N° d'ordre :
N° de série :

République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Superieur et de la Recherche Scientifique

---

UNIVERSITÉ ECHAHID HAMMA LAKHDAR
EL OUED
FACULTÉ DES SCIENCES ET DE TECHNOLOGIE

Mémoire de fin d'étude

LICENCE ACADEMIQUE

Domaine: Mathématiques et Informatique
Filière: Informatique
Spécialité: Informatique fondamentale

Thème

Conception et développement d'un Compilateur (Sur la base de la syntaxe algorithmique)

Présenté par:
Ab Del Ouohab Ahtirib
Aiman Haddad
Mohammed Saleh Hamdi

Sous la supervision de :

Mr. Khelaifa Abdennacer MAA

Année universitaire 2014 – 2015

Remerciements

Apres tous, nous tenons à remercier le Dieu Tout-Puissant qui nous a bénis, qui nous a aidés à terminer ce travail.

En présentant ce travail, nous tenons à remercier Mr. Khelaifa Abdennacer *, l'encadreur pour sa serviabilité, sa disponibilité et ses remarques constructives qui nous ont utiles tout au long de notre projet.*

Nous remercions tous ceux qui nous ont aidés à atteindre notre objectif, en particulier Mr .Bachir SAAID, Docteur en Informatique à l'université de Ouargla, Mr.Yacine KHALDI et Mr.Abdelkader Laouid pour leur soutien continu.

Nous tenons à remercier les messieurs membres de jury pour l'honneur qu'ils nous ont fait en acceptant de juger notre travail.

Je dédie :

- Notre source du savoir : prophète Mohammed
- A mon père qui n'a jamais hésiter de me soutenir
- A la prunelle de mes yeux Ma mère
- A toute la famille, mes frères et mes sœurs
- A mes amis et mes collègues de promotion
- A tout ce qui l'ont participé de prés ou loin tout au long de mon cursus scolaire
- Tous ceux qui me connaissent.

Abd El-oahab AHTIRIB

Je dédie ce travail à :

- Ma très chère mère.
- Mon cher père.
- Mes frères.
- Mes sœurs.
- Tous mes amis (es).
- Et tous ceux qui me connaissent.

Aiman Haddad

Je dédie ce travail à :

- Ma très chère mère.
- Mon cher père.
- Mes frères.
- Mes sœurs.
- Tous mes amis (es).
- Et tous ceux qui me connaissent.

Mohammed Saleh Hamdi

RESUME

En générale, l'objectif de ce projet vise à la programmation d'un compilateur qu'utilise la syntaxe algorithmique- afin de transformes un code source (Algorithme) vers un langage d'assemblage et de l'exécute, et aussi de générer un fichier exécutable (.exe) sur n'importe quelle machine.

Pour atteindre cet objectif, Nous allons utiliser un générateur d'analyseur lexical Quex, aussi nous allons utiliser un générateur d'analyseur syntaxique Bison, après cela, on écrit le code source de l'analyseur Sémantique, de la table de symbole et le la générateur de code nous-mêmes (manuellement) en utilisant le langage de programmation C++.

Après la conception et la programmation du compilateur, Nous aurons besoin de développer un IDE (Environnement de développement intégré), Et va être une interface utilisateur comprenant une zone de texte pour écrire le code source (Algorithme) au sein, et les menus et les boutons supplémentaires pour compiler et exécuter les algorithmes, Nous allons utiliser Microsoft Visual C # .NET pour développer cette IDE.

MOTS-CLES :
Quex, Bison, Microsoft Visuel C + + .NET, C++.

Summary

In général, the aime of This Project is to développe a compiler –winch is uses algorithmic syntax- to transform a source code (algorithm) to an assembly language and execute it, also generates an executable (.exe) file on any machine.

To achieve this goal, we will use a lexical generator called Quex, so we'll use a parser generator tool called Bison, after that, we write the source code of the Semantic Analyzer, the symbol table and the code generator ourselves (manually) using the C ++programming language.

After designing and programming the compiler, we will need to develop an IDE (Integrated Development Environment), and will be a user interface comprising a text area to write the source code (algorithm) within, and additional menus and buttons to compile and run algorithms, we will use Microsoft Visual C # .NET to develop this IDE.

Keywords :
Quex, Bison, Microsoft Visuel C + + .NET, C++.

SOMMAIRE

TABLE DES FIGURES

INTRODUCTION GENERALE

La conception du compilateur est un sujet que beaucoup croient être fondamental et indispensable de la science informatique. Cet est un sujet qui a été étudié intensivement depuis le début des années 1950 et continue d'être un domaine de recherche important aujourd'hui. (Bergmann, 2010)

La conception du compilateur est une partie importante du programme d'études de premier cycle pour de nombreuses raisons :

1. Il fournit aux étudiants une meilleure compréhension et l'appréciation de langages de programmation.
2. Les techniques utilisées dans les compilateurs peuvent être utilisés dans d'autres applications avec des langages de commande.
3. Il fournit la motivation pour l'étude de sujets théoriques.
4. Cet union véhicule pour un projeté programmation étendu.

Notre projet vise à la programmation d'un compilateur -qu'utilise la syntaxe algorithmique- afin de transformer un code source (Algorithme) vers un langage d'assemblage et de l'exécute, et aussi de générer un fichier exécutable (.exe) sur n'importe quelle machine de cet algorithme.

Pour atteindre cet objectif, Nous allons utiliser un générateur d'analyseur lexical (FLEX), aussi nous allons utiliser un générateur d'analyseur syntaxique (Bison), après cela, on écrit le code source de l'analyseur Sémantique, de la table de symbole et le la générateur de code nous-mêmes (manuellement) en utilisant le langage de programmation C++.

