UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE HOUARI BOUMEDIENE

FACULTE D'INFORMATIQUE

3ème ING

# Sécurité Informatique


## Projet de Compilation

CHEMLI Ayoub 222231387702

BELKADI Adam 222231581302

DENDANI Safwane Souhil 222231626913

CHELALI Mohamed Said Nail 222231355706

GUENANE Abdelkarim 222231568118

BOUCHAMA Imadeddine 222231498708

# Introduction :

Ce projet vise à concevoir un mini-compilateur pour le langage **MinING**, un langage simplifié conçu pour illustrer les concepts clés de la compilation. Il inclut les principales étapes :

- **Analyse lexicale** pour identifier les éléments du programme.
- **Analyse syntaxique** et **sémantique** pour valider la structure et la cohérence.
- **Génération de code intermédiaire** pour préparer la traduction en code machine.

En parallèle, une table des symboles sera gérée pour suivre les variables et constantes, et les erreurs seront détectées à chaque étape. Ce projet offre une compréhension pratique des processus fondamentaux de compilation.

# Description du langage MinING :

## 1) Structure générale :

MinING suit une structure organisée en trois blocs principaux :

- **VAR_GLOBAL** : contient les variables globales.
- **DECLARATION** : regroupe les variables locales et constantes.
- **INSTRUCTION** : définit les opérations exécutables

## 2) Commentaires :

Les commentaires, ignorés par le compilateur, débutent par %% et sont limités à une seule ligne.

## 3) Types de données :

Le langage prend en charge trois types de données principaux :

- **INTEGER** : pour les entiers signés ou non.
- **FLOAT** : pour les nombres à virgule flottante.
- **CHAR** : pour un seul caractère ou une chaîne (tableau de CHAR).

## 4) Déclaration :

Les variables et constantes sont déclarées avec leur type.

- **Variables simples** : TYPE nomVariable ;
- **Tableaux** : TYPE nomTable[taille] ;
- **Constantes** : CONST TYPE idf = valeur ;

### 5) Opérateurs :

MinING inclut des opérateurs :

- **Arithmétiques** : +, -, *, /
- **Logiques** : &&, ||, !
- **Comparaison** : <, >, <=, >=, ==, !=

### 6) Instructions :

- **Affectation** : Idf = expression ;
- **Condition** (IF) : IF (condition) { … } ELSE { … }
- **Boucles** (FOR) : FOR (init : pas : cond) { … }
- **Entrées/Sorties** : READ(nomVar), WRITE(…).

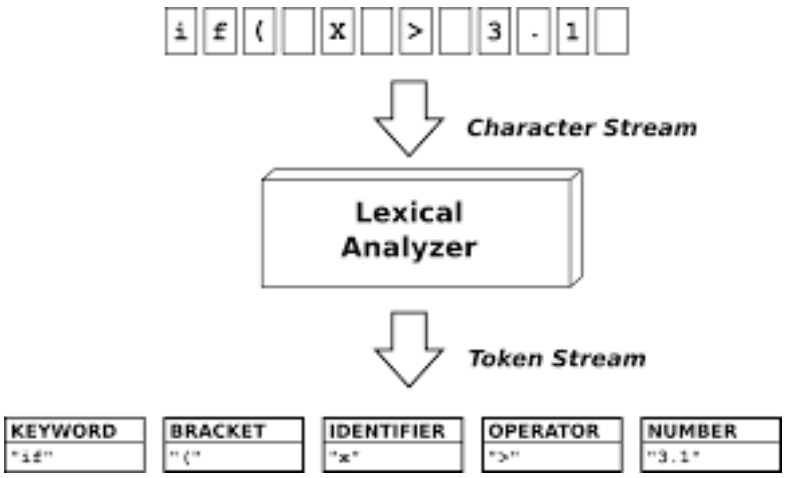
# Conception de Compilateur :

### 1) Analyse Lexical :

L'analyse lexicale est la première étape du compilateur, où le texte source est converti en une séquence de tokens (éléments syntaxiques comme les mots-clés, les opérateurs, etc.).

Nous avons réalisé un analyseur lexical à l'aide de l'outil libre **Flex** qui permet de générer des programmes qui reconnaissent des motifs lexicaux dans du texte.

Flex défini l'entité lexicale à reconnaitre à l'aide d'expressions régulières.



Le code est devisee en parties :

**Initialisation et Préparation de l'Analyseur Lexical**
Bloc %{ ... %} pour inclure les bibliothèques et initialiser les variables globales.

