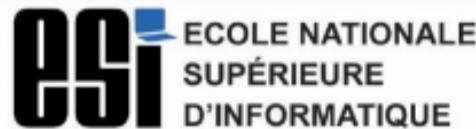


الجمهورية الجزائرية الديمقراطية الشعبية
+٤٢٨٠٦٤٣ ٩٨٥٧٥٥٤٦
République Algérienne Démocratique et Populaire
وزارة التعليم العالي والبحث العلمي
٠٢٣٤٣ ٨٤٠٨٤ ٣٢٣٦٨
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique



ECOLE NATIONALE
SUPÉRIEURE
D'INFORMATIQUE

المدرسة الوطنية العليا للإعلام الآلي
٠٢٣٤٣ ٨٤٠٨٤ ٣٢٣٦٨
École nationale Supérieure d'Informatique

Projet compilation

2ème année Cycle Supérieur (2CS)

Option : Systèmes Informatiques et Logiciels (SL)

Livrable final:

Projet Compilation

Réalisé par :

- Bengherbia Abdelkarim (G2)
- Yekene Sofiane (G2)
- Mezenner Fares (G2)
- Arabet Mohamed Ilyes (G2)
- Bouacha Chamel Nadir (G2)

Table des matières

Table des matières	II
Introduction Générale	1
Proposition du langage	2
Introduction	1
Structure Générale du Programme	3
2.1 Déclaration Des Variables	3
2.2 Instructions	3
2.3 Exemple Illustratif D'un Programme	4
Les Commentaires	5
Les Variables	5
4.1. Variables simples	5
4.2. Variables structurées	6
Les Types	6
5.1. Entier (int)	6
5.2. Flottant (float)	6
5.3. Booléen (bool)	7
5.4 Chaîne de caractères (string)	7
Instructions de base	7
6.1. Blocs de contrôle	7
6.2 Opérateurs logiques, de comparaison, et arithmétiques	9
6.3 Les différentes priorités et règles d'associativité	9
Conclusion	11
Analyse lexicale	12
Introduction	12
Structure du fichier analyseLexical.l	12
2.1 Section De Définitions	12
2.2 Section Des Actions	12
2.3 Section Du Code	13
Test par un code	13
3.1. Exemple de test	14
3.2. Résultat de test	14
Conclusion	15
Analyse syntaxique	16
Introduction	16
Génération des Tables d'Analyse Syntaxique	17

2.1. Choix du sous-langage à analyser	17
2.2. Méthode choisie : description et justification	19
2.3. Algorithme d'implémentation en C	19
2.4. Tests et validation des tables générées	19
Implémentation avec BISON	21
3.1. Introduction à l'outil BISON	21
3.2. Spécification des règles de grammaire dans le fichier BISON	21
3.3. Règles d'associativité et de priorité	24
Détection et affichage des erreurs	25
Tests et Validation	25
Passage des valeurs des tokens entre le lexer et Bison	27
Conclusion	28
Analyse sémantique	29
Introduction	29
Structure des Données	29
2.1 Table des Symboles	29
2.2 Gestion de la table	30
2.3 Rôle de la Table des Symboles	31
Vérifications Sémantiques	32
3.1 Vérification des Types	32
3.2 Existence variable	34
3.3 Verification des Tableaux	35
3.4 Division sur 0	35
Génération de Code Intermédiaire	36
4.1 Structure des Quadruplets	36
4.2 Gestion des quadruplets	36
4.3 Gestion des étiquettes et des boucles imbriquées	37
4.4 Exemples de génération de quadruplets : while et if-else	38
4.5 Exemples Complets de génération de quadruplets	40
Gestion des Erreurs	41
Points Forts et Améliorations	42
Conclusion	43
Conclusion Générale	44

Introduction Générale

La conception et l'implémentation d'un compilateur constituent une tâche complexe qui repose sur une série d'étapes interdépendantes. Chaque phase, de l'analyse lexicale à l'analyse sémantique, joue un rôle spécifique pour garantir que le code source respecte les règles syntaxiques et sémantiques du langage cible et qu'il peut être traduit en instructions exécutables.

L'analyse lexicale est le point de départ du processus. Elle se concentre sur la transformation du code source brut en une séquence de tokens bien définis, chaque token représentant une unité syntaxique significative comme un mot-clé, un opérateur ou une variable. Cette étape permet d'établir une base structurée pour les analyses suivantes tout en détectant les erreurs lexicales, telles que les caractères invalides ou les identifiants mal formés.

L'analyse syntaxique, étape centrale du processus, vérifie que la séquence de tokens respecte les règles grammaticales définies pour le langage. En utilisant des outils tels que les tables d'analyse LL(1) et des générateurs comme Bison, elle valide la structure hiérarchique du programme, identifiant les relations entre les différentes composantes syntaxiques. Cette phase repose sur des algorithmes robustes comme la descente récursive pour décomposer et interpréter chaque construction syntaxique.

Enfin, l'analyse sémantique intervient pour assurer que le programme est non seulement syntaxiquement valide mais également cohérent sur le plan logique. Elle inclut des vérifications telles que la déclaration préalable des variables, la compatibilité des types et le respect des règles de portée. À travers l'utilisation d'une table des symboles, cette phase garantit la validité des opérations et la cohérence globale du programme. Elle fournit également les bases nécessaires à la génération de code intermédiaire et à l'optimisation.

Ce rapport explore en profondeur ces trois phases en les appliquant à un langage spécifique, HTPL. Chaque étape est détaillée, depuis la détection des tokens jusqu'à la génération des quadruplets pour les structures complexes. En mettant en lumière les défis rencontrés et les solutions apportées, il illustre comment une conception méthodique et des outils bien choisis peuvent aboutir à un compilateur fonctionnel et efficace.

Proposition du langage

1. Introduction

Dans le cadre du projet de compilation, nous avons conçu un langage de programmation novateur nommé **HTPL**, qui allie simplicité, lisibilité et efficacité. Inspirée par la structure des balises XML/HTML, cette approche permet de créer une syntaxe intuitive et bien structurée, facilitant la compréhension et l'écriture des programmes. Ce choix de conception rend le langage accessible, même pour les débutants, tout en assurant une compilation rapide et performante. **HTPL** comprend toutes les fonctionnalités essentielles d'un langage de programmation moderne, telles que la déclaration de variables, la gestion des boucles et des conditions, ainsi que les opérations arithmétiques et logiques de base. Une attention particulière a été portée aux différentes priorités et règles d'associativité, afin de garantir une logique cohérente.

Ce document est structuré de manière à présenter les principales caractéristiques et fonctionnalités de **HTPL**. Nous débutons par la Structure Générale du Programme, qui donne un aperçu de l'organisation globale d'un programme en **HTPL**, permettant de saisir la logique de base du langage. La section suivante, Les Commentaires, explique comment insérer des annotations dans le code pour améliorer sa lisibilité et faciliter la documentation, un élément crucial pour le travail collaboratif et la maintenance.

La section Les Variables expose les méthodes de stockage et de manipulation des données, suivie de Les Types, qui décrit les différents types de données pris en charge par le langage, tels que les entiers, les flottants, les booléens et les chaînes de caractères. Ces concepts forment le socle de la gestion des données dans **HTPL** et permettent une grande flexibilité dans la programmation.

Et finalement, nous abordons les Instructions de base, qui couvrent les blocs de contrôle fondamentaux, incluant les conditions, les boucles, les affectations, ainsi que les entrées et sorties. Cette section fournit des exemples concrets sur la manière d'utiliser ces structures pour organiser la logique et le flux d'exécution d'un programme **HTPL**.

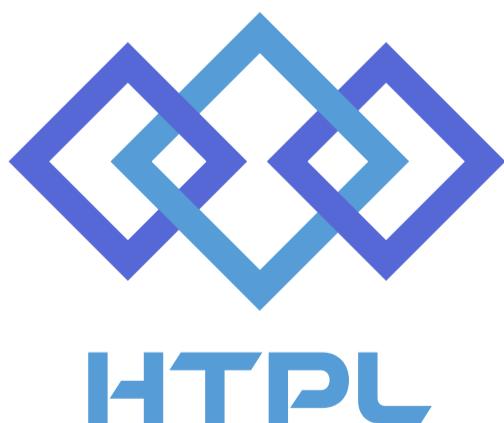


Figure1 : Logo du langage HTPL

2. Structure Générale du Programme

Un programme dans ce langage est encapsulé dans une balise <program>

```
<program>
    # Déclaration du programme "Example"

    <variables>
        # Déclaration des variables
    </variables>

    <instructions>
        # Section des instructions
    </instructions>
</program>
```

2.1. Déclaration des Variables

Les variables sont définies dans la balise <variables> et chaque variable utilise une balise spécifique comme <int> ou <string> pour indiquer son type.

Une initialisation optionnelle peut être incluse directement dans la balise.

```
<variables>
    <int name="x">(10)</int> # Déclare une variable
entière 'x' initialisée à 10
    <string name="message">"Bonjour"</string> # Déclare
une variable chaîne 'message' initialisée avec "Bonjour"
</variables>
```

2.2. Instructions

Les instructions du programme sont placées dans la balise <instructions>.

```

<instructions>
    # Condition SI/SINON (if/else)
    <if (x >> 0)>
        <print value="x is positive"/>
        #Si x est supérieur à 0, affiche que x est positif
    <else>
        <print value="x is less than or equal to 0"/>
        # Sinon, affiche que x est inférieur ou égal à 0
    </if>
</instructions>

```

2.3 Exemple illustratif d'un programme

```

<program name="Example">
    # Déclaration du programme nommé "Example"

    # Déclaration des variables utilisées dans le programme
    <variables>
        <int name="x">(10)</int> # Déclare une variable entière 'x'
initialisée à 10
        <string name="message">"Bonjour"</string> # Déclare une
variable chaîne 'message' initialisée avec "Bonjour"
    </variables>

    # Début des instructions du programme
    <instructions>
        # Boucle TANT_QUE (while) : Tant que la valeur de x est
supérieure à 0
        <while (x >> 0)>
            <print value="x is still greater than 0"/> # Affiche un
message indiquant que x est encore supérieur à 0
            <assign x=(x - 1)/> # Décrémente la valeur de x
        </while>

        # Condition SI/SINON (if/else)
        <if (x >> 0)>
            <print value="x is positive"/>
            #Si x est supérieur à 0, affiche un message disant que x
est positif
        <else>
            <print value="x is less than or equal to 0"/>
            # Sinon, affiche que x est inférieur ou égal à 0
        </if>
    </instructions>
</program>

```

3. Les Commentaires

Dans ce langage de programmation, les commentaires commencent avec le symbole # et s'étendent jusqu'à la fin de la ligne.

```
# Ceci est un commentaire  
<int name="age">(25)</int> # Cette variable stocke l'âge
```

4. Les Variables

Les variables doivent être déclarées dans une seule section dédiée au début du programme, délimitée par les balises **<variables>** et **</variables>**.

