الجمهورية الجزائرية الديمقراطية الشعبية
République Algérienne Démocratique et Populaire
وزارة التعليم العالي و البحث العلمي
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE

المدرسة الوطنية العليا للإعلام الآلي
(المعهد الوطني للتكوين في الإعلام الآلي سابقا)
Ecole nationale Supérieure d'Informatique
ex. INI (Institut National de formation en Informatique)

Livrable 5 : Projet Compilation Analyse Sémantique

Membres du groupe

Dehili Hind
Dinari Yasmine
Rezzoug Aicha

Spécialité

Systèmes informatiques et logiciels Groupe 01

Année universitaire : 2024/2025

TABLE DE MATIERE

1. Notre langage

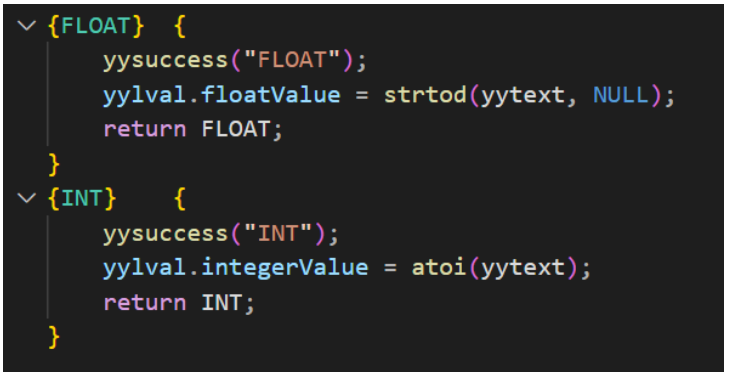
HumanScript est un langage de programmation qui s'inspire du Langage Khwarizm. À l'instar de ce langage éducatif, notre création adopte une syntaxe simple et intuitive, rappelant des structures proches du langage humain.L'objectif est de combiner l'efficacité d'un langage formel avec une approche naturelle, rendant la programmation plus accessible et compréhensible. Ce langage met l'accent sur la lisibilité et la facilité d'apprentissage, tout en s'appuyant sur des concepts informatiques solides. Exemple de code écrit en human script :

```
Let int x be 6
Let Dict nom be {"name": "yasmine", "prenom": "dinari"}
int r
r == 3
const float m be 3.14
const bool n be true
Let int y be 3
Let int z be y * x
Let float k be m + 1.3
```

2. Partie lexicale

L'analyse lexicale est la première étape du processus de compilation ou d'interprétation. Elle consiste à décomposer le code source en une série de tokens (ou lexèmes) qui peuvent être utilisés dans les étapes suivantes du processus, telles que l'analyse syntaxique. Cette étape permet de transformer une chaîne de caractères brute en une structure que le compilateur ou l'interpréteur peut comprendre et manipuler plus facilement.

```
NOTEQUALS "!="
ADDEQUALS "+="
SUBEQUALS "-="
MULEQUALS "*="
DIVEQUALS "/="
MODEQUALS "%="
NEG "!"
LESS "<"
LESSEQUALS "<="
GREATER ">"
GREATEREQUALS ">="
DOUBLEEQUALS "=="
AND "&&"
```



Nous avons apporté des changements à notre analyseur lexicale dans la partie **sémantique** , Les affectations de type ont été ajoutées pour permettre de **vérifier la compatibilité des types** tout au long de l'analyse sémantique du programme

3. Partie syntaxique

3.1 : Dans cette section, nous nous sommes concentrés sur la mise en œuvre de l'analyse syntaxique descendante en utilisant le langage C. L'objectif est de générer une table d'analyse syntaxique à partir d'un sous-ensemble de la grammaire de HumanScript. Les différentes étapes sont basées sur les transformations et les algorithmes étudiés en cours

Étapes de Réalisation

1. Choix du sous-ensemble de la grammaire TYPE ID.
2. Rendre la grammaire propre
3. Construction des Ensembles DEBUT et SUIVANT
4. Construction de la Table d'Analyse Syntaxique

5. **Vérification de la Table**

```
hind23@DESKTOP-Q5BNM2D:~$ gcc -o prog prog.c tableSymboles.c
hind23@DESKTOP-Q5BNM2D:~$ ./prog
Donner déclaration :  intt x
Erreur! intt
```

```
hind23@DESKTOP-Q5BNM2D:~$ gcc -o prog prog.c tableSymboles.c
hind23@DESKTOP-Q5BNM2D:~$ ./prog
Donner déclaration :  int c
La chaine est syntaxiquement correcte !

Table de symbole apres analyse:

+-----------+- ----------+-----------+-----------+
| ID    | Name        | Type      | Scope  |
+-----------+- ----------+-----------+-----------+
| 0     | c           | int       | 1      |
+-----------+- ----------+-----------+-----------+
```