Après la conception et la programmation du compilateur, Nous aurons besoin de développer un IDE (Environnement de développement), Et va être une interface utilisateur comprenant une zone de texte pour écrire le code source (Algorithme) au sein, et les menus et les boutons supplémentaires pour compiler et exécuter les algorithmes, Nous allons utiliser Java ou C # .NET pour développer cette IDE, nous décidons plus tard.

CHAPITRE 1 : QUELQUES NOTIONS DE COMPILATION

1.1 QU'EST-CE QU'UN COMPILATEUR ?

Les types d'instructions que le processeur de l'ordinateur est capable d'exécuter. En général, ce sont des opérations très simples et primitives. Par exemple, il y a souvent des instructions qui font ces types d'opérations :

(1) ajouter deux numéros enregistrés dans la mémoire,

(2) numéros de déplacer d'un endroit à un autre dans la mémoire,

(3) se déplacent d'informations entre le processeur et la mémoire.

Mais il ne est certainement pas seule instruction capable de calculer une expression arbitraire tels que $(1 + x)^3 + \left(\sqrt{x} + y\right)^2$ Et il n'y a aucun moyen de faire ce qui suit avec une seule instruction :

if (array6[loc] < MAX) sum = 0; else array6[loc] = 0;

Ces capacités sont mises en œuvre avec un logiciel de traduction, connu comme **un compilateur**. La fonction du compilateur est d'accepter des déclarations telles que celles ci-dessus et de les traduire en séquences d'opérations en langage machine qui, si chargé en mémoire et exécuté, porterait sur le calcul prévu.

Un compilateur est un programme informatique qui transforme un code source écrit dans un langage de programmation (le langage source) en un autre langage informatique (le langage cible) pour qu'il puisse être exploité par la machine.

Le compilateur traduit le code source, écrit dans un langage de haut niveau d'abstraction (comme C++, Pascal…etc.), facilement compréhensible par l'humain, vers un langage de plus bas niveau, un langage d'assemblage ou langage machine.

Si une partie de l'entrée au compilateur C ressemblait à ceci :

A = B + C * D ;

La sortie correspondant à cette entrée pourrait ressembler à ceci :

```
     LOD R1, C            // Charger la valeur de C dans reg 1
MUL R1, D            // Multiplier la valeur de D par reg 1
STO R1, TEMP1        //   stocker   le   résultat   dans   TEMP1
LOD R1, B            // Charger la valeur de B dans reg 1
ADD R1, TEMP1        // Ajouter valeur de Temp1 d'enregistrer une
STO R1, TEMP2        //   stocker   le   résultat   dans   TEMP2
```

```
MOV A, TEMP2          // Déplacement TEMP2 à A, le résultat final
```

(Alfred V. Aho, Monica S. Lam, Ravi Sethi, & Jeffrey D. Ullman, 2007).

Le compilateur doit être assez intelligent pour savoir que la multiplication devrait être faite avant l'addition même si l'addition est lue en premier lors de la numérisation de l'entrée.

Le compilateur doit également être suffisamment intelligent pour savoir si l'entrée est un programme correctement formée (ce qu'on appelle la vérification de syntaxe), et d'émettre des messages d'erreur utiles se il y a des erreurs de syntaxe. (Bergmann, 2010)

1.2 LES PHASES DE LA COMPILATION

Afin de simplifier le processus de conception et de construction compilateur, le compilateur est mis en œuvre en plusieurs phases. En général, un compilateur consiste en au moins trois phases :

1. l'analyse lexicale,
2. l'analyse de la syntaxe,
3. la génération de code.

En outre, il pourrait y avoir d'autres phases d'optimisation employées à produire des programmes efficaces objet. (Bergmann, 2010)

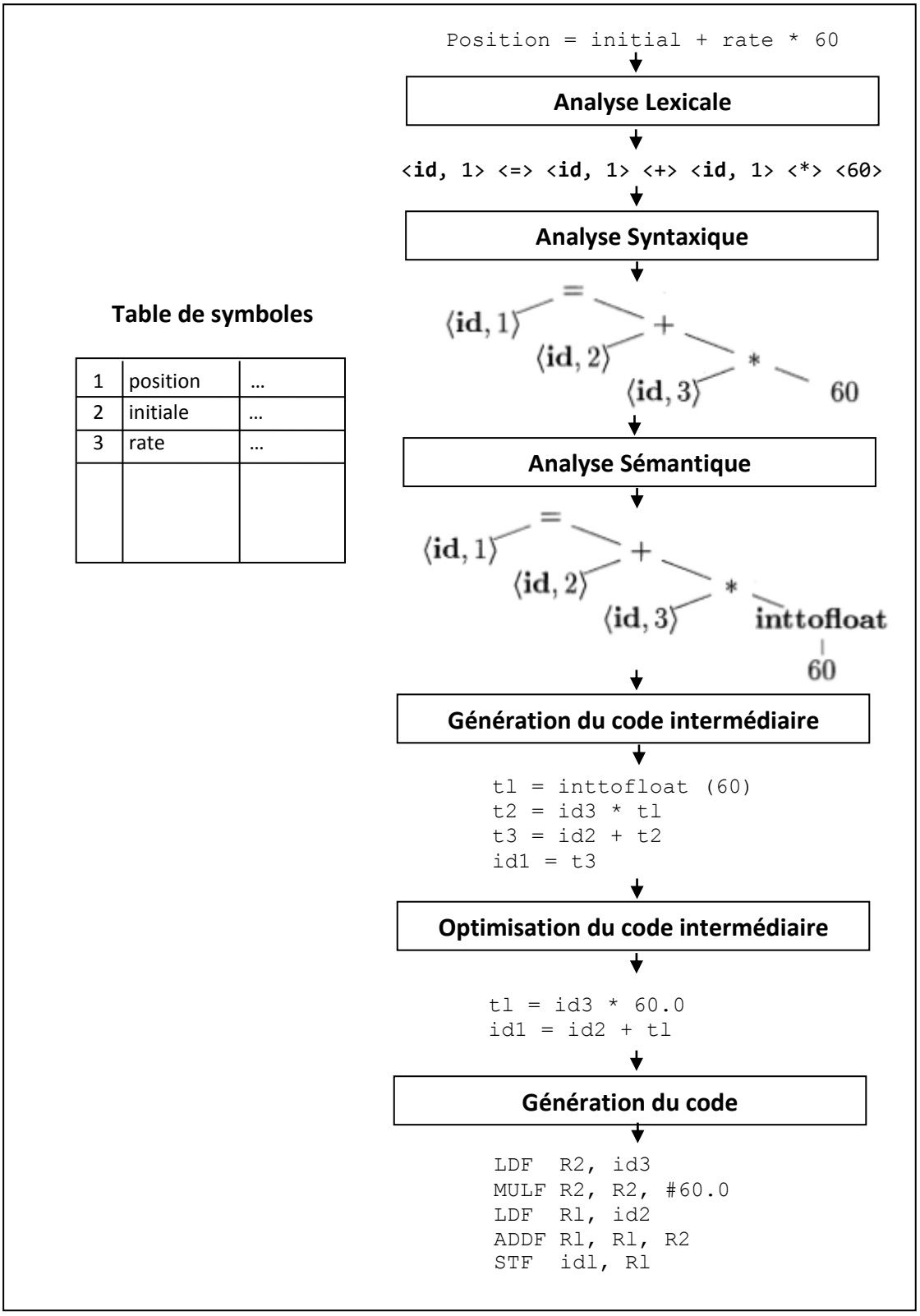

FIGURE 1: LES DIFFERENTES PHASES DE COMPILATION

1.2.1 ANALYSE LEXICALE

La première phase d'un compilateur est appelé analyse lexicale (et est également connu comme un **Lexical Scanner)**. Comme le laisse entendre son nom, analyse lexicale tente d'isoler les "mots" dans une chaîne d'entrée. Nous utilisons le mot "mot" dans un sens technique. Un mot, aussi connu comme un lexème, un élément lexical, ou d'une unité lexicale, est une chaîne de caractères d'entrée qui est pris comme une unité et transmises à la prochaine phase de compilation. Exemples de mots sont les suivants :

1) **mots-clés** : tant-que, vide, si, pour, ...
2) **identificateurs** déclarés par le programmeur
3) **opérateurs** : +, -, *, /, =, ==, ...
4) **des constantes numériques** : numéros tels que 124, 12,35, 0.09E-23, etc.
5) **les constantes de caractères** : caractères simples ou des chaînes de caractères entre guillemets.
6) **caractères spéciaux** : caractères utilisés comme délimiteurs tels que ( ) ; :
7) **Commentaires** : ignoré par les phases ultérieures. Ceux-ci doivent être identifiés par l'analyseur, mais ne sont pas inclus dans la sortie.

La sortie de la phase lexicale est un flux de jetons (TOKENS) correspondant aux mots décrits ci-dessus. En outre, cette phase s'appuie tableaux qui sont utilisés par les phases ultérieures du compilateur. Un tel tableau, appelé la table des symboles, des magasins tous les identifiants utilisés dans le programme source, y compris les informations et les attributs des identificateurs pertinents. Dans les langages de blocs-structuré, il peut être préférable de construire la table des symboles lors de la phase d'analyse syntaxique car les blocs de programmes (et les champs d'identification) peuvent être imbriquées. (Bergmann, 2010)

1.2.2 ANALYSE SYNTAXIQUE

Ceci est également connu comme **« Parsing »**. Il est à peu près l'équivalent de vérifier que du texte ordinaire écrite dans un langage naturel (par exemple l'anglais) est grammaticalement correct (sans se soucier de sens). (Bergmann, 2010)

Le but de l'analyse de syntaxe ou l'analyse est de vérifier que nous avons une séquence valide de jetons. Les jetons sont séquence valide de symboles, de mots clés, identificateurs etc. Notez que cette séquence n'est pas nécessairement significative ; aussi loin que la syntaxe va, une phrase telle que **«true + 3 »** est valable mais il ne fait pas de sens dans la plupart des langages de programmation.

L'analyseur prend les jetons (**TOKENS**) fabriqués au cours de la phase d'analyse lexicale, et tente de construire une sorte de structure en mémoire pour représenter cette entrée. Souvent, cette structure est un «arbre de syntaxe abstraite" (AST).

L'analyseur doit être capable de gérer le nombre infini de possibles programmes valables qui peuvent lui être présentées. La manière habituelle de définir la langue est de spécifier une grammaire. Une grammaire est un ensemble de règles (ou productions) qui spécifie la syntaxe de la langue (cet est une phrase valable dans la langue).

Il peut y avoir plus d'une grammaire d'une langue donnée. En outre, il est plus facile de construire des analyseurs pour certains que pour d'autres grammaires.

Heureusement, il existe une classe de programmes qui peut prendre une grammaire et de générer un analyseur pour elle dans une langue. Des exemples de tels programmes sont les suivants : ANTLR (ciblage Java, C ++ et C #), (Java ciblage), YACC (C ciblage), YACC ++ (ciblage C ++)…etc.