4.1 Variables simples

Les variables simples (ou primitives) permettent de stocker des données basiques comme des nombres et du texte. Pour déclarer une variable simple :

- Ouvrir une balise correspondant au type de variable.
- Utiliser l'attribut **name** pour nommer la variable.

```
<variables>  
    <int name="age"/> # Déclare une variable entière sans valeur  
    initiale  
  
    <string name="nom"/> # Déclare une variable chaîne de  
    caractère sans valeur initiale  
</variables>
```

Initialisation de variables simples :

- Pour initialiser une variable, ajoutez sa valeur directement entre les balises ouvrante et fermante :

```
<variables>  
    <int name="age">(25)</int> # La variable 'age' est  
    initialisée à 25  
  
    <string name="nom">"Alice"</string> # La variable 'nom'  
    contient le texte "Alice"  
</variables>
```

4.2 Variables structurées

Ce langage permet aussi de déclarer des variables structurés, comme des tableaux , et ces derniers doivent être définis avec :

- La balise `<array>`
- L'attribut `name` pour nommer le tableau
- L'attribut `type` pour préciser le type des éléments
- L'attribut `size` pour indiquer la taille du tableau.

Chaque élément du tableau est défini avec une balise `<element>` et l'attribut `value` pour sa valeur.

Exemple de tableau d'entiers:

Le tableau nombres est un tableau de 5 entiers , où chaque élément est déclaré avec `<element>` et initialisé avec une valeur via l'attribut `value`.

```
<variables>
  <array name="nombres" type="int" size="5">
    <element value=(1)/>
    <element value=(2)/>
    <element value=(3)/>
    <element value=(4)/>
    <element value=(5)/>
  </array>
</variables>
```

5. Les Types

Dans ce langage de programmation, les variables peuvent être définies en utilisant des types spécifiques permettant de gérer les nombres, les valeurs booléennes et les chaînes de texte. Voici un aperçu de chaque type :

5.1 Entier (int)

Le type `int` représente les nombres entiers dans une plage spécifique, allant de -32 768 à 32 767. Ce type est idéal pour stocker des valeurs entières sans virgule décimale, telles que des comptages, des indices ou toute donnée nécessitant une plage limitée de nombres entiers.

5.2 Flottant (float)

Le type `float` représente les nombres réels, permettant à la fois les valeurs positives et négatives avec des décimales. La plage de valeurs pour `float` est d'environ -3,4E38 à 3,4E38, ce qui le rend adapté pour gérer des nombres très grands ou très petits avec précision.

5.3 Booléen (bool)

Le type **bool** est utilisé pour les valeurs binaires **true** or **false**. Ce type de donnée ne permet que deux états : **true** or **false**, ce qui le rend essentiel pour les opérations logiques, les indicateurs et les conditions de contrôle de flux.

5.3 Chaîne de caractères (string)

Le type **string** représente des séquences de caractères, telles que des textes et des mots. Ce type n'a pas de restriction de longueur, permettant de stocker toute quantité de données textuelles. Il est adapté pour contenir des informations telles que des noms, des phrases ou tout autre contenu basé sur du texte.

Ces types de variables offrent une grande flexibilité et précision dans la gestion des données, en offrant des options pour représenter des données numériques, logiques et textuelles au sein du langage.

6. Instructions de base

6.1 Blocs de contrôle

Les blocs de contrôle suivent une syntaxe simple, facilitant la définition des flux logiques, des boucles, des affectations, des entrées et des sorties. Ci-dessous se trouve une explication de chaque bloc avec des descriptions et des exemples de code pour illustrer leur fonction.

6.1.1 Bloc de condition

Le bloc de condition contrôle la prise de décision dans le programme. En utilisant la balise **<if** (**condition1**)**>**, il est possible de spécifier des conditions qui déterminent quelles instructions s'exécutent en fonction de la logique. Si aucune des conditions ne correspond, un bloc **<else>** fournit une option par défaut.

```
<if condition=(x >> 5)>
    # Code to execute if x is greater than 5
<else>
    # Code to execute if all conditions are false
</if>
```

6.1.2 Bloc de boucle

Le bloc de boucle permet l'exécution répétée d'instructions tant qu'une condition spécifiée reste vraie. En entourant le code avec les balises `<while condition=(expr_logique)>` et `</while>`, une boucle est créée qui s'exécute jusqu'à ce que la condition dans la balise `<while condition=()>` devienne fausse.

```
<while condition=(x >> 0)>
    <assign x=(x - 1)/>
</while>
```

6.1.3 Bloc d'affectation

Le bloc d'affectation permet de définir ou de mettre à jour les valeurs des variables. En utilisant la balise `<assign />`, il est possible d'attribuer une valeur ou le résultat d'une expression à une variable spécifiée.

```
<assign x=(x - 1)/> # evaluate x-1 and store the value in x
<assign y="Hello"/> # store hello in y
```

6.1.4 Bloc d'entrée

Le bloc d'entrée capture les saisies de l'utilisateur et les affecte à une variable spécifiée. Avec la balise `<input />`, il est possible de demander une saisie à l'utilisateur et de stocker sa réponse dans une variable pour une utilisation ultérieure dans le programme.

```
<input var=(varName)/>
```

6.1.5 Bloc de sortie

Le bloc de sortie permet d'afficher des informations dans la console ou la sortie standard. La balise `<print />` évalue l'expression ou la chaîne fournie dans son attribut `value` et en affiche le résultat. Si la valeur contient des caractères spéciaux comme `\n`, la sortie peut inclure un formatage, tel que des retours à la ligne.

```
<print value="Hello\n"/> # This will print 'Hello' followed by a
new line
```

Pour évaluer des expressions mathématiques ou des variables, il est possible d'effectuer des opérations au sein de l'attribut `value` :

```
<print value=(x-1)/> # This will evaluate the expression 'x - 1'
and print the result
```

6.2 Opérateurs logiques, de comparaison, et arithmétiques

Ce langage propose un ensemble d'opérateurs permettant de réaliser des opérations logiques, des comparaisons entre valeurs, ainsi que des calculs arithmétiques. Voici une liste complète des opérateurs disponibles :

6.2.1 Opérateurs logiques

`!` : Not logique unaire.

6.2.2 Opérateurs arithmétiques

`*`, `/`, `%` : Multiplication, division, modulo.

`+`, `-` : Addition, soustraction.

6.2.3 Opérateurs de comparaison

`>>`, `<<`, `>>=`, `<<=` : Comparaisons (supérieur, inférieur, supérieur ou égal, inférieur ou égal).

`==`, `!=` : Égalité et inégalité.

6.2.4 Opérateurs logiques avancés

`&&` : ET logique.

`||` : OU logique.

Ces opérateurs peuvent être combinés dans des expressions pour construire des calculs ou des conditions complexes.

6.3 Les différentes priorités et règles d'associativité

Pour gérer correctement les expressions complexes, ce langage applique des règles strictes de priorité et d'associativité pour ses opérateurs.

6.3.1 Priorité des opérateurs

Les opérateurs sont évalués dans l'ordre de priorité suivant, du plus haut au plus bas

1. Plus haute priorité :
`!` : Not unaire.
2. Priorité élevée :
`*, /, %` : Multiplication, division, modulo.
3. Priorité intermédiaire :
`+, -` : Addition, soustraction.
4. Priorité basse :
`>>, <<, >>=, <<=, ==, !=` : Comparaison.
5. Plus basse priorité :
`&&, ||` : Logique ET, OU.

6.3.2 Règles d'associativité

Les règles d'associativité définissent l'ordre d'évaluation des opérateurs de même priorité :

- Opérateurs unaires :
 Associativité de droite à gauche (exemple `!`).
- Multiplication, division, modulo (`*, /, %`) :
 Associativité de gauche à droite.
- Addition, soustraction (`+, -`) :
 Associativité de gauche à droite.
- Comparaison (`>>, <<, >>=, <<=, ==, !=`) :
 Associativité de gauche à droite.
- Logique ET (`&&`) :
 Associativité de gauche à droite.
- Logique OU (`||`) :
 Associativité de gauche à droite.

Ces règles garantissent une évaluation cohérente des expressions, minimisant les erreurs et assurant une interprétation correcte des priorités entre les opérateurs.

7. Conclusion

En conclusion, le langage **HTPL** que nous avons conçu se distingue par sa simplicité, sa lisibilité et sa structure inspirée des balises XML/HTML, qui en font un outil efficace pour les développeurs de tous niveaux. Grâce à des blocs de contrôle clairs et faciles à utiliser, comme les blocs de condition, de boucle, d'affectation, d'entrée et de sortie, HTPL permet de créer des programmes bien structurés tout en minimisant la complexité syntaxique. Cette approche facilite l'apprentissage et l'utilisation du langage, tout en garantissant une exécution performante des instructions.

En intégrant les fonctionnalités essentielles d'un langage de programmation moderne, HTPL répond aux besoins de flexibilité et de précision pour la manipulation des données numériques, logiques et textuelles. Cette conception axée sur la clarté et la simplicité vise à offrir une expérience de programmation fluide, permettant aux utilisateurs de se concentrer sur la logique et la fonctionnalité de leurs programmes. Nous espérons que **HTPL** trouvera une place dans les environnements éducatifs et professionnels, offrant un exemple solide de langage axé sur la facilité d'utilisation sans compromis sur les performances.