3.2: Analyse syntaxique avec Bison

3.2.1 Déclaration des Tokens

```
/* Définition des tokens */
/* Types de données */
%token INT FLOAT BOOL STR
%token CONST ARRAY DICT FUNCTION
/* Mots-clés pour les déclarations et appels */
%token LET BE CALL WITH PARAMETERS
/* Structures de contrôle */
%token ELSEIF IF ELSE ENDIF
%token FOR EACH IN ENDFOR
%token WHILE ENDWHILE
%token REPEAT UNTIL ENDREPEAT
%token INPUT TO PRINT
%token SWITCH CASE DEFAULT ENDSWITCH
%token RETURN
/* Opérateurs */
%token ADD SUB MUL DIV INT_DIV MOD
%token EQUAL NOT_EQUAL GREATER_THAN LESS_THAN GREATER_EQUAL LESS_EQUAL
%token COLON LPAREN RPAREN LBRACE RBRACE COMMA LBRACKET RBRACKET
%token LOGICAL_AND LOGICAL_OR LOGICAL_NOT
/* Valeurs littérales et identificateurs */
%token TRUE FALSE
%token INT_LITERAL FLOAT_LITERAL STRING_LITERAL
%token ID COMMENT
```

3.2.2.Définition des Priorités et associativités des Opérateurs

```
/* Définition des priorités et associativités des opérateurs
 * L'ordre des déclarations définit la priorité (du plus bas au plus haut)*/
%left LOGICAL_OR
%left LOGICAL_AND
%left EQUAL NOT_EQUAL
%left LESS_THAN GREATER_THAN LESS_EQUAL GREATER_EQUAL
%left ADD SUB
%left MUL DIV MOD INT_DIV
%right LOGICAL_NOT
%nonassoc UMINUS  /* Pour la négation unaire */
```

3.2.3. Règles de production

```
/* Différents types de boucles */
LoopStatement:
    ForLoop
    | WhileLoop
    | RepeatLoop
    ;
/* Boucle For avec itération sur un tableau */
ForLoop:
    FOR EACH ID IN Expression COLON StatementList ENDFOR
    ;
/* Boucle While classique */
WhileLoop:
    WHILE Expression COLON StatementList ENDWHILE
    ;
/* Boucle Repeat Until */
RepeatLoop:
    REPEAT COLON StatementList UNTIL Expression ENDREPEAT
    ;
```

Remarque : Nous avons apporté des modifications à notre fichier bison (Injection des routines ,insertions des quadruplets , stockage dans la table de symbole , etc …) au niveau de l'analyse sémantique

4. Analyse sémantique

L'analyse sémantique a pour objectif de comprendre le sens d'un message en se basant sur la signification des mots, contrairement à l'analyse lexicale et syntaxique qui reposent sur un lexique ou une grammaire. En compilation, l'analyse sémantique intervient après l'analyse syntaxique afin de vérifier la conformité des opérandes aux spécifications du langage source et de générer une représentation intermédiaire du programme. Nous avons suivi les étapes suivantes :

1. Transformation de la grammaire syntaxique en grammaire sémantique.
2. Génération de code intermédiaire sous forme de quadruplets.
3. Ajout de routines sémantiques.

On tient à préciser que nous avons réussi à implémenter les instructions suivantes : opérations arithmétiques/logiques , Déclarations et affectation (Types simples et structurés) , Boucles (Repeat , While)

4.1 La Table des Symboles dans HumanScript

Dans le code fourni, la table des symboles est implémentée sous forme de table de hachage (hash table), où chaque entrée est stockée dans un "bucket" en fonction de la valeur de hachage du nom du symbole.

La table des symboles est définie par la structure suivante :

```
typedef struct SymbolTable {
    SymbolEntry *buckets[HASH_TABLE_SIZE];
    int nextId;
} SymbolTable;
```

Chaque entrée de la table des symboles (SymbolEntry) contient les informations suivantes :

- **ID** : Un identifiant unique associé au symbole.
- **Nom** : Le nom de la variable ou de la constante.
- **Type** : Le type de la variable (par exemple, entier, flottant, booléen, chaîne de caractères, etc.).
- **Valeur** : La valeur actuelle de la variable.
- **isConst** : Un booléen indiquant si le symbole est une constante.