**Déclaration des Options de Flex**
%option noyywrap pour désactiver la fonction par défaut yywrap().

**Déclaration des Expressions Régulières**
Définitions des expressions régulières pour identifier les structures lexicales : identificateurs, nombres, chaînes, etc.

**Reconnaissance des Mots-clés du Langage**
Associations des mots-clés aux tokens avec mise à jour de la position.

**Gestion des Identificateurs et Constantes**
Reconnaissance des identificateurs, chaînes de caractères et validation des constantes numériques avec vérification des limites.

**Traitement des Opérateurs et Symboles Spéciaux**
Reconnaissance des opérateurs logiques, arithmétiques, et des symboles de ponctuation.

**Gestion des Espaces, Retours à la Ligne et Commentaires**
Mise à jour de la position pour les espaces, les tabulations, les retours à la ligne, et les commentaires.

**Détection des Comparateurs et Opérateurs Relationnels**
Reconnaissance des opérateurs comme >, <, >=, <=, ==, !=.

**Gestion des Caractères Non Reconnus**
Affichage d'une erreur pour les entités lexicales non reconnues.

```
">"             {  colonne=colonne+ yyleng; return sup; }
"<"             {  colonne=colonne+ yyleng; return inf; }
">="            {  colonne=colonne+ yyleng; return supeg; }
"<="            {  colonne=colonne+ yyleng; return infeg; }
"=="            {  colonne=colonne+ yyleng; return eg; }
"!="            {  colonne=colonne+ yyleng; return noneg; }
.               {printf ("Erreur lexical, a ligne %d colonne %d: entitee %s non reconnu" ,nb_ligne,colonne,yytext);}

%%
```

## 2) Analyse Syntaxique + qlq Semantique (SyntaxicoSemantique):

Cette étape suit l'analyse lexicale et utilise les entités produites par cette dernière.

Pour notre projet nous avons réalisé un analyseur syntaxique à l'aide de l'outil Bison qui permet de générer des analyseurs avec des grammaires LALR (1).

L'analyseur:

- Définit la syntaxe d'un langage de programmation spécifique (avec variables, constantes, types, instructions, etc.).
- Valide la structure syntaxique et détecte les erreurs sémantiques comme :

- Déclarations multiples.
- Modifications interdites de constantes.
- Incompatibilités de types.

- Fournit des messages d'erreur clairs pour aider à corriger le code source.

```
VAR_GLOBAL {
    %%ce bloc est le meme que declaration%%
}
DECLARATION {
FLOAT B ;
INTEGER D ;
INTEGER E ;
INTEGER Tab[1];
CONST FLOAT Alpha= (-3.5);
CONST INTEGER Beta = 315;
CONST INTEGER D12 = (-5.4) ;
CONST CONST CHAR P[3] = "123";
FLOAT X,Y;
}
```

exemple.txt

```
Erreur Syntaxique a la ligne 12 et colonne 11:
PS C:\Users\21356\OneDrive\Bureau\Mini-compiler-main\Mini-compiler-main>
```

Erreur syntaxique

Cette partie de l'analyse produit une table de symbole :

```
VAR_GLOBAL {
    %%ce bloc est le meme que declaration%%
}
DECLARATION {
FLOAT B ;
INTEGER D ;
INTEGER E ;
INTEGER Tab[1];
CONST FLOAT Alpha= (-3.5);
CONST INTEGER Beta = 315;
CONST INTEGER D12 = (-5.4) ;
CONST CHAR P[3] = "123";
FLOAT X,Y;
}
```

PROBLÈMES SORTIE CONSOLE DE DÉBOGAGE TERMINAL PORTS

La table des symboles:

| Type | Nature | Nom de la Variable |
|---|---|---|
| Float | Variable | Y |
| Float | Variable | X |
| Char | Constant | P |
| Integer | Constant | Beta |
| Float | Constant | Alpha |
| Integer | Variable | Tab |
| Integer | Variable | E |
| Integer | Variable | D |
| Float | Variable | B |

exemple.txt Table de symboles

## 3) Gestion de la table des Symboles :

La table des symboles est une structure fondamentale dans tout compilateur, permettant de gérer les informations sur les variables, constantes et autres entités déclarées dans un programme.