Analyse lexicale

1. Introduction

Ce rapport présente le développement d'un analyseur lexical pour un langage de programmation appelé HTPL, inspiré de la syntaxe des balises XML. Ce projet s'inscrit dans le cadre d'un TP. L'objectif principal était de concevoir un analyseur capable de reconnaître les éléments syntaxiques du langage, tout en intégrant des modifications pour améliorer sa clarté et sa structure. L'analyseur lexical joue un rôle crucial dans le processus de compilation. Il permet de découper le code source en unités syntaxiques, qui serviront de base pour l'analyse syntaxique et sémantique. Pour HTPL, nous avons implémenté un analyseur capable de reconnaître des balises, des attributs et des expressions, en respectant une grammaire spécifique inspirée de HTML. En complément, une table des symboles a été développée pour gérer les informations associées aux identificateurs du langage (variables, constantes, etc.). Ce composant permet de suivre les types, valeurs et états des éléments déclarés dans le programme, ce qui est essentiel pour les phases ultérieures de vérification et d'exécution. Ce document contient une présentation détaillée de l'analyseur lexical, incluant son code source et les structures utilisées. La table des symboles est également expliquée, accompagnée d'exemples concrets pour illustrer son fonctionnement. Enfin, une annexe décrit les modifications apportées à la syntaxe d'HTPL, notamment l'introduction d'une nouvelle notation pour les conditions, les opérateurs et les déclarations de variables. Ce travail a permis de combiner théorie et pratique, tout en posant les bases pour les étapes suivantes du développement du compilateur, comme l'analyse syntaxique et la génération de code.

2. Structure du fichier analyseLexical.l

2.1. Section de définitions

```
chiffre      [0-9]
ID          [a-zA-Z_][a-zA-Z0-9_]*
int         {chiffre}+
float        {int}."{int}
string       \"(\\".|[^"\\\])*\""
boolean      "true"|"false"
type         ("int"|"float"|"string"|"boolean")
comment     \#[^\n]*

%%
```

2.2. Section des actions

```

{comment}          { /* Ignore comments */ }

"["
"]"
{ current_token = TOKEN_OPEN_BRACKET; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_CLOSE_BRACKET; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }

"<program>"
"</program>"
"<variables>"
"</variables>"
"<instructions>"
"</instructions>"
"<assign>"
"</assign>"
"<print>"
"</print>"
"<if>"
"</if>"
"<else>"
"<while>"
"</while>"
"<array>"
"</array>"
"<element>"

{ current_token = TOKEN_PROGRAM_OPEN; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_PROGRAM_CLOSE; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_VARIABLES_OPEN; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_VARIABLES_CLOSE; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_INSTRUCTIONS_OPEN; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_INSTRUCTIONS_CLOSE; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_ASSIGN_OPEN; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_ASSIGN_CLOSE; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_PRINT_OPEN; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_PRINT_CLOSE; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_IF_OPEN; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_IF_CLOSE; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_ELSE; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN WHILE_OPEN; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN WHILE_CLOSE; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_ARRAY_OPEN; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_ARRAY_CLOSE; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_ELEMENT_OPEN; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }

"<int>"
"</int>"          { current_token = TOKEN_VAR_INT_OPEN; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_VAR_INT_CLOSE; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }

"<float>"
"</float>"         { current_token = TOKEN_VAR_FLOAT_OPEN; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_VAR_FLOAT_CLOSE; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }

"<boolean>"
"</boolean>"       { current_token = TOKEN_VAR_BOOLEAN_OPEN; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_VAR_BOOLEAN_CLOSE; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }

"<string>"
"</string>"        { current_token = TOKEN_VAR_STRING_OPEN; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_VAR_STRING_CLOSE; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }

"+"
"-"
"*"
"/"
">>>
"><<
"><=
">>=
"("
")"
"="
"\\""

{ current_token = TOKEN_PLUS; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_MINUS; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_MULTIPLY; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_DIVIDE; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_GREATER_THAN; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_LOWER_THAN; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_EQUAL; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_LOWER_OR_EQUAL; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_GREATER_OR_EQUAL; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_OPEN_PARENTHESIS; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_CLOSE_PARENTHESIS; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }

{ float}
{ int}
{ boolean}
{ ID}
"/>"
">"              { current_token = TOKEN_ASSIGN; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_QUOTE; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }

{ yylval.floatVal = atof(yytext); current_token = TOKEN_FLOAT; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ yylval.intVal = atoi(yytext); current_token = TOKEN_INT; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ yylval.boolVal = (strcmp(yytext, "true") == 0); current_token = TOKEN_BOOLEAN; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }

{ current_token = IDENTIFICATEUR; token_action(current_token, yytext, output_file); yysuccess(yytext); yylval.strVal = strdup(yytext); return current_token; }

{ current_token = TOKEN_SELF_CLOSING_TAG; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }
{ current_token = TOKEN_END_TAG; token_action(current_token, yytext, output_file); yysuccess(yytext); return current_token; }

{ string}          { current_token = TOKEN_STRING; token_action(current_token, yytext, output_file); yysuccess(yytext); yylval.strVal = strdup(yytext); return current_token; }

.

{ yyerror("Unexpected token"); yyterminate();}


```

2.3. Section du code

```

95 int main(int argc, char **argv) {
96     // Open input file if provided
97     if (argc > 1) {
98         yyin = fopen(argv[1], "r");
99         if (!yyin) {
100             fprintf(stderr, "Cannot open file %s\n", argv[1]);
101             return 1;
102         }
103     }
104
105     printf("Starting lexical analysis:\n");
106     while (yylex() != 0) {
107         // Token processing happens inside yylex
108     }
109
110     if (argc > 1) {
111         fclose(yyin);
112     }
113
114     printf("Lexical analysis complete.\n");
115     return 0;
116 }

```

3. Test par un code

3.1. Exemple de test

```
1  <program>
2  <variables>
3  |     <int name="counter">(3)</int>
4  |     <string name="message">"Test"</string>
5  |     </variables>
6
7  <instructions>
8  |     <while condition=(counter >> 0)>
9  |     |     <print value="Counter: + counter"/>
10 |     |     </while>
11 |     </instructions>
12 </program>
13
```

3.2. Résultat du test

```
Token: 259, Content: <program>
Token: 261, Content: <variables>
Token: 279, Content: <int
Token: 304, Content: name
Token: 300, Content: =
Token: 288, Content: "counter"
Token: 277, Content: >
Token: 298, Content: (
Token: 305, Content: 3
Token: 299, Content: )
Token: 280, Content: </int>
Token: 285, Content: <string
Token: 304, Content: name
Token: 300, Content: =
Token: 288, Content: "message"
Token: 277, Content: >
Token: 288, Content: "Test"
Token: 286, Content: </string>
Token: 262, Content: </variables>
Token: 263, Content: <instructions>
Token: 272, Content: <while
Token: 304, Content: condition
Token: 300, Content: =
Token: 298, Content: (
Token: 304, Content: counter
Token: 293, Content: >>
Token: 305, Content: 0
Token: 299, Content: )
Token: 277, Content: >
Token: 267, Content: <print
Token: 304, Content: value
Token: 300, Content: =
Token: 288, Content: "Counter: + counter"
Token: 278, Content: />
Token: 273, Content: </while>
Token: 264, Content: </instructions>
Token: 260, Content: </program>
```

```
Token: 9, Content: <print
Token: 19, Content: value
Token: 16, Content: ="\"Counter initialized to 10\""
Token: 20, Content: />
Token: 23, Content: <while
Token: 19, Content: condition
Token: 16, Content: ="counter > 0"
Token: 21, Content: >
Token: 9, Content: <print
Token: 19, Content: value
Token: 16, Content: ="\"Counter: \" + counter"
Token: 20, Content: />
Token: 7, Content: <assign
Token: 19, Content: name
Token: 16, Content: ="counter"
Token: 19, Content: value
Token: 16, Content: ="counter - 1"
Token: 20, Content: />
Token: 22, Content: </while>
Token: 11, Content: <if
Token: 19, Content: condition
Token: 16, Content: ="counter == 0"
Token: 21, Content: >
Token: 9, Content: <print
```

5. Conclusion

Ce projet nous a permis de comprendre le fonctionnement d'un analyseur lexical en créant un outil adapté à un langage personnalisé. Nous avons appris à utiliser Lex pour identifier des éléments syntaxiques, gérer des états spécifiques et construire une table des symboles. Ce travail pose une base solide pour poursuivre avec les étapes suivantes du développement d'un compilateur.

Analyse syntaxique

1. Introduction

Ce projet a pour objectif de concevoir et d'implémenter un analyseur syntaxique pour le langage de programmation HTPL inspiré du format XML.

Pour cela, nous avons commencé par définir une grammaire adaptée à ce sous-langage. Cette grammaire a été conçue pour respecter les contraintes des analyseurs syntaxiques LL(1), ce qui permet une analyse descendante déterministe. Après avoir générée les tables d'analyse correspondantes, nous avons implémenté un analyseur syntaxique en langage C, en utilisant la méthode de descente récursive. Cette méthode, bien que simple, nécessite une bonne maîtrise de la grammaire et des étapes de l'analyse syntaxique.

Dans une deuxième phase, nous avons utilisé l'outil Bison pour générer un analyseur syntaxique de manière semi-automatique. L'objectif était de comparer cette approche automatisée à l'implémentation manuelle, tout en explorant les possibilités offertes par Bison, notamment la gestion des règles de priorité, des erreurs, et la structuration des grammaires complexes. Cette étape nous a permis d'optimiser l'analyseur et de tester des fonctionnalités avancées.

Le rapport présente en détail chaque étape de ce projet : la définition de la grammaire, la génération des tables d'analyse, l'implémentation en C, l'utilisation de Bison, et les tests de validation. Nous abordons également les défis rencontrés et les solutions apportées pour garantir une bonne analyse syntaxique.

Ce projet nous a permis de mieux comprendre les concepts fondamentaux des analyseurs syntaxiques tout en les appliquant dans un cadre pratique.

2. Implémentation de l'analyse syntaxique

2.1. Choix du sous-langage à analyser

Pour ce projet, nous avons choisi d'analyser un sous-langage dédié à la déclaration de variables de type String. Voici un exemple de syntaxe caractéristique de ce sous-langage :

```
<variables>
  <string name="unTexte">"HelloWorld"</string>
  <string name="unTexte2">"HelloWorld"</string>
</variables>
```

2.1.1 Grammaire utilisée

```
<V> ::= TOKEN_VARIABLES_OPEN <E> TOKEN_VARIABLES_CLOSE
      | ε

<E> ::= TOKEN_VAR_STRING_OPEN <A> TOKEN_END_TAG TOKEN_STRING TOKEN_VAR_STRING CLOSE <E>
      | TOKEN_VAR_STRING_OPEN <A> TOKEN_END_TAG TOKEN_STRING TOKEN_VAR_STRING CLOSE
      | TOKEN_VAR_STRING_OPEN <A> TOKEN_SELF_CLOSING_TAG <E>
      | TOKEN_VAR_STRING_OPEN <A> TOKEN_SELF_CLOSING_TAG

<A> ::= TOKEN_ATTRIBUTE_NAME TOKEN_ASSIGN TOKEN_STRING
```

2.1.2 Améliorations de la grammaire

Bien que cette grammaire ne contient pas de récursivité gauche ni directe ni indirecte, elle présente une redondance au niveau de l'état `<E>`. Pour y remédier, nous avons factorisé la grammaire en ajoutant un état intermédiaire `<E1>` :

```
<V> ::= TOKEN_VARIABLES_OPEN <E> TOKEN_VARIABLES_CLOSE
      | ε

<E> ::= TOKEN_VAR_STRING_OPEN <A> <E1>

<E1> ::= TOKEN_END_TAG TOKEN_STRING TOKEN_VAR_STRING CLOSE <E2>
      | TOKEN_SELF_CLOSING_TAG <E2>

<E2> ::= <E>
      | ε

<A> ::= TOKEN_ATTRIBUTE_NAME TOKEN_ASSIGN TOKEN_STRING
```

2.1.3. Analyse de la grammaire

La grammaire ainsi transformée est désormais adaptée à l'analyse LL. Cette transformation permet une meilleure gestion des ambiguïtés et facilite la construction de la table d'analyse.

Ensembles des début et suivant:

Nous avons ensuite déterminé les ensembles des débuts et suivants pour chaque état :

Non-Terminal	First Set (Début)	Follow Set (Suivant)
V	{TOKEN_VARIABLES_OPEN, ϵ }	{#}
E	{TOKEN_VAR_STRING_OPEN}	{TOKEN_VARIABLES_CLOSE}
E1	{TOKEN_END_TAG, TOKEN_SELF_CLOSING_TAG}	{TOKEN_VARIABLES_CLOSE}
E2	{TOKEN_VAR_STRING_OPEN, ϵ }	{TOKEN_VARIABLES_CLOSE}
A	{TOKEN_ATTRIBUTE_NAME}	{TOKEN_END_TAG, TOKEN_SELF_CLOSING_TAG}

2.1.4. Table d'analyse LL

En utilisant ces ensembles, nous avons généré la table d'analyse LL qui servira à guider l'analyseur syntaxique. Cette table peut être construite automatiquement en associant chaque combinaison de terminal et non-terminal à une règle de production appropriée.

Voici la numérotation des règles de production:

1.	<V>	::=	TOKEN_VARIABLES_OPEN	<E>	TOKEN_VARIABLES_CLOSE
2.	<V>	::=	ϵ		
3.	<E>	::=	TOKEN_VAR_STRING_OPEN	<A>	<E1>
4.	<E1>	::=	TOKEN_END_TAG	TOKEN_STRING	TOKEN_VAR_STRING_CLOSE
5.	<E1>	::=	TOKEN_SELF_CLOSING_TAG	<E2>	
6.	<E2>	::=	<E>		
7.	<E2>	::=	ϵ		
8.	<A>	::=	TOKEN_IDENTIFIER	TOKEN_ASSIGN	TOKEN_STRING

voici la table d'analyse:

Non-Terminal	TOKEN_VARIABLES_OPEN	TOKEN_VARIABLES_CLOSE	TOKEN_VAR_STRING_OPEN	TOKEN_END_TAG	TOKEN_SELF_CLOSING_TAG	TOKEN_IDENTIFICATEUR	#
V	1	ERREUR	ERREUR	ERREUR	ERREUR	ERREUR	2
E	ERREUR	ERREUR	3	ERREUR	ERREUR	ERREUR	ERREUR
E1	ERREUR	ERREUR	ERREUR	4	5	ERREUR	ERREUR
E2	ERREUR	7	6	ERREUR	ERREUR	ERREUR	ERREUR
A	ERREUR	ERREUR	ERREUR	ERREUR	ERREUR	8	ERREUR

La grammaire est essentiellement LL(1) car, pour chaque non-terminal, on n'a pas de chevauchement des différentes productions, la table est donc mono-définie.

2.2. Méthode choisie : description et justification

Pour l'analyse syntaxique, nous avons opté pour une analyse descendante déterministe par descente récursive. Cette méthode consiste à représenter chaque non-terminal de la grammaire par une fonction dans le code. À chaque fonction correspond un non-terminal, et les fonctions s'appellent récursivement pour traiter les symboles non-terminaux. Lorsqu'un terminal est attendu, il est simplement vérifié dans le flot d'entrée.

La descente récursive est particulièrement adaptée à notre grammaire car elle est LL(1), c'est-à-dire que chaque choix entre plusieurs règles de production peut être résolu avec un seul symbole. Cette méthode offre une implémentation simple et lisible, sans nécessiter de pile explicite pour gérer les états comme dans d'autres techniques d'analyse syntaxique.

2.3. Algorithme d'implémentation en C

Le code présenté implémente un analyseur syntaxique descendant déterministe pour le sous-langage de HTPL dédié à la déclaration de variables de type String. Il repose sur la méthode de descente récursive, avec chaque règle de la grammaire modélisée par une fonction en C.

2.4. Tests et validation

Le code présenté implémente un analyseur syntaxique descendant déterministe pour le sous-langage de HTPL dédié à la déclaration de variables de type String. Il repose sur la méthode de descente récursive, avec chaque règle de la grammaire modélisée par une fonction en C.

2.4.1. Test 1 : Succès

```
<variables>
    <string name="unTexte"> "HelloWorld" </string>
    <string name="unTexte2"> "HelloWorld" </string>
</variables>
```

Qui devient après analyse lexicale:

```
VARIABLES_OPEN VAR_STRING_OPEN IDENTIFICATEUR ASSIGN STRING END_TAG
STRING VAR_STRING_CLOSE VAR_STRING_OPEN IDENTIFICATEUR ASSIGN STRING
END_TAG STRING VAR_STRING_CLOSE VARIABLES_CLOSE #
```

Après analyse syntaxique, nous avons :

```
Résultat de l'analyse syntaxique de VARIABLES_OPEN VAR_STRING_OPEN
IDENTIFICATEUR ASSIGN STRING END_TAG STRING VAR_STRING_CLOSE
VAR_STRING_OPEN IDENTIFICATEUR ASSIGN STRING END_TAG STRING
VAR_STRING_CLOSE VARIABLES_CLOSE # :
-----
Statut: Succès
Message: Analyse syntaxique terminée avec succès
```

2.4.2. Test 2 : Échec

```
<variables>
<variables>
    <string name="unTexte"> "HelloWorld" </string>
    <string name="unTexte2"> "HelloWorld" </string>
</variables>
```

Qui devient après analyse lexicale:

```
VARIABLES_OPEN VARIABLES_OPEN VAR_STRING_OPEN IDENTIFICATEUR ASSIGN
STRING END_TAG STRING VAR_STRING_CLOSE VAR_STRING_OPEN IDENTIFICATEUR
ASSIGN STRING END_TAG STRING VAR_STRING_CLOSE VARIABLES_CLOSE #
```

Après analyse syntaxique, nous avons :

```
ERREUR SYNTAXIQUE: TOKEN_VAR_STRING_OPEN attendu
```

3. Implémentation avec BISON

3.1. Introduction à l'outil BISON

BISON est un générateur de parseur, similaire à YACC (Yet Another Compiler Compiler), utilisé pour analyser la structure syntaxique des entrées (comme du code source) en fonction d'une grammaire définie.

Utilisation principale : Création de compilateurs, interpréteurs, analyseurs de requêtes, etc.

3.2. Spécification des règles de grammaire dans le fichier BISON

3.2.1 Définitions de symboles terminaux (*tokens*):

```
/* Token declarations */
%token TOKEN_UNRECOGNIZED
%token TOKEN_PROGRAM_OPEN TOKEN_PROGRAM_CLOSE
%token TOKEN_VARIABLES_OPEN TOKEN_VARIABLES_CLOSE
%token TOKEN_INSTRUCTIONS_OPEN TOKEN_INSTRUCTIONS_CLOSE
%token TOKEN_ASSIGN_OPEN TOKEN_ASSIGN_CLOSE
%token TOKEN_PRINT_OPEN TOKEN_PRINT_CLOSE
%token TOKEN_IF_OPEN TOKEN_IF_CLOSE TOKEN_ELSE
%token TOKEN_WHILE_OPEN TOKEN_WHILE_CLOSE
%token TOKEN_ARRAY_OPEN TOKEN_ARRAY_CLOSE
%token TOKEN_ELEMENT_OPEN TOKEN_END_TAG TOKEN_SELF_CLOSING_TAG
%token TOKEN_VAR_INT_OPEN TOKEN_VAR_INT_CLOSE
%token TOKEN_VAR_FLOAT_OPEN TOKEN_VAR_FLOAT_CLOSE
%token TOKEN_VAR_BOOLEAN_OPEN TOKEN_VAR_BOOLEAN_CLOSE
%token TOKEN_VAR_STRING_OPEN TOKEN_VAR_STRING_CLOSE
%token TOKEN_EXPRESSION TOKEN_STRING
%token TOKEN_PLUS TOKEN_MINUS TOKEN_MULTIPLY TOKEN_DIVIDE
%token TOKEN_GREATER_THAN TOKEN_LOWER_THAN
%token TOKEN_GREATER_OR_EQUAL TOKEN_LOWER_OR_EQUAL TOKEN_EQUAL
%token TOKEN_OPEN_PARENTHESIS TOKEN_CLOSE_PARENTHESIS
%token TOKEN_ASSIGN TOKEN_QUOTE
%token IDENTIFICATEUR TOKEN_INT TOKEN_FLOAT TOKEN_BOOLEAN
```

3.2.2 Règles de grammaire:

Règles de grammaire:

```
program:
    TOKEN_PROGRAM_OPEN
    variables_list
    TOKEN_INSTRUCTIONS_OPEN instruction_list TOKEN_INSTRUCTIONS_CLOSE
    TOKEN_PROGRAM_CLOSE
;
```

la list des variables et des declarations

```
variables_list:
| TOKEN_VARIABLES_OPEN declaration_list TOKEN_VARIABLES_CLOSE
| /* empty */

declaration_list:
| TOKEN_VAR_INT_OPEN attributes TOKEN_END_TAG expr_arithmetique TOKEN_VAR_INT CLOSE declaration_list
| TOKEN_VAR_FLOAT_OPEN attributes TOKEN_END_TAG expr_arithmetique TOKEN_VAR_FLOAT CLOSE declaration_list
| TOKEN_VAR_STRING_OPEN attributes TOKEN_END_TAG TOKEN_STRING TOKEN_VAR_STRING CLOSE declaration_list
| TOKEN_VAR_BOOLEAN_OPEN attributes TOKEN_END_TAG expr_logique TOKEN_VAR_BOOLEAN CLOSE declaration_list
| TOKEN_ARRAY_OPEN attributes TOKEN_END_TAG elements TOKEN_ARRAY CLOSE declaration_list
| TOKEN_VAR_INT_OPEN attributes TOKEN_END_TAG expr_arithmetique TOKEN_VAR_INT CLOSE
| TOKEN_VAR_FLOAT_OPEN attributes TOKEN_END_TAG expr_arithmetique TOKEN_VAR_FLOAT CLOSE
| TOKEN_VAR_STRING_OPEN attributes TOKEN_END_TAG TOKEN_STRING TOKEN_VAR_STRING CLOSE
| TOKEN_VAR_BOOLEAN_OPEN attributes TOKEN_END_TAG expr_logique TOKEN_VAR_BOOLEAN CLOSE
| TOKEN_ARRAY_OPEN attributes TOKEN_END_TAG elements TOKEN_ARRAY CLOSE
| TOKEN_VAR_INT_OPEN attributes TOKEN_SELF_CLOSING_TAG declaration_list
| TOKEN_VAR_FLOAT_OPEN attributes TOKEN_SELF_CLOSING_TAG declaration_list
| TOKEN_VAR_STRING_OPEN attributes TOKEN_SELF_CLOSING_TAG declaration_list
| TOKEN_VAR_BOOLEAN_OPEN attributes TOKEN_SELF_CLOSING_TAG declaration_list
| TOKEN_ARRAY_OPEN attributes TOKEN_SELF_CLOSING_TAG declaration_list
| TOKEN_VAR_INT_OPEN attributes TOKEN_SELF_CLOSING_TAG
| TOKEN_VAR_FLOAT_OPEN attributes TOKEN_SELF_CLOSING_TAG
| TOKEN_VAR_STRING_OPEN attributes TOKEN_SELF_CLOSING_TAG
| TOKEN_VAR_BOOLEAN_OPEN attributes TOKEN_SELF_CLOSING_TAG
| TOKEN_ARRAY_OPEN attributes TOKEN_SELF_CLOSING_TAG
;
```

listes des attributs des variables

```
attributes:
| IDENTIFICATEUR TOKEN_ASSIGN TOKEN_STRING attributes
| IDENTIFICATEUR TOKEN_ASSIGN TOKEN_OPEN_PARENTHESIS expr_arithmetique TOKEN_CLOSE_PARENTHESIS attributes
| IDENTIFICATEUR TOKEN_ASSIGN TOKEN_OPEN_PARENTHESIS expr_logique TOKEN_CLOSE_PARENTHESIS attributes
| IDENTIFICATEUR TOKEN_ASSIGN TOKEN_STRING
| IDENTIFICATEUR TOKEN_ASSIGN TOKEN_OPEN_PARENTHESIS expr_arithmetique TOKEN_CLOSE_PARENTHESIS
| IDENTIFICATEUR TOKEN_ASSIGN TOKEN_OPEN_PARENTHESIS expr_logique TOKEN_CLOSE_PARENTHESIS
;
```

declarations des elements dun tableau

```
elements:
| element elements | /* void */
;

element:
| TOKEN_ELEMENT_OPEN attributes TOKEN_SELF_CLOSING_TAG
;

instruction_list:
| instruction_list instruction
| /* empty */
;
```

la partie instructions (ou on va commencer reelment a programmer)

```

instruction_list:
| instruction_list instruction
| /* empty */
;

instruction:
assignment
| if_statement
| while_statement
| print_statement
;

```

l'affectation , les condition , les boucles et affichage respectivement

```

assignment:
TOKEN_ASSIGN_OPEN attributes TOKEN_SELF_CLOSING_TAG
;

if_statement:
TOKEN_IF_OPEN attributes TOKEN_END_TAG
instruction_list
TOKEN_IF_CLOSE
| TOKEN_IF_OPEN attributes TOKEN_END_TAG
instruction_list
TOKEN_ELSE
instruction_list
TOKEN_IF_CLOSE
;

while_statement:
TOKEN WHILE_OPEN attributes TOKEN_END_TAG
instruction_list
TOKEN WHILE_CLOSE
;

print_statement:
TOKEN_PRINT_OPEN attributes TOKEN_SELF_CLOSING_TAG
;

```

les expressions arithmétiques (traitant la priorité avec le fix du grammaire)

```

expr_arithmetique:
    terme
    | expr_arithmetique TOKEN_PLUS terme
    | expr_arithmetique TOKEN_MINUS terme
    ;

terme:
    facteur
    | terme TOKEN_MULTIPLY facteur
    | terme TOKEN_DIVIDE facteur
    ;

facteur:
    TOKEN_INT
    | TOKEN_FLOAT
    | IDENTIFICATEUR
    | TOKEN_OPEN_PARENTHESIS expr_arithmetique TOKEN_CLOSE_PARENTHESIS
    ;

```

les expression logique :

```

expr_logique:
    expr_arithmetique TOKEN_EQUAL expr_arithmetique
    | expr_arithmetique TOKEN_GREATER_THAN expr_arithmetique
    | expr_arithmetique TOKEN_LOWER_THAN expr_arithmetique
    | expr_arithmetique TOKEN_GREATER_OR_EQUAL expr_arithmetique
    | expr_arithmetique TOKEN_LOWER_OR_EQUAL expr_arithmetique
    | TOKEN_OPEN_PARENTHESSES expr_logique TOKEN_CLOSE_PARENTHESSES
    | TOKEN_BOOLEAN

```

3.3. Règles d'associativité et de priorités

on a géré les priorités des opérateurs directement dans la grammaire, en définissant des règles de production spécifiques qui respectent l'ordre de priorité. Cela nous a permis d'éviter d'utiliser les directives %left et %right, et ainsi d'avoir un contrôle plus précis sur la structure de l'analyse syntaxique.

4. Détection et affichage des erreurs

La détection des erreurs syntaxiques est effectuée automatiquement par Bison. Cependant, Bison ne fournit pas la position exacte de l'erreur dans le fichier source. Pour cela, nous avons défini et utilisé plusieurs éléments.

1. `yysuccess(char* s)` : Fonction utilisée pour mettre à jour la position dans la colonne. Elle est appelée à chaque token reconnu dans le lexer afin de mettre à jour la variable `currentColumn` et ainsi suivre la position exacte du token dans le fichier source.
2. `yyerror(const char *s)` : Fonction utilisée pour afficher les messages d'erreur générés par Bison. Elle est appelée lorsque le parser rencontre une erreur de syntaxe.
3. `currentColumn` : Variable globale qui suit la colonne actuelle du lexer dans le fichier source. Elle est utilisée pour préciser la position exacte de l'erreur dans les messages d'erreur.

Ces éléments sont définis dans le fichier `syntaxique.c` (Bison) et utilisés dans le lexer de la manière suivante :

- `yysuccess(yytext)` : Appelée à chaque token détecté pour informer le parser que le token a été reconnu et pour suivre la position dans le fichier source.
- Gestion de la colonne actuelle :
 - `[\n\r]+ { currentColumn = 1; }` : Réinitialise la colonne à 1 après un saut de ligne.
 - `[\t]+ { currentColumn += yyleng; }` : Incrémente `currentColumn` de la longueur du token pour les espaces et tabulations.

Enfin, dans le parser, la fonction `yyerror` affiche un message d'erreur au format :

5. Tests et Validation

exemple de programme analysé

Pour tester et valider le fonctionnement de notre analyseur syntaxique, nous avons effectué deux types de tests : un test avec une entrée correcte et un test avec une erreur de syntaxe. Voici des exemples :

Exemple sans erreur (entrée valide) :

- Entrée : Un fichier source valide, respectant la grammaire définie.

```
<program>
<variables>
    <int name="counter">(0+8-5)</int>
    <string name="message">"Test"</string>
</variables>

<instructions>
    <assign counter=(counter - 1)/>
    <print value="Counter initialized to 10"/>

    <while condition=(counter >> 0)>
        <print value="Counter: + counter"/>
        <assign counter=(counter - 1)/>
    </while>

    <if condition=(counter == 0)>
        <print value="Counter reached zero"/>
    </if>
</instructions>
</program>
```

- **Résultat attendu** : Le parser détecte tous les tokens correctement sans afficher de message d'erreur.
- **Résultat obtenu** : Le programme s'exécute sans afficher de messages d'erreur, et indique “Parsing successful”

```

Token: 302, Content: counter
Token: 297, Content: ==
Token: 303, Content: 0
Token: 299, Content: )
Token: 277, Content: >
Token: 267, Content: <print
Token: 302, Content: value
Token: 300, Content: =
Token: 288, Content: "Counter reached zero"
Token: 278, Content: />
Token: 270, Content: </if>
Token: 264, Content: </instructions>
Token: 260, Content: </program>
Parsing successful
```

Exemple avec erreur (entrée invalide) :

- **Entrée** : Un fichier source contenant une erreur de syntaxe , le programme et le même que le précédent , mais la valeur de l'attribut et sans guillemets.

```

variables>
    <int name=counter>(0+
        string name="message"
```

- **Résultat attendu** : Le parser détecte l'erreur et affiche un message d'erreur contenant la ligne, la colonne et une description de l'erreur.
- **Résultat obtenu** : Le programme affiche un message d'erreur sous la forme suivante :

```

Token: 302, Content: name
Token: 300, Content: =
Token: 302, Content: counter
File output, line 1, character 26 :  syntax error
Parsing failed
```

6. Passage des valeurs des tokens entre le lexer et Bison

Dans le fichier de bison (syntaxique.c) nous avons défini une union pour les types des terminaux et non-terminaux, en incluant uniquement les terminaux dont nous avons besoin pour notre analyse. Cette union est partagée entre le lexer et le fichier Bison, permettant ainsi de passer les valeurs du lexer au parser via la variable yylval.

```
{float}          { yylval.floatVal = atof(yytext); current_token = T_FLOAT }
{int}           { yylval.intValue = atoi(yytext); current_token = T_INT }
{boolean}        { yylval.boolVal = (strcmp(yytext, "true") == 0); }
```

Ces valeurs sont ensuite utilisées pour peupler les quadruplets lors de la génération du code intermédiaire.

```
%token <strVal> IDENTIFICATEUR
%token <intValue> TOKEN_INT
%token <floatVal> TOKEN_FLOAT
%token <boolVal> TOKEN_BOOLEAN

/* Type declarations for non-terminals */
%type <attr> attributes
%type <intValue> expr_arithmetique terme facteur
%type <boolVal> expr_logique
```

7. Conclusion

En conclusion, cette étude s'inscrit dans le cadre d'un projet visant à analyser et traiter un langage spécifique, HTPL, un langage basé sur des balises. Bien que ce choix offre une grande flexibilité et modularité, il introduit également une complexité importante dans la conception et l'analyse.

L'analyse lexicale a permis d'identifier et de classer les éléments du langage, tandis que l'analyse syntaxique a joué un rôle crucial en validant la structure grammaticale et en décryptant les relations entre les balises. Cependant, la nature évolutive de HTPL, due à des modifications fréquentes dans ses spécifications, a rendu chaque étape d'analyse particulièrement difficile. Chaque ajustement dans la structure du langage a nécessité une révision des règles syntaxiques et des algorithmes de traitement, augmentant la complexité globale du projet.

Ces étapes, combinées, nous ont fourni des outils solides pour mieux comprendre la syntaxe de notre corpus et ouvrir la voie à des traitements plus avancés, tels que l'analyse sémantique ou la génération de code. Le travail accompli dans cette étude constitue donc un fondement essentiel pour les étapes suivantes du projet, renforçant notre capacité à interpréter, valider et manipuler efficacement le langage construit.

Analyse sémantique

1. Introduction

Dans le cadre de l'informatique moderne, la compilation joue un rôle central dans la transformation des idées programmatiques en instructions exécutables par une machine. Ce processus s'appuie sur une série d'étapes interdépendantes : l'analyse lexicale, syntaxique et sémantique. Après avoir réalisé l'analyse lexicale, qui décompose le code source en tokens structurés, et l'analyse syntaxique, qui valide la conformité des séquences de tokens aux règles grammaticales du langage, nous abordons dans ce rapport la phase suivante : l'analyse sémantique.

L'analyse sémantique occupe une place essentielle dans le processus de compilation. Elle garantit que les programmes, bien que syntaxiquement valides, respectent également les règles sémantiques du langage, telles que la compatibilité des types, la déclaration préalable des variables et la validité logique des opérations. Cette étape agit comme un filtre logique qui vérifie la cohérence interne des programmes et prépare le terrain pour une génération de code intermédiaire efficace et exempte d'erreurs.

Ce rapport explore les fondements théoriques et les applications pratiques de l'analyse sémantique dans le contexte de la conception d'un compilateur pour le langage HTPL.

De plus, ce document présente en détail les mécanismes de génération de code intermédiaire. À travers des exemples concrets et des solutions techniques avancées, il illustre les défis rencontrés, notamment la gestion des structures imbriquées et la résolution des priorités d'opérateurs, tout en proposant des réponses adaptées.

Ainsi, ce rapport s'inscrit dans la continuité du travail réalisé sur les phases lexicale et syntaxique, en élargissant notre compréhension des principes fondamentaux de la compilation.

2. Structure des Données

2.1. Table des Symboles

La table des symboles est une structure fondamentale qui maintient les informations sur toutes les variables déclarées dans le programme.

```

#define TABLE_SIZE 1000

typedef struct ArrayInfo {
    DataType elementType;
    int size;
} ArrayInfo;

typedef struct SymbolEntry {
    char *name;
    DataType type;
    union {
        ArrayInfo arrayInfo;
    } details;
    struct SymbolEntry *next;
} SymbolEntry;

typedef struct {
    SymbolEntry *entries[TABLE_SIZE];
} SymbolTable;

```

Elle contient trois champs principaux :

- Nom : le nom de la variable.
- Type : le type de la variable (entier, flottant, caractère, tableau, etc.).
- Détails : un champ utilisé principalement pour les tableaux, spécifiant le type des éléments (entier, flottant, etc.) et leur taille.

Les attributs détails sont particulièrement importants pour les tableaux, car ils permettent de :

1. Vérifier les limites : éviter les accès hors des bornes du tableau.

Garantir la cohérence des types : s'assurer que les opérations effectuées sur les éléments sont compatibles avec leur type.

Table des symboles :

Nom	Type	Details
message	chaine	
numbers	tableau	Type: entier, Taille: 10
counter	entier	

Grâce à son implémentation en hashmap, la table des symboles offre une recherche rapide des variables, améliorant ainsi l'efficacité du programme tout en renforçant sa fiabilité et sa sécurité.

2.2 Gestion de la table

La table des symboles est manipulée à l'aide d'un ensemble de fonctions définies dans tableSymbole.h et implémentées dans tableSymbole.c, facilitant ainsi son utilisation et sa maintenance.

Fonctions disponibles :

1. void initSymbolTable(SymbolTable *table): Initialise la table des symboles, en préparant sa structure pour l'ajout de nouvelles entrées.
- 2.
3. bool addSymbol(SymbolTable *table, const char *name, DataType type): Ajoute une nouvelle entrée dans la table des symboles avec le nom et le type spécifiés. Retourne true si l'ajout est réussi, ou false si l'entrée existe déjà.
- 4.
5. SymbolEntry *findSymbol(SymbolTable *table, const char *name): Recherche une entrée dans la table par son nom et retourne un pointeur vers cette entrée, ou NULL si elle n'existe pas.
- 6.
7. bool removeSymbol(SymbolTable *table, const char *name): Supprime une entrée de la table par son nom. Retourne true si la suppression est réussie, ou false si l'entrée n'existe pas.
8. bool updateSymbolValue(SymbolTable *table, const char *name, SymbolDetails details): Met à jour le champ details d'une entrée de la table par son nom. Cette fonction est essentielle pour affecter des informations supplémentaires (comme le type d'éléments et la taille des tableaux). Retourne true si la mise à jour est réussie, ou false si l'entrée n'existe pas.
6. void printSymbolTable(SymbolTable *table): Affiche le contenu de la table des symboles sous forme de tableau lisible. Cette fonction est utile pour le débogage et la validation.
7. void freeSymbolTable(SymbolTable *table): Libère la mémoire allouée pour la table des symboles et ses entrées, évitant ainsi les fuites de mémoire.

2.3 Rôle de la Table des Symboles

Comme indiqué déjà la table de symbole permet de garantir la validité et la cohérence du code source. Voici ses principales utilités :

1. Vérification des types des opérandes lors d'une opération : Lorsqu'une opération est effectuée (addition, comparaison, etc.), la table des symboles permet de vérifier si les types des opérandes sont compatibles. Par exemple, l'ajout d'un entier à une chaîne de caractères est une erreur que la table des symboles peut détecter, en analysant les types des variables impliquées.
2. Vérification de la déclaration des variables : Avant d'utiliser une variable dans le code, il est essentiel qu'elle ait été déclarée. La table des symboles permet de rechercher rapidement si une variable existe déjà, évitant ainsi les erreurs comme l'utilisation de variables non déclarées.
3. Détection de la redéclaration des variables : Une variable ne peut pas être déclarée deux fois. La table des symboles permet d'identifier une tentative de redéclaration en vérifiant si une variable du même nom existe déjà.

- Vérification des accès "index out of bound" pour les tableaux : Le champ details de la table des symboles est crucial pour les tableaux. Il contient des informations sur la taille et le type des éléments du tableau. Lorsqu'un tableau est référencé avec un index, la table permet de vérifier si cet index est dans les limites autorisées.