- **isInitialized** : Un booléen indiquant si la variable a été initialisée.
- **scopeLevel** : Le niveau de portée du symbole, permettant de gérer les blocs imbriqués.
- **next** : Un pointeur vers l'entrée suivante dans le même bucket, utilisé pour gérer les collisions.

Les déclarations se trouvent au niveau du fichier 'tableSymboles.h' et le corps des fonctions dans 'tableSymboles.c'.

4.1.1 Fonctions Clés de la Table des Symboles

```c
//Initialise et retourne une nouvelle table des symboles vide.
SymbolTable *createSymbolTable();
//Ajoute un nouveau symbole à la table des symboles.
void insertSymbol(SymbolTable *table, const char *name, const char *type,
const char *value, int scopeLevel, bool isConst, bool isInitialized);
//Recherche un symbole par son nom et son niveau de portée.
SymbolEntry *lookupSymbolByName(SymbolTable *table, const char *name, int scopeLevel);
//Recherche un symbole par son identifiant et son niveau de portée.
SymbolEntry *lookupSymbolById(SymbolTable *table, int id, int scopeLevel);
//Supprime un symbole de la table par son nom.
void deleteSymbolById(SymbolTable *table, int id);
//Supprime un symbole de la table par son identifiant.
void deleteSymbolByName(SymbolTable *table, const char *name);
//Libère la mémoire allouée pour la table des symboles et ses entrées.
void freeSymbolTable(SymbolTable *table);
//Supprime tous les symboles de la table sans libérer la table elle-même.
void clearSymbolTable(SymbolTable *table);
//Vérifie si un symbole existe dans la table par son nom.
int symbolExistsByName(SymbolTable *table, const char *name, int scopeLevel);
//Vérifie si un symbole existe dans la table par son identifiant.
int symbolExistsById(SymbolTable *table, int id, int scopeLevel);
//Affiche le contenu complet de la table des symboles.
void listAllSymbols(SymbolTable *table);
//Met à jour la valeur d'un symbole existant.
void updateSymbolValue(SymbolTable *table, int id,const char *newValue, int scopeLevel);
//Libère la mémoire allouée pour une entrée de la table des symboles.
void freeSymbolEntry(SymbolEntry *entry);
```

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

4.2.Explication de la Partie semantic.c

Le fichier semantic.c joue un rôle crucial dans l'analyse sémantique en fournissant des fonctions pour manipuler les tableaux, les listes d'expressions, et convertir les types et les valeurs. Ces fonctions permettent de construire des structures de

données complexes et de les stocker de manière efficace dans la table des symboles.

```
// Crée un tableau dynamique.
ArrayType* createArray();
// Crée un tableau à partir d'une liste d'expressions.
ArrayType* createArrayFromExprList(ExpressionList* list);
// Convertit un type numérique en une chaîne de caractères représentant le type.
void getTypeString(int type, char *typeStr);
// Convertit une valeur d'un type donné en une chaîne de caractères pour le stockage dans la table des symboles.
void createValueString(int type, const char *inputValue, char *valueStr);
// Crée un nœud pour une liste d'expressions à partir d'une expression donnée.
ExpressionList* createExpressionNode(expression expr);
// Ajoute une expression à la fin d'une liste d'expressions.
ExpressionList* addExpressionToList(ExpressionList* list, expression expr);
```

4.3.Traduction des structures de contrôle -repeat until-

Pour gérer et accéder aux différentes données nécessaires dans les routines sémantiques, il est indispensable de typer certains non-terminaux. Par défaut, un non-terminal est associé à %union. Nous définirons donc des structures spécifiques (struct) pour chaque non-terminal, en fonction des besoins.

Boucle REPEAT-UNTIL:
La boucle REPEAT-UNTIL est une structure de contrôle qui exécute un bloc de code au moins une fois, puis vérifie une condition à la fin pour décider si elle doit répéter l'exécution. Voici comment elle est traduite en quadruplets :
Routine RepeatStart
Objectif : Initialiser la boucle et générer une étiquette de début.
Actions :Générer un identifiant unique pour la boucle (repeatId).
Créer une étiquette de début (repeatStartLabel) pour marquer le début de la boucle.
Insérer un quadruplet pour l'étiquette de début.
Empiler l'identifiant de la boucle sur la pile pour une utilisation ultérieure.
Routine RepeatEnd
Objectif : Gérer la fin de la boucle et vérifier la condition.
Actions :Dépiler l'identifiant de la boucle pour récupérer les informations nécessaires.
Créer une étiquette de fin (repeatEndLabel) pour marquer la fin de la boucle.
Vérifier que la condition de la boucle est de type booléen.
Insérer un quadruplet pour le saut conditionnel (BZ) : si la condition est fausse, la boucle se termine.
Insérer un quadruplet pour l'étiquette de fin.

```
RepeatLoop: RepeatStart StatementList RepeatEnd;
RepeatStart: REPEAT COLON {
    int repeatId = qc;
    char repeatStartLabel[20];
    sprintf(repeatStartLabel, "REPEAT_START_%d", repeatId);
    insererQuadreplet(&q, repeatStartLabel, "", "", "", qc++);
    empiler(stack, repeatId);
}
RepeatEnd : UNTIL Expression ENDREPEAT
{
int repeatId = depiler(stack);
    char repeatStartLabel[20];
    char repeatEndLabel[20];
    sprintf(repeatStartLabel, "REPEAT_START_%d", repeatId);
    sprintf(repeatEndLabel, "REPEAT_END_%d", repeatId);
    char typeStr[MAX_TYPE_LENGTH];
        getTypeString($2.type, typeStr);

    if (!strcmp(typeStr,"boolean")) {
        yyerror("Repeat-until condition must be a boolean expression");
        YYERROR;
    }
    insererQuadreplet(&q, "BZ", repeatStartLabel, "",  $2.value, qc++);
    insererQuadreplet(&q, repeatEndLabel, "", "", "", qc++);
}
;
```

4.4. Jeu d’essai

```
int x
Let int num be 10
Let float y be 3.14
Let int o be 5
Let int h be num + o
Let bool isActive be true
Let str name be "Aicha" + "Rezzoug"
Let Array numbers be [1, 2, 3, 4, 5]
Let Dict userInfo be {"name": "Aicha", "age": 21}
Let int counter be 10
While counter < 5:
    Print "Counter is: "
    Let counter be counter + 1
EndWhile
Repeat:
Let counter be counter - 1
Until counter == 0
EndRepeat
```

```
Quad[1]=[ := , 0 ,  , x ]
Quad[2]=[ := , 10 ,  , num ]
Quad[3]=[ := , 3.14 ,  , y ]
Quad[4]=[ := , 5 ,  , o ]
Quad[5]=[ + , 10 , 5 , t5 ]
Quad[6]=[ := , 15 ,  , h ]
Quad[7]=[ := , true ,  , isActive ]
Quad[8]=[ CONCAT , Aicha , Rezzoug , t8 ]
Quad[9]=[ := , AichaRezzoug ,  , name ]
Quad[10]=[ ARRAY_DECL , numbers , [1,2,3,4,5] , t10 ]
Quad[11]=[ := , 10 ,  , counter ]
Quad[12]=[ < , 10 , 5 , t12 ]
Quad[13]=[ WHILE_COND_13 ,  ,  ,  ]
Quad[14]=[ BZ , WHILE_END_13 ,  , false ]
Quad[15]=[ + , 10 , 1 , t15 ]
Quad[16]=[ := , 11 ,  , counter ]
Quad[17]=[ BR ,  ,  , WHILE_COND_13 ]
Quad[18]=[ WHILE_END_13 ,  ,  ,  ]
Quad[19]=[ REPEAT_START_19 ,  ,  ,  ]
Quad[20]=[ - , 11 , 1 , t20 ]
Quad[21]=[ := , 10 ,  , counter ]
Quad[22]=[ == , 10 , 0 , t22 ]
Quad[23]=[ BZ , REPEAT_START_19 ,  , false ]
Quad[24]=[ REPEAT_END_19 ,  ,  ,  ]
```

```
Symbol Table Contents:
ID      Name    Type    Scope   Value
----------------------------------------
4       h       int     0       15
6       name    string  0       AichaRezzoug
3       o       int     0       5
0       x       int     0       (uninitialized)
2       y       float   0       3.14
7       numbers array   0       [1,2,3,4,5]5    isActive        bool    0       true
1       num     int     0       10
8       counter int     0       10
```

4.5.Conclusion

Nous avons réussi à construire un mini-compilateur capable de générer un code intermédiaire sous forme de quadruplets ! cette expérience a été enrichissante, nous permettant d'explorer l'univers de la compilation et de comprendre les principes de base des compilateurs.