Voici les principales fonctions associées :

**Structure de la Table des Symboles :**

```c
typedef struct element *TS;
typedef struct element {
    char nom[20];
    int type;   //! 0=intgr, 1=float, 2=char
    int nature; //! 0=var, 1=cst
    char value[20];
    TS suivant;
} element;
```

**Recherche d'une entité dans la table :**

```c
// Initialisation de la table des symboles
TS TableSymbole = NULL;  // La tête de la liste

//* Recherche d'une variable dans la table des symboles
TS Recherche(char n[]) {
    TS p = TableSymbole;
    while (p != NULL && strcmp(p->nom, n) != 0) {
        p = p->suivant;
    }
    return p;
}
```

**Insertion dans la Table des Symboles :**

```
int insertion(char name[20], int type, int nature) {
    TS nv;
    if (Recherche(name) == NULL) {
        nv = (TS)malloc(sizeof(element));
        nv->type = type;
        nv->nature = nature;
        strcpy(nv->nom, name);
        nv->suivant = TableSymbole;
        TableSymbole = nv;

        printf("%s %d %d\n", nv->nom, nv->type, nv->nature);
        return 1;
    } else {
        printf("Insertion non effectuée\n");
        return 0;
    }
}
```

**Fonctions secondaires :**

- **insererVal()** : Permet d'ajouter une valeur à une entité existante dans la table.
- **getValue()** : Récupère la valeur associée à une entité donnée.
- **getType()** : Retourne le type d'une entité.
- **insererparam()** : Gère l'insertion des paramètres dans une liste intermédiaire.

## 4) Analyse Sémantique :

L'analyse sémantique constitue une étape cruciale dans le processus de compilation. Elle vise à garantir que les structures syntaxiquement valides respectent les règles sémantiques du langage MinING. Cela inclut la vérification des types, des déclarations, et des instructions.

**Vérification des déclarations :**

La fonction suivante vérifie si une entité utilisée est déjà déclarée :

```
int non_declarer(char name[20]){

    TS x= Recherche(name);
    if (x!=NULL){return 1;} //faux : valeur declarer
    else{ return 0;} // vrai : valeur non declarer
}
```

**Vérification des types :**

Cette vérification assure que les entités manipulées dans une expression sont compatibles :

```
int incompatible_type(int type1, int type2){
    if (type1 != type2) { return 0;} // faux : type icompatible
    return 1; // vrai : le mm type
}
```

**Gestion des constantes :**

Une tentative de modification d'une constante est interceptée avec la fonction suivante:

```
int modification_cst(char name[20]){
    TS x= Recherche(name);
    if (x->nature== 1) {

        return 0;// vrai : on modifie une constante
        }
    return 1 ; // faux: c'est une variable
}
```

L’analyseur signalera toute erreur sémantique :

```
VAR_GLOBAL {
    %%ce bloc est le meme que declaration%%
}
DECLARATION {
FLOAT B ;
INTEGER D ;
INTEGER E ;
INTEGER Tab[1];
CONST FLOAT Alpha= (-3.5);
CONST INTEGER Beta = 315;
CONST INTEGER D12 = (-5.4) ;
CONST CONST CHAR P[3] = "123";
FLOAT X,Y;
}
```

exemple.txt

```
Erreur Semantique: Incompatiblite du type de constante D12 dans la ligne 11 et la colonne 23
```

## 5) Génération du Code Intermédiaire :

Cette partie du projet de compilation concerne la génération du code intermédiaire sous forme de quadruplets. Il s'agit d'une étape intermédiaire importante qui transforme le code source en une représentation à trois adresses, facilitant ainsi les optimisations ultérieures et la génération du code final.

Le code fourni comprend :

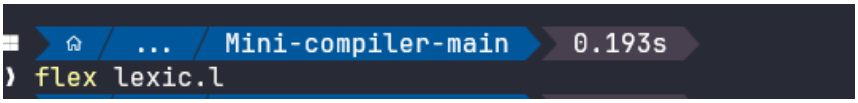
- La structure des quadruplets et leur gestion (quadruplet.h)
- L'intégration de la génération des quadruplets dans l'analyseur syntaxique
- Les règles de production pour générer les quadruplets pour différentes instructions

Cette partie s'insère dans un projet de compilation plus large qui doit inclure d'autres composants comme :

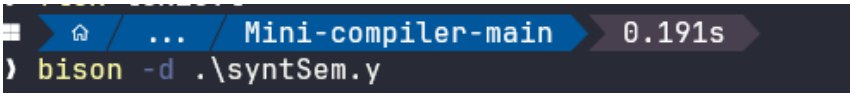
- L'analyse lexicale
- L'analyse sémantique
- La table des symboles
- La génération du code final

# Methode utilisation:

1) étape 1 :




2) étape 2 :




3) étape 3:


4) étape 4: