole spécifique :

```
StateSet move(AFN* afn, StateSet states, int symbol) {
    StateSet result;
    result.count = 0;

    for (int i = 0; i < states.count; i++) {
        int state = states.states[i];

        for (int j = 0; j < afn->trans_count; j++) {
            if (afn->transitions[j].from_state == state &&
                afn->transitions[j].symbol == symbol) {

                int target = afn->transitions[j].to_state;

                // Éviter les doublons
                bool found = false;
                for (int k = 0; k < result.count; k++) {
                    if (result.states[k] == target) {
                        found = true;
                        break;
                    }
                }

                if (!found && result.count < MAX_STATES) {
                    result.states[result.count++] = target;
                }
            }
        }
    }

    return result;
}
```

Listing 3.4 – Fonction move pour un symbole

d) Construction complète de l'AFD

L'algorithme principal combine ces deux fonctions pour construire l'AFD :

```
AFD subsetConstruction(AFN* afn) {
    AFD afd;
```

```
    afd.state_count = 0;

    // Initialiser la table de transitions à -1
    for (int i = 0; i < MAX_STATES; i++) {
        for (int j = 0; j < 256; j++) {
            afd.transitions[i][j] = -1;
        }
        afd.final_states[i] = false;
    }

    // État initial = epsilon-closure de l'état initial de l'AFN
    StateSet initial;
    initial.count = 1;
    initial.states[0] = afn->start_state;
    initial = epsilonClosure(afn, initial);

    afd.states[afd.state_count++] = initial;
    afd.start_state = 0;

    // File d'états non traités
    int unprocessed[MAX_STATES];
    int unprocessed_count = 1;
    unprocessed[0] = 0;

    while (unprocessed_count > 0) {
        int current_idx = unprocessed[--unprocessed_count];
        StateSet current = afd.states[current_idx];

        // Pour chaque symbole possible
        for (int symbol = 0; symbol < 256; symbol++) {
            if (!isalnum(symbol) && symbol != '_' && symbol != '.') continue;

            StateSet m = move(afn, current, symbol);
            if (m.count == 0) continue;

            StateSet target = epsilonClosure(afn, m);
            if (target.count == 0) continue;

            // Vérifier si cet ensemble existe déjà
            int target_idx = findStateSet(&afd, target);

            if (target_idx == -1) {
                // Nouvel état
                target_idx = afd.state_count;
                afd.states[afd.state_count++] = target;
                unprocessed[unprocessed_count++] = target_idx;
            }

            afd.transitions[current_idx][symbol] = target_idx;
        }
    }

    // Marquer les états finaux
    for (int i = 0; i < afd.state_count; i++) {
        StateSet state = afd.states[i];
        for (int j = 0; j < state.count; j++) {
            for (int k = 0; k < afn->final_count; k++) {
                if (state.states[j] == afn->final_states[k]) {
                    afd.final_states[i] = true;
                    break;
                }
            }
        }
    }

    return afd;
}
```

Listing 3.5 – Construction par sous-ensembles

L'algorithme maintient une file d'états à traiter et explore systématiquement toutes les transitions possibles. Un état de l'AFD est marqué comme final s'il contient au moins un état final de l'AFN.

3.2.4 Utilisation des AFD pour la Reconnaissance

Une fois les AFD construits, ils sont utilisés par l'analyseur lexical pour reconnaître les tokens dans le code source :

```
Token scanWithAFD(AFD* afd, TokenType default_type) {
    Token token;
    token.line = line_num;
    token.column = col_num;
    int i = 0;

    int current_state = afd->start_state;
    int last_accept_state = -1;
    int last_accept_pos = 0;

    int start_pos = pos;

    while (peek() != '\0' && i < 255) {
        char ch = peek();
        int next_state = afd->transitions[current_state][(unsigned char)ch];

        if (next_state == -1) break;

        token.lexeme[i++] = advance();
        current_state = next_state;

        if (afd->final_states[current_state]) {
            last_accept_state = current_state;
            last_accept_pos = i;
        }
    }

    if (last_accept_state != -1) {
        token.lexeme[last_accept_pos] = '\0';
        token.type = (default_type == IDENTIFIER) ?
                     checkKeyword(token.lexeme) : default_type;

        pos = start_pos + last_accept_pos;
        return token;
    }

    token.type = ERROR_TOK;
    strcpy(token.lexeme, "ERROR");
    return token;
}
```

Listing 3.6 – Reconnaissance avec un AFD

Cette fonction implémente le principe du "longest match" : elle continue de consommer des caractères tant qu'une transition est possible, mais mémorise la dernière position où un état final a été atteint. Ainsi, si la reconnaissance échoue plus loin, on peut revenir à la dernière reconnaissance valide.

3.3 Approche 2 : Analyse Complète avec FLEX

Pour traiter l'ensemble du langage QueryLang, nous avons utilisé FLEX, un générateur d'analyseurs lexicaux qui automatise la transformation des expressions régulières en code C optimisé.

3.3.1 Structure du Fichier FLEX

Un fichier FLEX se compose de trois sections séparées par %% :

```
%{
    /* Section 1 : Code C préliminaire */
    /* Déclarations, includes, variables globales */
%}

    /* Définitions (optionnel) */
    DIGIT    [0-9]
    LETTER   [a-zA-Z]

%%
    /* Section 2 : Règles de reconnaissance */
    pattern1    { action1 }
    pattern2    { action2 }

%%
    /* Section 3 : Code C additionnel */
    /* Fonctions utilitaires, main(), etc. */
```

Listing 3.7 – Structure générale d'un fichier FLEX

3.3.2 Gestion de la Table des Symboles Enrichie

Dans cette approche FLEX, nous avons considérablement enrichi la table des symboles pour supporter l'analyse sémantique future. La structure d'un symbole contient maintenant :

```
typedef struct SymbolEntry {
    char name[256];
    char type[64];
    SymbolCategory category;    // Variable, Constante, Tableau, etc.
    int line;
    int column;
    ScopeLevel scope;
    int scope_id;

    // Informations pour les tableaux
    int is_array;
    int array_size;
    char element_type[64];

    // Informations pour les enregistrements
    int is_record;
    FieldEntry* fields;
    int field_count;

    // Informations pour les constantes
    int is_constant;
    char constant_value[256];

    // Informations de gestion mémoire
    int address;
    int size;
    int is_initialized;
    int usage_count;
    int is_used;

    struct SymbolEntry* next;
} SymbolEntry;
```

Listing 3.8 – Structure enrichie d'un symbole

Cette richesse d'information nous permet de faire des vérifications sémantiques basiques dès l'analyse lexicale, notamment la détection de double déclaration.

3.3.3 Gestion des Portées

Une des fonctionnalités importantes de notre analyseur est la gestion des portées imbriquées. QueryLang supporte les blocs de portée dans les structures de contrôle :

```
void enterScope() {
    symTable.current_scope_id++;
}

void exitScope() {
    SymbolEntry* curr = symTable.head;
    while (curr != NULL) {
        if (curr->scope_id == symTable.current_scope_id && !curr->is_used) {
            fprintf(stderr, "Warning: Variable '%s' declared but never used\n"
    ↪ ,
                    curr->name);
        }
        curr = curr->next;
    }
    symTable.current_scope_id--;
}
```

Listing 3.9 – Gestion des portées

Ces fonctions sont appelées lors de la reconnaissance de mots-clés spécifiques :

```
"WHEN"              { printToken("KW_WHEN", yytext); enterScope(); }
"END WHEN"          { printToken("KW_END_WHEN", yytext); exitScope(); }
"LOOP"              { printToken("KW_LOOP", yytext); enterScope(); }
"END LOOP"          { printToken("KW_END_LOOP", yytext); exitScope(); }
```

Listing 3.10 – Appel de gestion de portée dans FLEX

3.3.4 Détection et Gestion des Erreurs

Notre analyseur implémente un système de gestion d'erreurs complet qui permet de :

- Continuer l'analyse après une erreur (mode panic désactivable)
- Limiter le nombre d'erreurs affichées pour éviter la pollution de sortie
- Catégoriser les erreurs par type
- Fournir un contexte précis (ligne, colonne, caractères environnants)

```
typedef enum {
    ERR_LEXICAL,
    ERR_INVALID_CHAR,
    ERR_UNTERMINATED_STRING,
    ERR_UNTERMINATED_COMMENT,
    ERR_INVALID_NUMBER,
    ERR_IDENTIFIER_TOO_LONG,
    ERR_STRING_TOO_LONG
} ErrorType;
```

Listing 3.11 – Types d'erreurs lexicales

Les erreurs sont détectées soit par des règles FLEX spécifiques, soit par la règle par défaut :

```
    /* Chaîne non terminée */
'[^\n']*$           {
                        reportError(ERR_UNTERMINATED_STRING,
                                    "String not terminated before end of line",
                                    yytext);
                    }

    /* Commentaire multi-ligne non terminé */
\/\*([^*]|\*+[^*/])*$  {
                        reportError(ERR_UNTERMINATED_COMMENT,
                                    "Multi-line comment not terminated",
                                    "/* ...");
                    }

    /* Caractère invalide (règle par défaut) */
.                   {
                        char error_msg[256];
                        sprintf(error_msg, "Unexpected character '%s'", yytext
    ↪ );
                        reportError(ERR_INVALID_CHAR, error_msg, yytext);
                        col_num++;
                    }
```

Listing 3.12 – Détection d'erreurs dans FLEX

3.3.5 Règles de Reconnaissance

FLEX nous permet de définir des règles de reconnaissance très expressives. Voici quelques exemples caractéristiques :

a) Commentaires

QueryLang supporte deux types de commentaires (ligne simple et multi-ligne). Le traitement diffère légèrement :

```
COMMENT_LINE    --[^\n]*
COMMENT_MULTI   \/\*([^*]|\*+[^*/])*\*+\/

%%

{COMMENT_LINE}      { printToken("COMMENT", "-- ..."); }
{COMMENT_MULTI}     {
                        printToken("COMMENT", "/* ... */");
                        for (int i = 0; i < yyleng; i++) {
                            if (yytext[i] == '\n') {
                                line_num++;
                                col_num = 1;
                            }
                        }
```

```
15                      }
```

Listing 3.13 – Reconnaissance des commentaires

Pour les commentaires multi-lignes, on doit compter les retours à la ligne pour maintenir le suivi précis de la position.

b) Identification des mots-clés

Les mots-clés nécessitent parfois des actions spécifiques. Par exemple, pour les types de données, nous mettons à jour automatiquement la table des symboles si nous sommes dans un contexte de déclaration :

```
"INTEGER"           {
                        printToken("KW_INTEGER", yytext);
                        if (in_declaration && last_identifier[0] != '\0') {
                            insertSymbol(last_identifier, "INTEGER",
    ↪ CAT_VARIABLE);
                            last_identifier[0] = '\0';
                            in_declaration = 0;
                        }
                        strcpy(last_type, "INTEGER");
                    }
```

Listing 3.14 – Reconnaissance des types avec insertion automatique

Cette technique permet de construire la table des symboles au fur et à mesure de l'analyse, sans attendre une phase séparée.

c) Opérateurs composés

Certains opérateurs comme <= doivent être distingués de leurs préfixes <. FLEX résout automatiquement ces ambiguïtés en appliquant la règle du "longest match" :

```
"<="                { printToken("OP_LTE", yytext); }
">="                { printToken("OP_GTE", yytext); }
"<>"                { printToken("OP_NEQ", yytext); }
"<"                 { printToken("OP_LT", yytext); }
">"                 { printToken("OP_GT", yytext); }
"="                 { printToken("OP_EQ", yytext); }
```

Listing 3.15 – Opérateurs de comparaison

L'ordre des règles n'a pas d'importance ici, FLEX choisira automatiquement la règle qui consomme le plus de caractères.

d) Validation de longueur

Pour éviter les dépassements de buffer, nous validons la longueur des identificateurs et des chaînes :

```
{IDENTIFIER}        {
                        if (yyleng > 255) {
```

```
                reportError(ERR_IDENTIFIER_TOO_LONG,
                            "Identifier exceeds maximum length",
                            yytext);
            } else {
                printToken("IDENTIFIER", yytext);
                if (in_declaration) {
                    strcpy(last_identifier, yytext);
                } else {
                    markSymbolUsed(yytext);
                }
            }
        }
```

Listing 3.16 – Validation de longueur

3.3.6 Suivi de Position

Le suivi précis de la position (ligne et colonne) est important pour fournir des messages d'erreur utiles. Nous le gérons manuellement :

```
void printToken(const char* type, const char* lexeme) {
    printf("Line %3d, Col %3d: %-20s '%s'\n",
           line_num, col_num, type, lexeme);
    col_num += strlen(lexeme);
}
```

Listing 3.17 – Mise à jour de la position

```
{NEWLINE}           { line_num++; col_num = 1; }
{WHITESPACE}        { col_num += yyleng; }
```

Listing 3.18 – Gestion des retours à la ligne

3.4 Comparaison des Deux Approches

Les deux approches présentent des avantages et des limites complémentaires :

L'approche manuelle nous a permis de comprendre en détail les transformations théoriques et de manipuler concrètement les structures de données des automates. C'était particulièrement formateur pour appréhender les concepts d'epsilon-transition, de déterminisation et de minimisation.

L'approche FLEX, quant à elle, nous a montré comment les outils industriels permettent de gérer efficacement la complexité d'un langage complet. Elle nous a aussi permis d'implémenter des fonctionnalités avancées comme la gestion de portée et la détection d'erreurs sophistiquée, que nous aurions eu du mal à intégrer dans l'approche manuelle.

3.5 Résultats et Validation

Nous avons testé notre analyseur lexical sur plusieurs fichiers de test couvrant différents aspects du langage.

TABLE 3.1 – Comparaison des approches d'analyse lexicale

Critère	Approche manuelle	Approche FLEX
Valeur pédagogique	Excellente - compréhension profonde des mécanismes	Limitée - abstraction du processus
Complexité d'implémentation	Élevée - plusieurs centaines de lignes de code	Faible - règles déclaratives
Couverture du langage	Partielle - 3 types de tokens	Complète - tous les tokens
Performances	Bonnes mais non optimisées	Excellentes - code généré optimisé
Maintenabilité	Difficile - modifications nécessitent de revoir l'AFD	Facile - ajout/modification de règles simple
Taille du code	600 lignes (C pur)	400 lignes (FLEX + C)
Débogage	Complexe - nécessite de tracer l'exécution de l'AFD	Simple - règles lisibles

3.5.1 Exemple de Sortie

Pour un programme QueryLang simple :

```
BEGIN PROGRAM TestLexer;
    SET x INTEGER = 10;
    SET nom STRING = 'Alice';
    PRINT x;
END PROGRAM;
```

Listing 3.19 – Exemple de code source QueryLang

L'analyseur produit :

```
Line   1, Col   1: KW_BEGIN            'BEGIN'
Line   1, Col   7: KW_PROGRAM          'PROGRAM'
Line   1, Col  15: IDENTIFIER          'TestLexer'
Line   1, Col  24: SEP_SEMICOLON       ';'
Line   2, Col   5: KW_SET              'SET'
Line   2, Col   9: IDENTIFIER          'x'
Line   2, Col  11: KW_INTEGER          'INTEGER'
Line   2, Col  19: OP_EQ               '='
Line   2, Col  21: INT_LITERAL         '10'
Line   2, Col  23: SEP_SEMICOLON       ';'
...
```

Listing 3.20 – Sortie de l'analyseur lexical

3.5.2 Détection d'Erreurs

L'analyseur détecte correctement les erreurs lexicales :

```
SET message STRING = 'texte non terminé
SET @invalid = 10;
```

Listing 3.21 – Exemple avec erreurs

Produit :

```
File "source.ql", line 1, character 25: unterminated string:
String not terminated before end of line
File "source.ql", line 2, character 5: invalid character:
Unexpected character '@'
```

3.6 Perspectives d'Amélioration

Plusieurs améliorations pourraient être apportées :

- **Minimisation de l'AFD** : Les AFD générés ne sont pas minimisés. L'ajout de l'algorithme de Hopcroft réduirait leur taille.
- **Messages d'erreur plus informatifs** : Actuellement, nous signalons qu'un caractère est invalide, mais nous pourrions suggérer ce que l'utilisateur voulait probablement écrire.
- **Mode de récupération d'erreur** : Après une erreur, l'analyseur pourrait essayer de se resynchroniser sur le prochain token valide plutôt que de continuer aveuglément.
- **Support Unicode** : QueryLang ne gère actuellement que l'ASCII. Le support UTF-8 nécessiterait de revoir la gestion des transitions.

3.7 Conclusion

L'analyse lexicale, bien que première phase du compilateur, s'est révélée riche en apprentissages. La double approche (manuelle puis avec FLEX) nous a permis de comprendre à la fois les fondements théoriques et les contraintes pratiques du développement d'un compilateur. L'implémentation manuelle des transformations (Thompson, subset construction) nous a donné une appréciation concrète de la complexité algorithmique et des choix de structures de données. L'approche FLEX nous a montré l'importance des outils dans le développement logiciel moderne. En quelques centaines de lignes de spécification déclarative, nous avons obtenu un analyseur robuste et performant pour l'ensemble du langage. Les tokens produits par cette phase servent maintenant d'entrée à l'analyse syntaxique, qui construira l'arbre syntaxique abstrait du programme. La table des symboles initialisée ici sera enrichie au fil des phases suivantes.

Chapitre

4 Analyse Syntaxique

4.1 Introduction

L'analyse syntaxique constitue la deuxième étape majeure du processus de compilation. Son rôle est de vérifier si la suite de lexèmes (tokens) fournie par l'analyseur lexical respecte les règles grammaticales définies pour le langage **QueryLang**. Contrairement à l'analyse lexicale qui travaille de manière linéaire, l'analyse syntaxique impose une structure hiérarchique au code source, généralement représentée sous forme d'arbre syntaxique.

4.2 Méthode de travail sur BISON

L'implémentation de l'analyseur syntaxique avec **Bison** repose sur une collaboration étroite avec l'analyseur lexical (**Flex**). Notre méthodologie s'est articulée autour de quatre étapes clés :

4.2.1 Récupération des Tokens depuis Flex

La première étape consiste à établir le pont entre `Flex` et `Bison`. `Bison` définit les terminaux (*tokens*) via la directive `%token`. Ces identifiants sont ensuite partagés avec `Flex` (via le fichier d'en-tête généré) pour que le lexeur puisse renvoyer les bonnes unités lexicales lors de la lecture du code source. Chaque jeton porteur de valeur, tel qu'un nombre ou un identifiant, est associé à un type spécifique défini dans l'union `%union` afin d'assurer un transfert correct des données sémantiques.

```
%union {
    int ival;
    float fval;
    char* str;
    ASTNode* node;
    DataType dtype;
}
```

FIGURE 4.1 – Déclaration de lunion (%union) pour spécifier les différents types de données

```
%token KW_BEGIN KW_PROGRAM KW_END
%token KW_SET KW_CREATE KW_RECORD KW_ARRAY KW_DICTIONARY
%token KW_INTEGER KW_STRING KW_FLOAT KW_BOOLEAN
%token KW_TRUE KW_FALSE
%token KW_WHEN KW_CASE_WHEN KW_THEN KW_OTHERWISE KW_CASE KW_ELSE
%token KW_LOOP KW_ITERATE KW_FROM KW_TO
%token KW_FOREACH KW_IN
%token KW_PRINT KW_INPUT
%token OP_AND OP_OR OP_NOT
%token OP_EQ OP_NEQ OP_LT OP_GT OP_LTE OP_GTE
%token OP_PLUS OP_MINUS OP_MULT OP_DIV OP_MOD
%token SEP_LPAREN SEP_RPAREN SEP_LBRACKET SEP_RBRACKET
%token SEP_LBRACE SEP_RBRACE SEP_COMMA SEP_SEMICOLON SEP_DOT
```

FIGURE 4.2 – Flux de communication entre l'analyseur lexical (Flex) et l'analyseur syntaxique (Bison)

4.2.2 Définition de la Grammaire et des Règles de Production

Une fois les *tokens* déclarés, nous avons défini la grammaire hors-contexte de **QueryLang** en utilisant la notation **BNF** (*Backus-Naur Form*)[1]. Pour chaque règle, nous avons spécifié les structures syntaxiques valides :

- **La structure globale :** Le programme est rigoureusement délimité par les mots-clés `BEGIN PROGRAM` et `END PROGRAM`.
- **Les instructions :** Nous avons défini des règles pour les affectations, les boucles (`LOOP`), les conditions (`WHEN/CASE`) et les déclarations de variables.
- **Les expressions :** La gestion des priorités mathématiques a été assurée par les directives de précédence (`%left`, `%right`) pour éviter les ambiguïtés sur les opérateurs.

```
Program:
    KW_BEGIN KW_PROGRAM IDENTIFIER SEP_SEMICOLON Instrs KW_END KW_PROGRAM SEP_SEMICOLON
    {
        $$ = createProgramNode($3, NULL, $5);
        root = $$;
        printf("\n✓ Program '%s' parsed successfully!\n", $3);
        free($3);
    }
    ;


Decl:
    KW_SET IDENTIFIER BasicType OptInit SEP_SEMICOLON
    {
        $$ = createDeclNode($2, $3, $4);
        free($2);
    }
    | KW_SET IDENTIFIER IDENTIFIER OptInit SEP_SEMICOLON
    {
        $$ = createRecordInstanceNode($2, $3, $4);
        free($2); free($3);
    }
    | RecordDecl { $$ = $1; }
    | ArrayDecl  { $$ = $1; }
    | DictDecl   { $$ = $1; }
    ;
```

FIGURE 4.3 – Extrait De Grammaires et Regle de Production

1. La notation **BNF** est une syntaxe formelle permettant de décrire la structure d'un langage informatique en définissant comment les symboles de base peuvent être combinés pour former des instructions valides.

4.2.3 Actions Sémantiques et Construction de l'AST

À chaque règle de grammaire est associée une action sémantique entre accolades `{ }`. Ces actions ne se contentent pas de valider la syntaxe ; elles appellent des fonctions de création de nuds (ex : `createBinOpNode`, `createIfNode`) pour construire dynamiquement l'**Arbre de Syntaxe Abstraite (AST)**.

```
Cond:
    Cond OP_AND Cond     { $$ = createBinOpNode(AST_OP_AND, $1, $3); }
  | Cond OP_OR Cond      { $$ = createBinOpNode(AST_OP_OR, $1, $3); }
  | OP_NOT Cond          { $$ = createUnaryOpNode(AST_OP_NOT, $2); }
  | Expr OP_EQ Expr      { $$ = createBinOpNode(AST_OP_EQ, $1, $3); }
  | Expr OP_NEQ Expr     { $$ = createBinOpNode(AST_OP_NEQ, $1, $3); }
```

FIGURE 4.4 – Example d'une action sémantique

```
ASTNode* createBinOpNode(OperatorType op, ASTNode* left, ASTNode* right) {
    ASTNode* node = (ASTNode*)malloc(sizeof(ASTNode));
    node->type = NODE_EXPR_BINOP;
    node->line = line_num;
    node->column = col_num;
    node->data.binOp.op = op;
    node->data.binOp.left = left;
    node->data.binOp.right = right;
    return node;
}
```

FIGURE 4.5 – Example d'une action sémantique sur AST

Cet arbre représente la structure logique du programme et servira de base aux étapes ultérieures d'analyse sémantique et de génération de code.

```
PROGRAMME: TestSimple
├─ Déclarations:
  (nul)
└─ Instructions:
    SI
  ├─ Condition:
    OP_BINAIRE: >
    ├─ Gauche:
      IDENTIFIANT: x
    └─ Droite:
      LITTÉRALE: 5 (ENTIER)
  ├─ Alors:
        IMPRESSION
    └─ Expression:
      LITTÉRALE: 'Greater' (CHAÎNE)
    IMPRESSION
  └─ Expression:
    IDENTIFIANT: x
    DÉCL: x : ENTIER
  └─ Initialiseur:
    LITTÉRALE: 10 (ENTIER)

✓ Compilation successful!
```

FIGURE 4.6 – Example de L'Arbre de Syntaxe Abstraite

4.2.4 Gestion des Conflits et Erreurs

Durant le développement, nous avons utilisé les rapports de `Bison` pour identifier et résoudre les conflits de type **"Décalage/Réduction"** (*Shift/Reduce*), notamment dans les structures de contrôle imbriquées. La fonction `yyerror` a également été personnalisée pour fournir des messages d'erreur précis, incluant le numéro de ligne et de colonne, garantissant ainsi un débogage efficace en cas de syntaxe invalide.

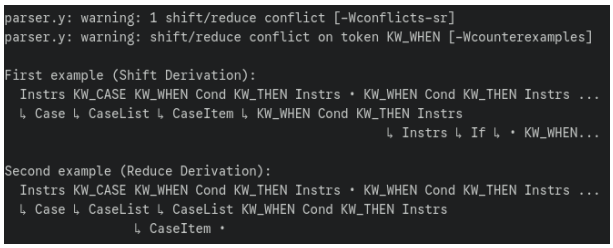

FIGURE 4.7 – Example d'un error Syntaxique

4.3 Table de symboles enrichie

4.3.1 Rôle et Importance

La table des symboles est une structure de données centrale du compilateur *QueryLang*. Son rôle principal est de stocker toutes les informations relatives aux identificateurs rencontrés lors de l'analyse du code source. Contrairement à une table de symboles basique, notre version est dite **enrichie** car elle gère :

- **La Portée (Scope) :** Distinction entre variables globales et locales (imbrication de blocs).
- **Le Typage complexe :** Gestion des types primitifs, mais aussi des structures (*Record*), tableaux et dictionnaires.
- **Le suivi d'état :** Vérification de l'initialisation des variables pour prévenir les erreurs d'exécution.
- **La gestion mémoire :** Attribution d'adresses relatives pour la future génération de code.

4.3.2 Mécanisme d'Obtention et Mise à Jour

La table est construite de manière incrémentale à travers les phases suivantes :

1. **Analyse Lexicale :** Chaque identificateur est détecté et passé à l'analyseur syntaxique.
2. **Analyse Syntaxique (Bison) :** Lors de la détection d'une déclaration (mot-clé `SET` ou `CREATE`), le compilateur appelle la fonction `insererSymbole()`.
3. **Gestion des blocs :** À chaque entrée dans une structure de contrôle (`WHEN`, `LOOP`), le niveau de portée est incrémenté. À la sortie, les variables locales sont conservées dans la table pour l'affichage final afin de permettre un audit complet de la compilation.

4.3.3 Représentation de la Table des Symboles (Exemple)

Le tableau suivant est présenté à titre d'**exemple illustratif**. Il représente l'état final de la table après l'analyse d'un programme de test spécifique comprenant diverses structures imbriquées et différents types de données.

Nom	Type Sym	Type Data	Portée	Adr.	Init.	Info
x	Variable	Entier	0	0	Oui	-
y	Variable	Entier	0	1	Oui	-
message	Variable	Chaîne	0	2	Oui	-
price	Variable	Réel	0	3	Oui	-
active	Variable	Booléen	0	4	Oui	-
inactive	Variable	Booléen	0	5	Oui	-
scores	Variable	Tableau	0	6	Oui	Taille=5
Student	Constante	Enregistrement	0	7	Oui	-
Point	Constante	Enregistrement	0	8	Oui	-
student1	Variable	Enregistrement	0	9	Non	-
point1	Variable	Enregistrement	0	10	Non	-
contacts	Variable	Dictionnaire	0	11	Non	-
i	Variable	Entier	1	13	Oui	Local (Niv. 1)
j	Variable	Entier	2	15	Oui	Local (Niv. 2)
result	Variable	Entier	0	16	Oui	-
sum	Variable	Réel	0	17	Oui	-
negative	Variable	Entier	0	18	Oui	-

TABLE 4.1 – Exemple d'état de la table des symboles obtenu lors d'un test d'analyse sémantique.

4.3.4 Analyse des résultats de l'exemple

L'observation de cet exemple permet de valider plusieurs points critiques de notre implémentation :

— **Imbrication :** Le variables *j* possède une portée de niveau 2, confirmant la bonne gestion des structures de contrôle imbriquées par le mécanisme d'`entrerPortee` et `sortirPortee`.

— **Adressage :** L'adresse mémoire (colonne *Adr.*) est incrémentée de façon séquentielle pour chaque nouveau symbole, simulant une allocation mémoire ordonnée.

— **Sécurité :** Les variables *student1* et *point1* sont marquées comme non-initialisées (`Init: Non`), permettant ainsi au compilateur de détecter et de signaler toute tentative d'utilisation de ces variables avant affectation.

4.4 Conclusion

L'analyse syntaxique permet de valider la forme du programme. Une fois la structure validée et la table des symboles complétée, le compilateur peut passer à l'analyse sémantique pour vérifier la cohérence du sens (typage, portée des variables) et préparer la génération de code.

Chapitre

5 Analyse Sémantique

5.1 Analyse Sémantique

L'analyse sémantique constitue la troisième phase majeure de notre compilateur QueryLang. Contrairement aux phases précédentes qui se concentraient sur la forme du code (lexicale) et sa structure (syntaxique), cette phase vérifie la cohérence et le sens du programme.

5.1.1 Objectifs de l'Analyse Sémantique

Notre analyseur sémantique a été conçu pour détecter et signaler plusieurs catégories d'erreurs qui ne peuvent pas être capturées par l'analyse syntaxique seule :

— **Déclarations et portée** : Vérification que toutes les variables utilisées ont été préalablement déclarées, et détection des redéclarations dans une même portée.
— **Compatibilité des types** : Contrôle de la cohérence des types dans les expressions arithmétiques, les affectations et les conditions.
— **Initialisation des variables** : Détection de l'utilisation de variables non initialisées, ce qui constitue une source fréquente d'erreurs difficiles à déboguer.
— **Accès aux structures de données** : Validation des accès aux tableaux (indices valides, types corrects) et aux enregistrements (existence des champs).
— **Erreurs d'exécution potentielles** : Détection statique de situations comme la division par zéro lorsque le diviseur est une constante.

5.1.2 Architecture de l'Analyseur Sémantique

5.1.2.1 Structures de Données

Pour gérer les erreurs et avertissements de manière structurée, nous avons défini plusieurs structures dans `semantic.h` :

```
typedef enum {
    SEM_ERROR_UNDECLARED,      // Variable non déclarée
    SEM_ERROR_REDECLARED,      // Double déclaration
    SEM_ERROR_TYPE_MISMATCH,   // Incompatibilité de types
    SEM_ERROR_DIV_BY_ZERO,     // Division par zéro
    SEM_ERROR_ARRAY_INDEX,     // Indice de tableau invalide
    SEM_ERROR_UNINITIALIZED,   // Variable non initialisée
    SEM_ERROR_CONST_ASSIGNMENT // Tentative de modification d'une constante
} SemanticErrorType;
```

Listing 5.1 – Types d'erreurs sémantiques

Les avertissements (warnings) sont distingués des erreurs car ils signalent des situations suspectes mais pas nécessairement bloquantes :

```
typedef enum {
    SEM_WARNING_UNUSED_VAR,   // Variable déclarée mais jamais utilisée
    SEM_WARNING_UNINIT_USE    // Utilisation d'une variable non initialisée
} SemanticWarningType;
```

Listing 5.2 – Types d'avertissements

5.1.2.2 Intégration avec la Table des Symboles

L'analyse sémantique s'appuie fortement sur la table des symboles enrichie développée dans la phase syntaxique. Cette table, déjà peuplée lors du parsing, contient pour chaque symbole :

- Son nom et son type (variable, constante, tableau...)
- Son type de données (entier, réel, chaîne...)
- Sa portée (niveau d'imbrication)
- Son statut d'initialisation
- Sa position dans le code source (ligne, colonne)
- Des informations spécifiques (taille pour les tableaux, valeur pour les constantes)

Cette richesse d'information permet à l'analyseur sémantique d'effectuer des vérifications précises et de fournir des messages d'erreur détaillés.

5.1.3 Vérifications Implémentées

5.1.3.1 Vérification des Déclarations

La fonction `checkDeclaration()` vérifie qu'une déclaration est sémantiquement correcte. Bien que la détection des doubles déclarations soit déjà effectuée lors de l'insertion dans la table des symboles (dans le parser), cette fonction se concentre sur la validation de l'initialiseur :

```
int checkDeclaration(ASTNode* node) {
    if (node->data.declaration.initializer != NULL) {
        DataType exprType;

        // Vérifier l'expression d'initialisation
        if (!checkExpression(node->data.declaration.initializer, &exprType)) {
            return 0;
        }

        // Vérifier la compatibilité des types
        if (!areTypesCompatible(node->data.declaration.dataType, exprType)) {
            reportSemanticError(SEM_ERROR_TYPE_MISMATCH,
                "Type de l'initialiseur incompatible", ...);
            return 0;
        }
```

```
    }
    return 1;
}
```

Listing 5.3 – Vérification d'une déclaration

5.1.3.2 Vérification des Affectations

Les affectations font l'objet de plusieurs vérifications :

1. **Existence de la variable** : On vérifie que la variable à gauche du signe = a bien été déclarée.
2. **Modification des constantes** : Une erreur est levée si on tente de modifier une constante.
3. **Compatibilité des types** : Le type de l'expression à droite doit être compatible avec celui de la variable.

```
// Vérifier qu'on n'assigne pas à une constante
if (sym->typeSymbole == TYPE_CONSTANTE) {
    reportSemanticError(SEM_ERROR_CONST_ASSIGNMENT,
        "Impossible de modifier la constante", ...);
    return 0;
}

// Vérifier la compatibilité des types
if (!areTypesCompatible(varType, exprType)) {
    reportSemanticError(SEM_ERROR_TYPE_MISMATCH,
        "Type incompatible pour l'affectation", ...);
    return 0;
}
```

Listing 5.4 – Extrait de la vérification d'affectation

5.1.3.3 Vérification des Expressions

La fonction `checkExpression()` est récursive et parcourt l'arbre syntaxique de l'expression pour :

— Déterminer le type résultant de l'expression
— Vérifier que les opérations sont valides pour les types utilisés
— Détecter les divisions par zéro lorsque c'est possible statiquement

Pour les opérations binaires, nous avons implémenté une logique de promotion de types : si l'un des opérandes est de type `FLOAT`, le résultat sera également `FLOAT`.

```
case NODE_EXPR_BINOP: {
    DataType leftType, rightType;

    checkExpression(node->data.binOp.left, &leftType);
    checkExpression(node->data.binOp.right, &rightType);

    // Vérification division par zéro
    if ((node->data.binOp.op == AST_OP_DIV ||
         node->data.binOp.op == AST_OP_MOD)) {
```

```
        if (isLiteralZero(node->data.binOp.right)) {
            reportSemanticError(SEM_ERROR_DIV_BY_ZERO,
                "Division par zéro détectée", ...);
        }
    }

    // Déterminer le type résultat
    if (leftType == TYPE_FLOAT || rightType == TYPE_FLOAT) {
        *resultType = TYPE_FLOAT;
    } else {
        *resultType = leftType;
    }
}
```

Listing 5.5 – Vérification d'une opération binaire

5.1.3.4 Vérification des Accès aux Tableaux

Les tableaux nécessitent des vérifications particulières qui combinent plusieurs aspects :

1. **Existence du tableau** : Vérifier que l'identifiant référence bien un tableau déclaré.
2. **Type de l'indice** : L'indice doit être de type `INTEGER`.
3. **Dépassement de bornes** : Lorsque l'indice est une constante, on peut vérifier statiquement qu'il respecte les limites du tableau.

```
int checkArrayIndex(ASTNode* node) {
    // Vérifier le type de l'index
    DataType indexType;
    checkExpression(indexExpr, &indexType);

    if (indexType != TYPE_INTEGER) {
        reportSemanticError(SEM_ERROR_TYPE_MISMATCH,
            "Index de tableau doit être ENTIER", ...);
        return 0;
    }

    // Vérification du dépassement (si constante)
    int constIndexValue;
    if (evaluateConstantInt(indexExpr, &constIndexValue)) {
        if (constIndexValue < 0 || constIndexValue >= arraySym->taille) {
            reportSemanticError(SEM_ERROR_ARRAY_INDEX,
                "Index hors limites", ...);
            return 0;
        }
    }
    return 1;
}
```

Listing 5.6 – Vérification de l'accès à un tableau

La fonction `evaluateConstantInt()` tente d'évaluer une expression à la compilation si elle ne contient que des constantes, permettant ainsi de détecter certaines erreurs avant l'exécution.

5.1.3.5 Gestion de la Portée

QueryLang supporte les portées imbriquées (dans les structures de contrôle comme `WHEN`, `LOOP`, etc.). L'analyseur sémantique doit donc :

- Entrer dans une nouvelle portée au début d'un bloc
- Vérifier que les variables sont accessibles selon leur niveau de portée
- Sortir de la portée à la fin du bloc

Par exemple, pour une structure conditionnelle :

```
int checkIfStatement(ASTNode* node) {
    // Vérifier la condition
    checkCondition(node->data.ifStmt.condition);

    // Bloc THEN
    entrerPortee(&tableGlobale);
    checkStatements(node->data.ifStmt.then_block);
    sortirPortee(&tableGlobale);

    // Bloc ELSE (si présent)
    if (node->data.ifStmt.else_block != NULL) {
        entrerPortee(&tableGlobale);
        checkStatements(node->data.ifStmt.else_block);
        sortirPortee(&tableGlobale);
    }

    return 1;
}
```

Listing 5.7 – Gestion de portée dans un IF

5.1.4 Détection des Variables Non Utilisées

Un aspect intéressant de notre analyseur est la détection des variables déclarées mais jamais utilisées. Cette fonctionnalité s'appuie sur un système de marquage à trois états :

- `initialise = 0` : Variable déclarée mais non initialisée
- `initialise = 1` : Variable déclarée et initialisée mais jamais utilisée
- `initialise = 2` : Variable utilisée dans le programme

La fonction `markVariableAsUsed()` parcourt récursivement l'AST et marque toutes les variables référencées :

```
void markVariableAsUsed(ASTNode* node) {
    if (node->type == NODE_EXPR_IDENTIFIER) {
        Symbole* sym = obtenirSymbole(&tableGlobale,
                                      node->data.identifier.name);
        if (sym != NULL) {
            sym->initialise = 2;  // Marquer comme utilisée
        }
    }
    // Cas récursifs pour les autres types de nuds...
}
```

Listing 5.8 – Marquage des variables utilisées

Après le parcours complet, la fonction `checkUnusedVariables()` examine la table des symboles et génère des avertissements pour toutes les variables dont le marqueur est resté à 0 ou 1.

5.1.5 Système de Rapports d'Erreurs

Notre compilateur distingue soigneusement les erreurs (qui bloquent la compilation) des avertissements (qui signalent des situations suspectes). Cette distinction est importante car elle permet de :

— Compiler du code avec des avertissements (par exemple, variable non utilisée)
— Bloquer la compilation en présence d'erreurs réelles (variable non déclarée)
— Collecter toutes les erreurs en un seul passage plutôt que de s'arrêter à la première

Les messages d'erreur incluent systématiquement :

— Le type d'erreur (pour une classification facile)
— Un message descriptif
— La position exacte (ligne et colonne)

```
ERREURS SÉMANTIQUES DÉTECTÉES


File "test_division.ql", line 10, character 23: semantic error
  Type de l'initialiseur (CHAÎNE) incompatible avec le type déclaré (RÉEL)
    ↪ pour 'x'
File "test_division.ql", line 9, character 25: semantic error
  Division par zéro détectée
```

Listing 5.9 – Exemple de sortie d'erreurs sémantiques

5.1.6 Intégration dans le Processus de Compilation

L'analyse sémantique est appelée depuis le programme principal (`main.c`) immédiatement après le succès de l'analyse syntaxique :

```c
// Phase 1: Analyse lexicale et syntaxique
int result = yyparse();

if (result != 0 || root == NULL) {
    printf("Analyse syntaxique échouée !\n");
    return 1;
}

// Phase 2: Analyse sémantique
int semanticResult = performSemanticAnalysis(root);

if (semanticResult) {
    printf("Compilation réussie !\n");
```

```
14      return 0;
15  } else {
16      printf("Compilation échouée - erreurs sémantiques\n");
17      return 1;
18  }
```

Listing 5.10 – Appel de l'analyse sémantique

5.1.7 Limitations et Améliorations Futures

Bien que notre analyseur sémantique couvre les aspects essentiels, certaines améliorations pourraient être envisagées :

- **Vérification des enregistrements** : Actuellement, la validation des accès aux champs d'enregistrement est basique. Une vérification plus approfondie de l'existence des champs serait souhaitable.
- **Analyse de flot de contrôle** : Détecter le code inaccessible ou les boucles infinies évidentes.
- **Inférence de types** : Pour les structures complexes, un système d'inférence de types plus sophistiqué pourrait améliorer l'expérience développeur.
- **Optimisations** : Certaines vérifications pourraient être optimisées pour les programmes de grande taille, notamment en utilisant des tables de hachage pour la recherche de symboles.

5.1.8 Conclusion

L'analyse sémantique que nous avons développée permet de détecter efficacement les erreurs qui échappent aux phases précédentes. Grâce à la vérification des types, la gestion des portées et la détection des situations problématiques, le compilateur peut signaler les problèmes avant l'exécution du programme.

Le système de messages d'erreur que nous avons mis en place s'est révélé particulièrement utile lors des tests. En indiquant précisément la ligne, la colonne et la nature du problème, il devient beaucoup plus facile de corriger le code. La distinction entre erreurs bloquantes et simples avertissements permet également une certaine souplesse dans le développement.

Il faut noter que cette phase d'analyse sémantique demande un travail considérable - elle constitue une partie importante du compilateur. Cependant, ce travail est nécessaire pour assurer la robustesse du système. De plus, les vérifications effectuées ici faciliteront grandement la génération de code intermédiaire, puisque nous pourrons partir d'un AST dont la cohérence a été validée.

Chapitre

6 Génération du code intermédiaire

6.1 Introduction

La génération de code intermédiaire constitue l'étape charnière d'un compilateur, située entre l'analyse sémantique et la génération du code machine cible. Son rôle principal est de transformer l'**Arbre Syntaxique Abstrait (AST)**, une structure hiérarchique complexe, en une forme linéaire et simplifiée appelée **code à trois adresses** ou **quadruplets**.

Chaque quadruplet est représenté par la structure $(Op, Arg1, Arg2, Res)$, où :

- **Op** est l'opérateur (arithmétique, logique ou de saut).
- **Arg1** et **Arg2** sont les opérandes.
- **Res** est le résultat de l'opération ou l'adresse cible d'un saut.

Cette représentation permet de s'abstraire des spécificités de la machine cible tout en offrant un support idéal pour les phases ultérieures d'optimisation de code.

6.2 Analyse de la fonction `generer_code`

La fonction `generer_code` implémente le parcours récursif de l'AST. Elle assure la traduction de chaque construction syntaxique du langage *QueryLang* en une suite logique d'instructions atomiques.

6.2.1 Gestion des Expressions et Temporaires

Pour évaluer des expressions complexes, le compilateur génère des variables temporaires $(T_0, T_1, \dots)$. Cela permet de décomposer une instruction comme `x = (a + b) * c` en plusieurs étapes élémentaires.

```
case NODE_EXPR_BINOP: {
    char* t1 = generer_code(node->data.binOp.left);
    char* t2 = generer_code(node->data.binOp.right);
    char* res = creer_temp(); // Allocation d'un nouveau registre temporaire
    generer_quad(operatorToString(node->data.binOp.op), t1, t2, res);
    return res;
}
```

Listing 6.1 – Traduction des opérations binaires

6.2.2 Contrôle de Flux et Backpatching

Le contrôle de flux (boucles et conditions) nécessite une gestion précise des adresses de saut. Puisque l'adresse de destination d'un branchement n'est pas connue au moment de sa création, nous utilisons la technique du **Backpatching**.

6.2.2.1 Instructions Conditionnelles (IF-THEN-ELSE)

Lors de la rencontre d'un bloc `IF`, un saut conditionnel `BZ` (Branch if Zero) est généré. Si la condition est fausse, le programme doit sauter au bloc `ELSE` ou à la fin de l'instruction. L'adresse de ce saut est mise à jour dynamiquement une fois que la position cible est générée dans la table.

6.2.2.2 Boucles de Répétition (WHILE)

La boucle nécessite deux ancres : une adresse de début pour réévaluer la condition à chaque itération, et un saut inconditionnel `BR` à la fin du corps de la boucle pour assurer la répétition.

6.2.3 Structures complexes : Tableaux et Records

L'implémentation de *QueryLang* se distingue par sa gestion robuste des types structurés :

— **Tableaux :** Utilisation de `BOUNDS` pour la déclaration et de l'opérateur `[]=` pour l'initialisation indexée.
— **Records :** Linéarisation des accès aux champs (ex : `student1.age`) avec l'usage de `strdup` pour garantir que les noms composés persistent en mémoire.

```
case NODE_RECORD_ACCESS_ASSIGN: {
    char* val = generer_code(node->data.recordAccessAssign.expression);
    char path[128];
    sprintf(path, "%s.%s", node->data.recordAccessAssign.recordName,
                           node->data.recordAccessAssign.fieldName);
    generer_quad("=", val, "", path);
    return strdup(path); // Securite memoire
}
```

Listing 6.2 – Linéarisation d'un accès Record

6.3 Comparaison Code Source vs Quadruplets

La figure ci-dessous démontre la fidélité de la génération de code par rapport au programme source initial.

6.4 Analyse des Résultats Expérimentaux

L'observation des quadruplets produits valide les mécanismes implémentés :

— **Allocation (6-11) :** Le tableau `scores` est correctement initialisé via une séquence d'instructions `[]=`.

Extrait du Code Source **Quadruplets Résultants**

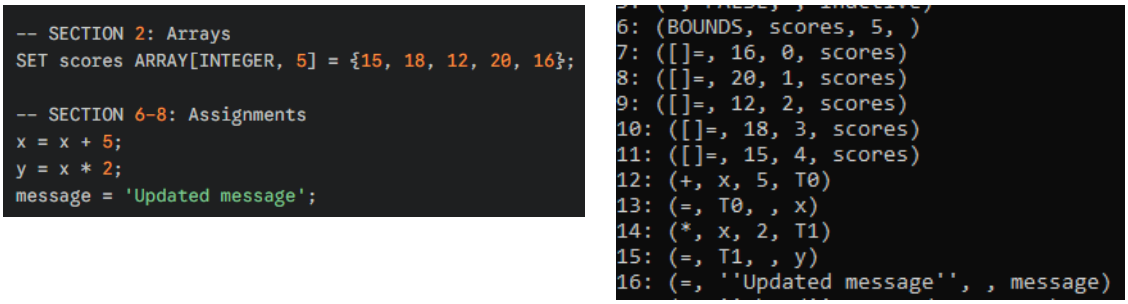

FIGURE 6.1 – Visualisation de la transformation du code source en code intermédiaire.

— **Linéarisation (12-15) :** Les calculs arithmétiques utilisent efficacement les registres temporaires.
— **Contrôle (37-41) :** La structure de la boucle est respectée avec une évaluation de condition suivie d'un branchement de retour.

6.5 Conclusion

La phase de génération de code intermédiaire constitue l'aboutissement logique de l'analyse syntaxique et sémantique. Notre implémentation, basée sur une gestion rigoureuse de la récursivité et de la mémoire dynamique, produit une table de quadruplets fiable et optimisée. Ce module garantit que toute structure complexe du langage *QueryLang* peut être réduite à une suite d'opérations élémentaires compréhensibles par une machine virtuelle.

Conclusion

Ce projet a été pour nous une véritable aventure intellectuelle au cur des rouages de l'informatique. Bien plus qu'un simple exercice de programmation, il nous a permis de lever le voile sur les mystères de la compilation et de transformer les notions théoriques vues en cours analyses lexicale, syntaxique et sémantique en un outil concret et fonctionnel.

L'exploration du langage **QueryLang** nous a offert un aperçu approfondi de la structure interne des langages de programmation, nous permettant d'étudier des aspects techniques souvent invisibles et complexes. Nous avons réalisé que chaque instruction, aussi simple soit-elle, cache une logique rigoureuse et une hiérarchie fascinante.

Ce projet restera une expérience fondatrice qui nous a appris que derrière la simplicité apparente d'un code source se cache un monde de précision que nous avons désormais hâte de continuer à explorer.