3. Vérifications Sémantiques

3.1 Vérification des Types

La vérification des types est une étape cruciale dans l'analyse sémantique pour garantir la validité des données manipulées par le programme. Cela se fait dans plusieurs contextes :

3.1.1 Lors de la déclaration des variables :

Lorsqu'une variable est déclarée, il est important de vérifier que le type de la variable correspond à la valeur qui lui est affectée. Cela permet d'assurer que la déclaration est cohérente, voici l'exemple:

```

declaration_list:
    TOKEN_VAR_INT_OPEN attributes TOKEN_END_TAG expr_arithmetique TOKEN_VAR_INT_CLOSE {
        addSymbol(&symbolTable, $2.attrs[0].name, TYPE_INTEGER);
        char temp[15];
        sprintf(temp, "%s", $4);
        if (!isInteger(temp) && !isVariable(temp)) {
            yyerror("Invalid value for integer variable");
        }
        createQuad(":=", temp, "", $2.attrs[0].name);
    }
    declaration_list

| TOKEN_VAR_FLOAT_OPEN attributes TOKEN_END_TAG expr_arithmetique TOKEN_VAR_FLOAT_CLOSE {
    addSymbol(&symbolTable, $2.attrs[0].name, TYPE_FLOAT);
    char temp[15];
    sprintf(temp, "%s", $4);
    if (!isFloat(temp) && !isInteger(temp) && !isVariable(temp)) {
        yyerror("Invalid value for float variable");
    }
    createQuad(":=", temp, "", $2.attrs[0].name);
}
declaration_list

| TOKEN_VAR_STRING_OPEN attributes TOKEN_END_TAG TOKEN_STRING TOKEN_VAR_STRING_CLOSE {
    addSymbol(&symbolTable, $2.attrs[0].name, TYPE_STRING);
    char temp[15];
    sprintf(temp, "%s", $4);
    if (!isString(temp)) {
        yyerror("Invalid value for string variable");
    }
    createQuad(":=", temp, "", $2.attrs[0].name);
}
declaration_list;

```

3.1.2 Vérification des types lors d'une affectation :

Lorsqu'une valeur est affectée à une variable, il est essentiel de vérifier que le type de la valeur est compatible avec le type de la variable. Cela garantit que l'opération est valide, voici l'exemple:

```
else { // normal variable
    entry = findSymbol(&symbolTable, $2.attrs[0].name);
    if (!entry) {
        yyerror("Variable undefined");
    }
}

char temp[15];
sprintf(temp, "%s", $2.attrs[0].value);

if (entry->type == TYPE_INTEGER) {
    if (!isInteger(temp) && !isVariable(temp)) {
        yyerror("Invalid value for integer variable");
    }
} else if (entry->type == TYPE_FLOAT) {
    if (!isFloat(temp) && !isInteger(temp) && !isVariable(temp)) {
        yyerror("Invalid value for float variable");
    }
} else if (entry->type == TYPE_STRING) {
    if (!isString(temp)) {
        yyerror("Invalid value for string variable");
    }
} else if (entry->type == TYPE_BOOLEAN) {
    if (!isBoolean(temp) && !isVariable(temp)) {
        yyerror("Invalid value for boolean variable");
    }
}

// TODO: handle array type and array reference type
createQuad(":=", temp, "", $2.attrs[0].name);
```

3.1.3 Dans le cas des opérations arithmétiques et logiques :

Pour les opérations arithmétiques ou logiques, la vérification explicite des types n'est pas nécessaire, car les règles du langage et la réduction des tokens par Bison garantissent automatiquement que les types sont corrects.

3.2 Existence variable

Lorsqu'une variable ou une référence est utilisée hors de la section de déclaration, il est crucial de vérifier son existence dans la table des symboles. Cela permet de s'assurer que toute utilisation d'une variable ou d'un tableau est valide et conforme aux règles du langage, voici l'exemple:

```
assignment:
TOKEN_ASSIGN_OPEN attributes TOKEN_SELF_CLOSING_TAG {
    // Update the value of a variable
    SymbolEntry* entry;
    if (isArrayReference($2.attrs[0].name)) {
        char* arrayName = getArrayName($2.attrs[0].name);
        int index = getArrayIndex($2.attrs[0].name);
        entry = findSymbol(&symbolTable, arrayName);
        if (!entry) {
            yyerror("Array undefined");
        }
        if (entry->type != TYPE_ARRAY) {
            yyerror("Variable is not an array");
        }
        free(arrayName);
    } else { // normal variable
        entry = findSymbol(&symbolTable, $2.attrs[0].name);
        if (!entry) {
            yyerror("Variable undefined");
        }
    }
}
```

De plus ,La vérification de l'existence d'une variable au moment de sa déclaration est essentielle pour éviter les redondances les redéclarations multiples d'une même variable, voici l'exemple:

```
declaration_list:
TOKEN_VAR_INT_OPEN attributes TOKEN_END_TAG expr_arithmetique TOKEN_VAR_INT_CLOSE {
    if (findSymbol(&symbolTable, $2.attrs[0].name)) {
        yyerror("Variable already declared");
    } else {
        addSymbol(&symbolTable, $2.attrs[0].name, TYPE_INTEGER);
        char temp[15];
        sprintf(temp, "%s", $4);

        if (!isInteger(temp) && !isVariable(temp)) {
            yyerror("Invalid value for integer variable");
        }
        createQuad(":=", temp, "", $2.attrs[0].name);
    }
}
declaration_list;
```

3.3 Verification des Tableaux

La vérification inclut l'indice de la référence et la vérification si la variable est en effet de type tableau ce qui garantit que les indices utilisés pour accéder aux éléments d'un tableau sont valides, ce qui évite les erreurs d'exécution liées à un accès mémoire incorrect, voici l'exemple:

```
array_reference:
IDENTIFICATEUR TOKEN_OPEN_BRACKET expr_arithmetique TOKEN_CLOSE_BRACKET {
    char temp[15];
    sprintf(temp, sizeof(temp), "%s[%s]", $1, $3);
    $$ = strdup(temp);

    SymbolEntry *entry = findSymbol(&symbolTable, $1);
    if (!entry) {
        yyerror("Array undefined");
    }

    if (entry->type != TYPE_ARRAY) {
        yyerror("Variable is not an array");
    }

    int index = atoi($3);
    if (index < 0 || index >= entry->details.arraySize) {
        yyerror("Array index out of bounds");
    }
};
```

3.4 Division sur 0

Notre compilateur vérifie la division par zéro uniquement lorsque l'opérande est explicitement '0'. Si l'opérande est une variable avec une valeur de zéro, cela sera détecté uniquement au moment de l'exécution, voici l'exemple:

```
terme TOKEN_DIVIDE facteur {
    if (strcmp($3, "0") == 0) {
        yyerror("Division by zero error");
        YYABORT;
    }

    char temp[15];
    sprintf(temp, "T%d", ti++);
    createQuad("/", $1, $3, temp);
    $$ = strdup(temp);
};
```

4. Génération de Code Intermédiaire

Dans le compilateur HTPL, les instructions intermédiaires sont représentées sous forme de quadruplets. Un quadruplet est une structure qui contient quatre champs : l'opération, les deux opérandes et le résultat. Ces quadruplets sont stockés dans un tableau pour une gestion facile.

4.1 Structure des Quadruplets

Ils sont stockés dans un tableau global quad de taille 1000. Chaque quadruplet contient quatre champs : l'opération, les deux opérandes et le résultat.

```
int QC = 0;
int TI = 0;

typedef struct quadruplet {
    char op[15];
    char opr1[15];
    char opr2[15];
    char res[15];
} Quad;

Quad quad[1000];
```

les variables gloables : QC est le Pointeur des quadruplets , et TI est le numéro de la variable intermédiaire Ti

4.2 Gestion des quadruplets

on a utilise 2 Fonctions pour gérer les quadruplets :

1. Création de quadruplet :

```
void createQuad(char* op, char* opr1, char* opr2, char* res) {
    strcpy(quad[QC].op, op);
    strcpy(quad[QC].opr1, opr1);
    strcpy(quad[QC].opr2, opr2);
    strcpy(quad[QC].res, res);
    QC++;
}
```

Cette fonction crée un nouveau quadruplet avec les opérandes et l'opération spécifiés, puis l'ajoute au tableau des quadruplets.

2. Affichage des quadruplets :

```
void printQuad() {
    printf("\n== Quadruplets ==\n");
    for (int i = 0; i < QC; i++) {
        printf("%d- (%s, %s, %s, %s)\n", i, quad[i].op, quad[i].opr1, quad[i].opr2, quad[i].res);
    }
    printf("=====\\n");
}
```

Cette fonction affiche tous les quadruplets générés jusqu'à présent.

4.3 Gestion des étiquettes et des boucles imbriquées

Pour gérer les boucles imbriquées et les structures conditionnelles imbriquées (if-else), des piles sont utilisées pour stocker les adresses des étiquettes.

```
int sauv_begin_if[100];
int sauv_fin_if[100];
int sauv_fin_else[100];
int sauv_begin_while[100];
int sauv_fin_while[100];

int top_begin_if = -1;
int top_fin_if = -1;
int top_fin_else = -1;
int top_begin_while = -1;
int top_fin_while = -1;

void push(int stack[], int *top, int value) {
    stack[++(*top)] = value;
}

int pop(int stack[], int *top) {
    if (*top == -1) {
        fprintf(stderr, "Stack underflow\\n");
        exit(EXIT_FAILURE);
    }
    return stack[(*top)--];
}
```

Ces piles permettent de gérer les adresses des étiquettes pour les structures de contrôle imbriquées

4.4 Exemples de génération de quadruplets : while et if-else

1. Génération de quadruplets pour if-else

```
if_statement:
    if_condition TOKEN_END_TAG instruction_list {
        char temp[15];
        sprintf(temp, "%d", QC);
        strcpy(quad[sauv_begin_if[top_begin_if--]].opr1, temp);

        TOKEN_IF_CLOSE {
            sauv_fin_if[++top_fin_if] = QC;
            createQuad("BR", "", "", "");
        }

        char temp[15];
        sprintf(temp, "%d", QC);
        strcpy(quad[sauv_begin_if[top_begin_if--]].opr1, temp);
    }

    TOKEN_ELSE instruction_list TOKEN_IF_CLOSE {
        char temp[15];
        sprintf(temp, "%d", QC);
        strcpy(quad[sauv_fin_if[top_fin_if--]].opr1, temp);
    }

if_condition:
    TOKEN_IF_OPEN attributes {
        if (strcmp($2.attrs[0].name, "condition") == 0) {
            sauv_begin_if[+top_begin_if] = QC;
            char temp[15];
            sprintf(temp, "%s", $2.attrs[0].value);
            createQuad("BZ", "", "", temp);
        } else {
            yyerror("Invalid attribute for if statement");
        }
    }
```

Explication:

1. *if_condition* :

Lorsqu'une condition if est rencontrée, un quadruplet BZ (branch if zero) est créé avec la condition. L'adresse de ce quadruplet est sauvegardée dans sauv_begin_if pour mise à jour ultérieure.

- *if_statement* :

Après l'exécution de instruction_list dans le bloc if, l'adresse de l'instruction suivante est mise à jour dans le quadruplet BZ. Si une clause else est présente, un quadruplet BR (branch) est créé pour sauter le bloc else. L'adresse de ce quadruplet est sauvegardée dans sauv_fin_if pour mise à jour après le bloc else. Après l'exécution de instruction_list dans le bloc else, l'adresse de l'instruction suivante est mise à jour dans le quadruplet BR.

2. Génération de quadruplets pour while

```
while_statement:
    while_condition TOKEN_END_TAG instruction_list TOKEN WHILE CLOSE {
        int begin_while = sauv_begin_while[top_begin_while--];
        char temp[15];
        sprintf(temp, "%d", begin_while);
        createQuad("BR", temp, "", "");

        sprintf(quad[begin_while].opr1, "%d", QC);
    }

while_condition:
    TOKEN WHILE OPEN attributes {
        if (strcmp($2.attrs[0].name, "condition") == 0) {
            sauv_begin_while[+top_begin_while] = QC;
            char temp[15];
            sprintf(temp, "%s", $2.attrs[0].value);
            createQuad("BZ", "", "", temp);
        } else {
            yyerror("Invalid attribute for if statement");
        }
    }
```

Explication:

1. *while_condition* :

Lorsqu'une condition while est rencontrée, un quadruplet BZ (branch if zero) est créé avec la condition. L'adresse de ce quadruplet est sauvegardée dans sauv_begin_While pour mise à jour ultérieure.

• *while_statement* :

Après l'exécution de instruction_list dans le bloc while, un quadruplet BR (branch) est créé pour revenir au début de la boucle. L'adresse de ce quadruplet est mise à jour avec l'adresse de la condition while. L'adresse de l'instruction suivante après la boucle est mise à jour dans le quadruplet BZ.

4.5 Exemples Complets de génération de quadruplets

Input :

```
<program>
    <variables>
        <int name="counter">(3)</int>
        <string name="message">"Test"</string>
        <array name="numbers" type="int" size="5">
            <element value="1"/>
            <element value="2"/>
        </array>
    </variables>

    <instructions>
        <print value="Counter initialized to 10"/>

        <while condition="(counter > 0)">
            <assign numbers[2]=(numbers[1] + 1)/>
            <print value="Counter: " + counter"/>
        </while>

        <if condition="(counter == 0)">
            <assign counter="(3 - 1)"/>
            <assign counter="(4 - 1)"/>
        <else>
            <assign counter="(counter + 1)"/>
            <assign counter="(counter + 1)"/>
        </if>
    </instructions>
</program>
```

Output :

```
== Quadruplets ==
0- (:=, 3, , counter)
1- (:=, "Test", , message)
2- (Bounds, 1, 5, )
3- (ADEC, numbers, , )
4- (:=, 1, , numbers[1])
5- (:=, 2, , numbers[2])
6- (>, counter, 0, T0)
7- (BZ, 11, , T0)
8- (+, numbers[1], 1, T1)
9- (:=, T1, , numbers[2])
10- (print, "Counter: ", counter)
11- (BR, 7, , )
12- (==, counter, 0, T2)
13- (BZ, 18, , T2)
14- (-, 3, 1, T3)
15- (:=, T3, , counter)
16- (-, 4, 1, T4)
17- (:=, T4, , counter)
18- (BR, 22, , )
19- (+, counter, 1, T5)
20- (:=, T5, , counter)
21- (+, counter, 1, T6)
22- (:=, T6, , counter)
=====
```

5. Gestion des Erreurs

La détection des erreurs syntaxiques est effectuée automatiquement par Bison. Cependant, Bison ne fournit pas la position exacte de l'erreur dans le fichier source. Pour cela, nous avons défini et utilisé plusieurs éléments:

- `yysuccess(char* s)` : Fonction utilisée pour mettre à jour la position dans la colonne. Elle est appelée à chaque token reconnu dans le lexer afin de mettre à jour la variable `currentColumn` et ainsi suivre la position exacte du token dans le fichier source.
- `yyerror(const char *s)` : Fonction utilisée pour afficher les messages d'erreur générés par Bison. Elle est appelée lorsque le parser rencontre une erreur de syntaxe.
- `currentColumn` : Variable globale qui suit la colonne actuelle du lexer dans le fichier source. Elle est utilisée pour préciser la position exacte de l'erreur dans les messages d'erreur.

```
void yysuccess(char *s){  
    currentColumn+=yyleng;  
}  
  
void yyerror(const char *s) {  
    fprintf(stdout, "File output, line %d, character %d : %s \n", yylineno, currentColumn, s);  
    exit(1);  
}
```

6. Points Forts et Améliorations

Points Forts:

Le langage HTPL se distingue par sa syntaxe intuitive inspirée du HTML/XML, le rendant particulièrement accessible aux développeurs tout en maintenant une structure rigoureuse. Il offre un système de typage fort avec support pour les types de données fondamentaux (int, float, boolean, string) et les tableaux, assurant ainsi une robustesse lors de la compilation. Les structures de contrôle essentielles (if-else, while) sont présentes, accompagnées d'un système complet d'expressions arithmétiques et logiques. La déclaration explicite des variables et la gestion claire de leur portée contribuent à la lisibilité et à la maintenabilité du code, tandis que la vérification de type à la compilation permet de détecter les erreurs avant l'exécution.

Améliorations:

Le langage présente un potentiel d'évolution significatif avec plusieurs axes d'amélioration envisageables. L'introduction de fonctionnalités avancées comme les fonctions, les procédures et des structures de données plus complexes pourrait enrichir ses capacités. Le développement d'outils dédiés (IDE, débogueur) et l'optimisation du code généré renforceraient l'écosystème du langage. L'ajout de concepts de programmation orientée objet, de gestion de modules, et de bibliothèques standards étendrait considérablement ses cas d'utilisation. Une documentation exhaustive, incluant des guides de bonnes pratiques et des tutoriels, faciliterait son adoption. Ces améliorations permettraient à HTPL de devenir un langage plus mature et polyvalent, tout en conservant sa philosophie de simplicité et de clarté.

7. Conclusion

L'analyse sémantique constitue un pilier fondamental dans la conception et l'implémentation des compilateurs modernes. En assurant une vérification rigoureuse des types, une gestion efficace des données et une détection précoce des erreurs, elle joue un rôle déterminant dans la production de logiciels fiables et performants. Les différentes composantes abordées, telles que la table des symboles et la génération de code intermédiaire, illustrent la richesse et la complexité de ce domaine.

Cependant, ce rapport met également en lumière les limites actuelles, notamment l'absence de gestion des fonctions et de types de données avancés. Ces aspects soulignent les perspectives d'amélioration et d'évolution pour les futures itérations des compilateurs, dans un contexte où les besoins des développeurs et la diversité des langages ne cessent de croître.

Ainsi, cette exploration sémantique nous rappelle l'importance d'un design réfléchi et d'une implémentation minutieuse dans la chaîne de compilation. Les enseignements tirés de cette analyse ouvrent la voie à des avancées prometteuses, notamment dans l'optimisation des performances et l'extension des capacités fonctionnelles des outils de compilation. Ces progrès contribueront sans aucun doute à l'évolution de l'informatique et à la satisfaction des exigences toujours plus élevées des utilisateurs finaux.

Conclusion Générale

Le développement d'un compilateur pour le langage HTPL a permis de démontrer l'importance des étapes fondamentales de l'analyse lexicale, syntaxique et sémantique dans le processus de compilation. Chacune de ces phases apporte une contribution essentielle à la transformation d'un code source brut en un programme interprétable par une machine.

L'analyse lexicale, en décomposant le code en tokens distincts, a jeté les bases d'une analyse systématique et précise. Les choix effectués pour la reconnaissance des tokens et leur traitement ont permis d'assurer une correspondance claire avec les règles du langage, tout en minimisant les erreurs initiales.

L'analyse syntaxique a ensuite validé la structure du programme, garantissant que chaque construction respecte la grammaire du langage. L'utilisation de méthodes telles que la descente récursive et des outils comme Bison a non seulement permis une implémentation robuste mais a également facilité la gestion des priorités, des ambiguïtés et des erreurs.

Enfin, l'analyse sémantique a assuré la cohérence logique du programme. En vérifiant la déclaration et l'utilisation correcte des variables, la compatibilité des types et la validité des opérations, elle a renforcé la fiabilité du code généré. De plus, la génération de code intermédiaire a préparé le terrain pour des optimisations futures et une exécution efficace.

Cependant, ce projet a également mis en lumière certaines limitations, telles que le manque de prise en charge des fonctions et des types de données complexes, ainsi que des portées limitées. Ces aspects ouvrent des perspectives intéressantes pour de futures améliorations, incluant le support de fonctionnalités plus avancées et l'extension du langage HTPL.

En conclusion, ce travail a permis de mieux comprendre les interactions entre les différentes phases de compilation et leur importance dans la conception d'un compilateur performant. Les résultats obtenus constituent une base solide pour des travaux ultérieurs, que ce soit pour enrichir les capacités du langage ou pour appliquer ces principes à d'autres projets de compilation.