Une **grammaire** est un formalisme permettant de définir une syntaxe et donc un langage formel, c'est-à-dire un ensemble de mots admissibles sur un alphabet donné. Par exemple, On peut définir la grammaire des expressions arithmétiques de la façon suivante :

```
exp ::= exp + exp
       | exp × exp
       | (exp)
       | num
num ::= chiffre num
       | chiffre
chiffre ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
9
```

FIGURE 2: GRAMMAIRE DES EXPRESSIONS ARITHMITIQUES

Les non-terminaux sont ici implicitement exp, num et chiffre, les terminaux sont +, ×, (,) et les chiffres. L'axiome est exp.
La dérivation suivante est un exemple d'utilisation de cette grammaire.
exp → exp × exp → num × exp → chiffre × exp → 3 × exp → 3 × num → 3 × chiffre num →3 × 1 num → 3 × 1 chiffre → 3 × 18

Définir un langage de programmation simple n'est pas très compliqué. Cette grammaire reconnaît un langage de programmation ressemblant à Pascal. Voici un exemple de programme calculant **fact(10)**

```
begin
  int a;
  int b;
  a:=10;
  b:=1;
while(a>1) do
    b:=a*b;
    a:=a-1;
  end;
print b;
end.
```

FIGURE 3: PROGRAMME SIMPLE EN PASCAL

La grammaire correspondante est :

```
  program    ::= ''begin'' listinstr ''end''
  listinstr  ::= instr listinstr
                 | instr
  Instr      ::= ''int'' id '';''
                 | id '':='' expr '';''
                 | ''print'' expr '';''
                 |  ''while''  ''(''  cond  '')''  ''do''
listinstr
            ''od'' '';''
  Expr       ::= expr ''-'' expr1| expr1
  expr1      ::= expr1 ''*'' expr2
                 | expr2
  expr2      ::= id
                 | num
                 | ''('' expr '')''
  cond       ::= expr condsymb expr
  condsymb   ::= ''>'' | ''<'' | ''>='' | ''<='' | ''!=''
                 | ''=''
```

FIGURE 4: GRAMMAIRE SIMPLE

1.2.3 ANALYSE SEMANTIQUE

Cet est à peu près l'équivalent de vérifier que certains texte ordinaire écrite dans un langage naturel (par exemple l'anglais) signifie réellement quelque chose (si oui ou non cet est ce qu'il a entend).

Le but de l'analyse sémantique est de vérifier que nous avons une séquence significative de jetons. Notez qu'une séquence peut être significative sans être correcte ; dans la plupart des langages de programmation, l'expression «x + 1 » serait considérée comme une expression arithmétique significative. Cependant, si le programmeur vraiment destiné à écrire « x – 1 », alors il ne est pas correct.

L'analyse sémantique est l'activité d'un compilateur pour déterminer quels sont les différents types de valeurs, comment ces types interagissent dans les expressions, et si ces interactions sont sémantiquement raisonnables. Par exemple, vous ne pouvez pas raisonnablement multiplier une chaîne de caractères par un nom de classe, même si aucun éditeur ne vous empêche de l'écriture :

```
"abc" * MyClass
```

Pour ce faire, le compilateur doit d'abord identifier les déclarations et les champs, et enregistre généralement le résultat de cette étape dans une série de tableaux de symboles. Ce qu'il dit ce que signifie identifiants spécifiques dans des contextes spécifiques. Il doit également déterminer les types de différentes constantes littérales ; "abc" est un type différent de celui 12.2E-5. (Bergmann, 2010)

1.2.4 GENERATION DE CODE INTERMEDIAIRE

Dans le procédé de traduction d'un programme source en un code cible, un compilateur peut construire une ou plusieurs représentations intermédiaires, qui peuvent avoir une variété de formes. Arbres de syntaxe sont une forme de représentation intermédiaire, ils sont couramment utilisés lors de l'analyse syntaxique et sémantique. Après l'analyse syntaxique et sémantique du programme source, de nombreux compilateurs génèrent un niveau faible, explicite ou représentation intermédiaire comme la machine, que nous pouvons considérer comme un programme pour une machine abstraite. Cette représentation intermédiaire devrait avoir deux propriétés importantes, il devrait être facile à produire et il devrait être facile à traduire dans la machine cible. (Alfred V. Aho, Monica S. Lam, Ravi Sethi, & Jeffrey D. Ullman, 2007)

Nous considérons une forme intermédiaire appelée code à trois adresses, qui consiste en une séquence d'instructions d'assemblage comme avec trois opérandes par instruction. Chaque opérande peut agir comme un registre. La sortie du générateur de code intermédiaire sur la Figure : 1 est constitué par la séquence de code à trois adresses :

```
t1 = inttofloat (60)
t2 = id3 * t1                                    (1.3)
```

```
t3 = id2 + t2
id1 = t3
```

1.2.5 OPTIMISATION DU CODE INTERMEDIAIRE

La phase d'optimisation de code tente d'améliorer le code intermédiaire, de sorte que l'amélioration de code cible entraînera. Habituellement meilleurs moyens plus rapides, mais d'autres objectifs peuvent être souhaitables, telles que le code plus court, ou un code cible qui consomme moins d'énergie. Par exemple, un simple algorithme génère le code intermédiaire (1.3), en utilisant une instruction pour chaque opérateur dans la représentation arborescente qui vient de l'analyseur sémantique. (Alfred V. Aho, Monica S. Lam, Ravi Sethi, & Jeffrey D. Ullman, 2007)

Un algorithme simple de génération de code intermédiaire suivie par l'optimisation du code est un moyen raisonnable pour générer un code de bonne cible. L'optimiseur peut en déduire que la conversion de 60 à partir de nombre entier en virgule flottante peut être fait une fois pour toutes au moment de la compilation, de sorte que l'opération "inttofloat" peut être éliminé par le remplacement de l'entier 60 par le nombre à virgule flottante 60,0. En outre, t3 est utilisé qu'une seule fois pour transmettre sa valeur pour ID1 afin que l'optimiseur peut transformer (1,3) dans la séquence plus courte :

```
t1 = id3 * 60.0
id1 = id2 + t1                                    (1.4)
```

1.2.6 GENERATION DE CODE

La génération de code natif est l'étape du processus de compilation transformant l'arbre syntaxique abstrait enrichi d'informations sémantiques en code machine ou en « bytecode » spécialisé pour la plateforme cible. C'est l'avant-dernière étape du processus de compilation qui se situe avant l'édition des liens. (Alfred V. Aho, Monica S. Lam, Ravi Sethi, & Jeffrey D. Ullman, 2007)

La phase de génération de code natif inclut généralement :

- Le choix des instructions à émettre ;
- L'ordonnancement des instructions : dans quel ordre émettre les instructions. L'ordonnancement est une optimisation de la vitesse d'exécution qui peut être critique pour les architectures pipelinées ;
- L'allocation de registres : l'allocation des variables aux registres du processeur.

Par exemple, utilisant des registres R1 et R2, le code intermédiaire pourrait être traduit dans le code de la machine :

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  Rl, id2
ADDF Rl, Rl, R2
STF  idl, Rl
```

CHAPITRE 2 : DEFINITION ET CONCEPTION DE LA LANGAGE

2.1 INTRODUCTION

Comme toutes les autres langages, la langage que nous utilisons besoin de définir des alphabets, des règles et de sémantique, avant de commencer, il faut préciser les mots-clés et la structure du langage, et cela se fait à travers la description et la définition du grammaire appropriées et l'automate d'analyse.

Nous avons sélectionné le langage algorithmique afin de réaliser notre compilateur, un algorithme est une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre un problème, Le mot **algorithme** vient du nom latinisé du mathématicien perse **Al-Khawarizmi**, écrivant en langue arabe, surnommé « le père de l'algèbre ». Le domaine qui étudie les algorithmes est appelé l'algorithmique. On retrouve aujourd'hui des algorithmes dans de nombreuses applications telles que la cryptographie, le routage d'informations, la planification et l'utilisation optimale des ressources, la bio-informatique, etc.

2.2 L'ANALYSE LEXICALE

Dans la Langage de Définition Algorithmique certains mots sont réservés pour un usage bien défini, on les nomme les mots-clés. Ce sont les mots que le langage utilise pour son fonctionnement, un mot clé ne peut pas être déclaré comme identificateur. Ils ne peuvent être utilisés comme variables

Les mots-clés du langage algorithmique sont spécifiques et bien connue par tous les informaticiens :

- **algorithme :** permet de définir ou de donner le nom à l'algorithme.
- **début** : marque le commencement de l'algorithme.
- **fin.** : marque la fin de l'algorithme
- **var** : c'est une partie de l'algorithme qui Permet de déclarer des variables **; une variable** est un objet dont le contenu peut changer au cours de l'exécution de l'algorithme.
- **const** : c'est une partie de l'algorithme qui permet de déclarer des constantes**. Un constant** est un objet dont le contenu reste invariant lors de l'exécution d'un algorithme.
- **réel, caractère, entier, chaine** et **booléen** sont des mots clés qui permettent de définir des types.

- **si, finsi, tantque, fintantque, pour, finpour, répéter, jusqu'à**… : mots clés permettant de définir les structures itératives, conditionnelles…
- Les fonctions d'entrées et de sorties **lire** et **écrire**
- L'affectation :=
- Les operations arithmitiques **+, -, /, *, ^**
- La comparaison **<, >, <=, >=, <>, =**
- Les **commentaires** sont utilisés pour permettre une interprétation aisée de l'algorithme. Leur utilisation est vivement conseillée. De ce fait, tout texte placé entre les symboles **/* */** sera considéré comme un commentaire.

algorithme	**var**	**caractère**	**booléen**	**tantque**	**finpour**
Début	**const**	**Entier**	**si**	**fintantque**	**répéter**
fin.	**réel**	**Chaine**	**finsi**	**pour**	**jusqu'à**
Lire	**écrire**	**Faire**			

FIGURE 5: LES MOTS-CLES DU LANGAGE D'ALGORITHMIQUE

De ce point, on pet définir la structure générale d'un algorithme comme suivant :

```
algorithme Nom_Algorithme;
var
     /*Déclarations des variables*/
const
     /*Liste des constants*/
début
     /*action 1*/
     /*action 2*/
     /*…*/
     /*action n*/
fin.
```

FIGURE 6: STRUCTURE GENERALE D'UN ALGORITHME

2.3 L'ANALYSE SYNTAXIQUE

Une grammaire est un ensemble de règles permettant de dire si une phrase, c'est-à-dire une suite de mots ou lexèmes (cf. l'analyse lexicale), est correcte ou non. Les règles qui nous intéressent en informatique sont celles qui donnent une description générative du langage, en indiquant comment précisément construire de telles phrases n'est pas génératif : elle ne nous dit pas comment effectivement construire une subordonnée relative.

```
Program      → algorithme Identifier ; Declarations
                    debut ListeInstrs fin.
ListeInstrs  → Instr ListeInstrs | ε
IdentListe   → Identifier | IdentListe , Identifier
Declarations → var DecListe | ε
DecListe     → IdentListe : Type ; | IdentListe : Type ; DecListe
      IdentListe [Expr] : Type ; | IdentListe [Expr] : Type ; DecListe
Type         → entier | reel | char | chaine | booleen
Instr        → PourInstr | TantqueInstr | AffectInstr | SiInstr
                | Lire | Ecrire | EcrireLn | ε
PourInstr    → pour Identifier := Expr a Expr faire Instr finpour
TantqueInstr → tantque Comp faire Instr fintantque
Comp         → Expr Comparaison Expr
Comparaison  → < | > | <= | >= | <> | =
AffectInstr  → Identifier := Expr ; | Identifier [Nombre] := Expr ;
SiInstr      → si Comp donc Instr SinonPart
SinonPart    → sinon Instr | ε
Lire         → lire ( LireArgs )
LireArgs     → Identifier | Identifier , LireArgs
Ecrire       → ecrire ( EcrireArgs )
EcrireLn     → ecrireln ( EcrireArgs )
EcrireArgs   → Expr | Expr , EcrireArgs
Expr         → + Expr | – Expr | Expr++ | Expr-- | Expr + Expr |
               Expr – Expr | Expr * Expr | Expr / Expr | ( Expr )
               | Identifier | IdentListe [Expr] | Donnée
Donnée       → Nombre | Caractère | Chaine_character | vrai | faux
```

FIGURE 7: LE GRAMMAIRE CORRESPONDANT A LA LANGAGE D'ALGORITHMIQUE

On représente l'identifies, les nombres, les caractères et les chaine de caractères par des expressions régulières :

```
Identifier       : [a-z][a-z0-9_]*
Nombre           : [0-9]+\.? [0-9]+?( [eE][+-]? [0-9]+)?
Caractère        : '.'
Chaine_character : "(.)* "
```

2.3 LA GENERATION DU CODE

La phase de génération de code natif doit prendre en compte au mieux les caractéristiques de la plateforme d'exécution (microprocesseur, machine virtuelle) pour générer du code qui s'y exécute le plus vite possible. Dans le cas d'un processeur, sa microarchitecture joue un rôle déterminant :

- Le compilateur doit favoriser l'unité de prédiction de branchement pour éviter les erreurs de branchements. Celles-ci sont très couteuses en cycles d'horloge car elles nécessitent de vider le pipeline et de le remplir à nouveau avant de pouvoir continuer l'exécution d'un programme. GCC permet aux programmeurs de préciser sur une instruction if quel résultat a le plus de chances d'arriver afin de privilégier une branche pour l'instruction de branchement suivante ;
- Le compilateur doit réarranger les instructions pour profiter au mieux du pipeline et éviter autant que possible les pauses dues aux dépendances de données ;
- Le compilateur doit réarranger les instructions afin d'éviter les caches miss qui nécessitent de récupérer des informations dans la mémoire vive, beaucoup plus lente que la mémoire cache.

Dans le cas de la compilation à la volée, la génération de code natif doit être rapide et consommer peu de mémoire afin de ne pas pénaliser l'exécution du programme compilé. Il faut lors utiliser des algorithmes différents que pour une compilation séparée de l'exécution. De plus, un compilateur JIT peut profiter des informations de profilage obtenues lors de l'exécution pour choisir quelles parties du code sont à optimiser au maximum.

Il y a beaucoup d'outils de génération de code, par exemple, Acceleo, AtomWeaver, Click2Code, CodeCooker ... etc

En fait, nous avons décidé de ne pas utiliser un de ces outils, nous allons générer directement un code d'assemblage, selon le code source de l'algorithme, et en utilisant un assembleur (masm32), nous allons générer un code exécutable (.exe), ce schéma explique tout :

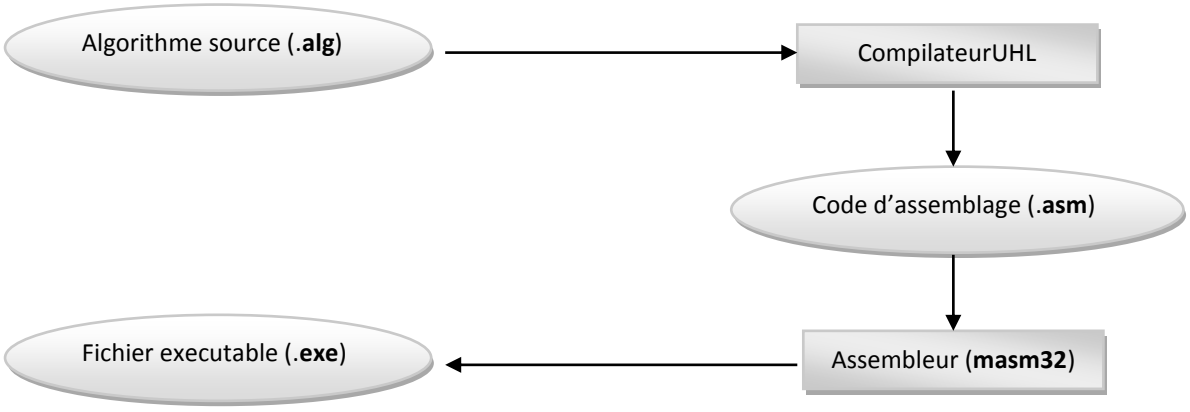

FIGURE 8 : LA GENERATION DU CODE

CHAPITRE 3 : IMPLEMENTATION ET MISE EN-ŒUVRE DU COMPILATEUR

3.1 INTRODUCTION

Le programmeur du compilateur, comme n'importe quel développeur de logiciels, peut utiliser avec profit les environnements de développement de logiciels modernes contenant des outils tels que les éditeurs de langue, débogueurs, des gestionnaires de version, profileurs, harnais de test, et ainsi de suite. En plus de ces outils généraux de développement de logiciels, d'autres outils plus spécialisés ont été créés pour aider à mettre en œuvre différentes phases d'un compilateur. (Alfred V. Aho, Monica S. Lam, Ravi Sethi, & Jeffrey D. Ullman, 2007)

Certains outils de construction de compilateur couramment utilisés comprennent :

1. Générateurs d'analyseurs lexicaux, qui produisent automatiquement les analyseurs de syntaxe d'une description grammaticale d'un langage de programmation.
2. Générateurs d'analyseurs syntaxiques qui produisent des analyseurs syntaxiques depuis la description d'expression régulière des jetons d'une langue.
3. Moteurs de traduction Syntaxe-dirigé qui produisent des collections de routines pour marcher un arbre d'analyse et de génération de code intermédiaire.
4. Générateurs de générateur de code produisant un générateur de code à partir d'un ensemble de règles pour traduire chaque opération du langage intermédiaire en langage machine pour une machine cible.

Pour atteindre notre objectif, nous décidons d'utiliser l'outil **Quex** pour générer l'analyseur lexical, et **Bison** afin de générer l'analyseur de syntaxe, le code généré est en C ++, mais nous allons écrire le code de table de symboles manuellement, et donc pour le générateur de code, notre compilateur va générer un code d'assemblage, ainsi, un assembleur est nécessaire de relier et de générer du code machine.

En outre, nous allons développer une interface utilisateur (GUI) et de le relier au compilateur pour écrire et exécuter de code sources Algorithmes.

L'ensemble d'outils et langages qu'on va utiliser sont : **Quex**, **Python 2.7 (nécessaire pour Quex), Bison, C++, C# .NET**
Dans une vision globale, notre compilateur avoir la structure suivante :

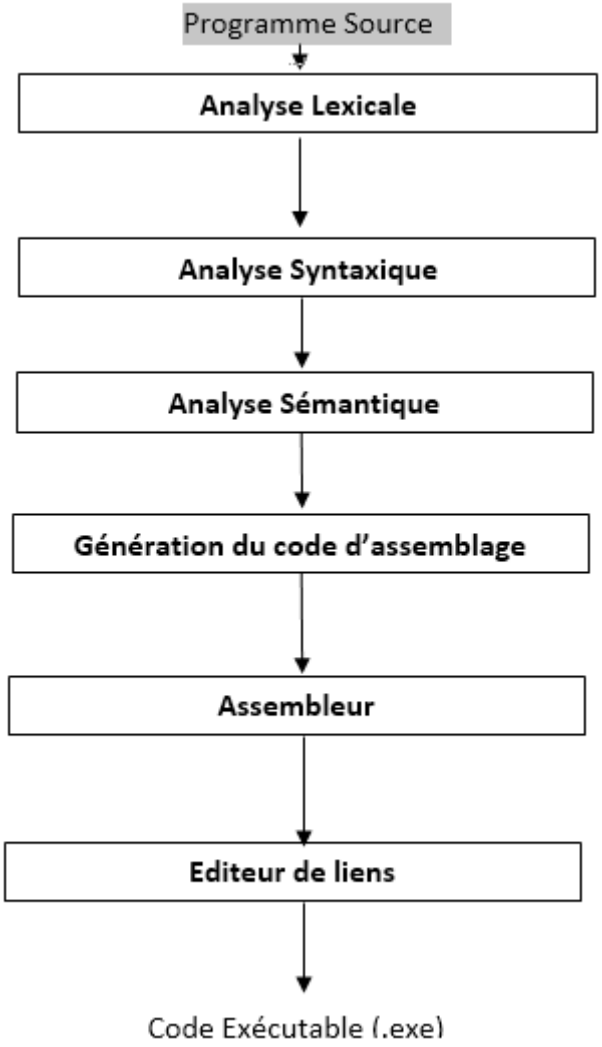

FIGURE 9: LA STRUCTURE DU COMPILATEUR A REALISE

3.2 GENERATION DE L'ANALYSEUR LEXICALE

Quex est peut être utilisé avec un autre utilitaire appelé Bison. Bison est un générateur d'analyseurs qui génère une fonction en **C ++, yyparse ()**, qui contient des appels à **compalgo_yylex ()** quand il veut lire un jeton. Cependant, il est également possible d'utiliser **Quex** indépendamment de Bison ou d'utiliser Bison indépendamment de **Quex.**

Quex peut être utile pour tout projet de développement de logiciel qui nécessite une analyse lexicale, pas simplement compilateurs. Par exemple, un langage de requête de base de données qui permet des déclarations telles que : RETRIEVE ALL RECORDS FOR SALARY >= $100,000 exigerait l'analyse lexicale et pourrait être mis en œuvre avec **Quex.**

3.2.1 QUEX SOURCE

L'entrée de **Quex** est stockée dans un fichier avec un suffixe .**qx** (comme pascal.qx). La structure de ce fichier, composé de trois sections, est présentée ci-dessous :

```
start = PROGRAM; //Section to start with
header {
      //C declarations and #includes
}
define {
      //Quex definitions
}
//Quex sections and actions
```

3.2.2 LES MODELS DE QUEX

Les modèles et les actions **Quex**, est la partie la plus importante. Cela est parfois appelé la section des règles parce que ces règles définissent les jetons lexicaux. Chaque ligne de cette section se compose d'un motif et une action. Chaque fois que la fonction **yylex_compalgo()** est capable de trouver l'entrée qui correspond à l'un des modèles, l'action correspondante est exécutée. Ce langage de schéma est juste une extension des expressions régulières et est décrit ci-dessous. Dans ce qui suit, x et y représentent n'importe quel motif.

Pattern	Meaning
C	Le caractère "c"
"c"	Le caractère "c", même si cet est un caractère spécial dans ce tableau
\c	Même que "c", utilisé pour citer un seul omble
[cd]	Le caractère c ou d
[a-z]	Tout caractère unique dans l'intervalle de a à z
[^c]	Tous caractères mais c
.	Tous caractères, mais retour à la ligne
^x	Le motif x se il se produit au début d'une ligne
x?	Une option x
x*	Zéro ou plusieurs occurrences Du motif x
x+	Un ou plusieurs occurrences du motif x
Xy	La concaténation motif d x avec le motif y
x|y	Un X ou un Y
(x)	un x
x/y	Un x seulement si elle est suivie par y

FIGURE 10 : TABLE DES MODELES QUEX

3.2.3 LA DEFINITION LEXICAL AVEC QUEX

La phase de l'analyse lexicale est implémenter avec **Quex**, donc, le fichier de définitions des Jetons et des règles lexicale est nommé **« compalgoLexer.qx »** :

```
start = PROGRAM;
define {
   P_WHITESPACE          [ \r\t\n]+
   P_IDENTIFIER          ([_a-zA-Z])([_a-zA-Z0-9])*
   P_NUMBER              [0-9]+(\.[0-9]+)?

   P_STRING              "\""(\\"\""|[^"])*"\""
   P_QUOTED_CHAR_1       ("'\\''")|("'"[^']?"'")
   P_QUOTED_CHAR_2       "'\\"[0-9abcfnrtv\\]"'"
   P_QUOTED_CHAR         ({P_QUOTED_CHAR_1}|{P_QUOTED_CHAR_2})
}
header {
      #include <fstream>
      extern std::ofstream  fresult;
      extern int line_number;
}
mode EOF_AND_FAILURE:
<inheritable: only>
{
      <<FAIL>>   {printf("Erreur lexicale:[%s]\n",Lexeme);}
}

mode PROGRAM : EOF_AND_FAILURE
<skip:       [ \r\n\t]>
<skip_range: "/*" "*/">
{
     "?"   => TKN_QUESTION_MARK();
     "("   => TKN_BRACKET_O();
     ")"   => TKN_BRACKET_C();
     "["   => TKN_CORNER_BRACKET_O();
     "]"   => TKN_CORNER_BRACKET_C();
     ":="  => TKN_OP_ASSIGNMENT();
     "+"   => TKN_PLUS();
     "-"   => TKN_MINUS();
     "*"   => TKN_MULT();
     "/"   => TKN_DIV();
     "^"   => TKN_EXPONENTIATION();
     "%"   => TKN_MODULO();
     "="   => TKN_EQ();
     "<>"  => TKN_NOT_EQ();
     ">"   => TKN_GREATER();
     ">="  => TKN_GR_EQ();
     "<"   => TKN_LESS();
     "<="  => TKN_LE_EQ();
     ":"   => TKN_COLON();
     ";"   => TKN_SEMICOLON();
     "."   => TKN_DOT();
     ","   => TKN_COMMA();

     "algorithme"=> TKN_ALGORITHME();
     "debut"               => TKN_DEBUT();
     "fin"            => TKN_FIN();
     "var"            => TKN_VAR();
     "const"               => TKN_CONST();
     "entier"         => TKN_INT();
     "reel"           => TKN_REEL();
```

```
    "booleen"         => TKN_BOOLEEN();
    "char"            => TKN_CHAR();
    "chaine"          => TKN_STRING();
    "si"              => TKN_IF();
    "alors"                => TKN_THEN();
    "sinon"                => TKN_ELSE();
    "finsi"                => TKN_ENDIF();
    "vrai"          => TKN_TRUE();
    "faux"            => TKN_FALSE();
    "pour"            => TKN_FOR();
    "a"                    => TKN_A();
    "finpour"       => TKN_ENDFOR();
    "faire"         => TKN_DO();
    "tantque"       => TKN_WHILE();
    "fintantque"    => TKN_ENDWHILE();
    "ecrire"        => TKN_PRINT();
    "ecrireLn"      => TKN_PRINTLINE();
    "lire"          => TKN_SCAN();

    {P_NUMBER}      => TKN_NUMBER(Lexeme);
    {P_STRING}      => TKN_STRING(Lexeme);
    {P_QUOTED_CHAR} => TKN_QUOTED_CHAR(Lexeme);
    {P_IDENTIFIER}  => TKN_IDENTIFIER(Lexeme);
    .                  {fresult<<"Erreur:    symbole    inconnu:
["<<Lexeme<<"] : ligne "<<line_number<<std::endl;}
}
```

FIGURE 11: LE FICHIER COMPALGOLEXER.QX

Le modèle P_NUMBER est défini comme étant une chaîne d'un ou plusieurs chiffres, la partie exposant d'un nombre est optionnel.

P_IDENTIFIER défini comme étant le modèle pour les identificateurs, ([_A-zA-Z])([_ a-zA-Z0-9])*, qui indique qu'un identifiant est une chaîne de lettres, de chiffres numériques, et souligne ce qui commence par une lettre (soit minuscule ou majuscule).

Le fichier qui contient les définitions des Jetons est « compalgoParser.tab.hpp » Et il est généré par la suite par Bison.

3.2.3 EXECUTION DE QUEX

Afin de générer le code source finale en C++ de l'analyseur lexical, en exécuter cette commande :

C:\> quex -i compalgoLexer.qx -o compalgoLexer --foreign-token-id-file compalgo_token-ids.h --token-prefix TKN_

3.3 TABLE DE SYMBOLES

La table de symboles est une centralisation des informations rattachées aux identificateurs d'un programme informatique. C'est une fonction accélératrice de compilation, dont l'efficacité dépend de la conception. Dans une table des symboles,

on retrouve des informations comme : le type, l'emplacement mémoire, la portée, la visibilité, etc.

La table de symboles est créée dynamiquement. Une première portion est créée au début de la compilation. Puis, de façon opportuniste, en fonction des besoins, elle est complétée.

La première fois qu'un symbole est vu (au sens des règles de visibilité du langage), une entrée est créée dans la table.

La classe qui définir la structure de la table des symboles est **CSymbolTable**, elle est implémentée dans les fichiers **SymbolTable.h** et SymbolTable.cpp.

```
typedef struct {
	std::string name;
	int  type;
	int  max_index;
	bool isConst;
}stEle;

class CSymbolTable: public std::list<stEle>{
public:
	int current_type;
public:
	CSymbolTable(void);
	~CSymbolTable(void);
	stEle* operator [] (int) const;
	stEle* operator [] (std::string) const;
	void putSym(std::string, int, bool);
};
```

FIGURE 12 : LA CLASSE CSYMBOLTABLE

Quand l'utilisateur déclare un variable dans l'algorithme, le compilateur fait un appelle à la fonction **putSym()** afin d'installer ce symbole dans la table des symboles, donc, si ce variable est déclaré précédemment, un message d'erreur apparaît dit : « Erreur: la variable x est déjà déclaré: ligne 12 ».

3.4 GENERATION DE L'ANALYSEUR SYNTAXIQUE / SEMANTIQUE

Bison génère une fonction en **C ++ nommé yyparse(),** qui est stocké dans un fichier nommé « compalgoParser.tab.cpp ». Cette fonction appelle une fonction nommée **compalgo_yylex()** chaque fois qu'il a besoin d'un jeton d'entrée. La fonction **compalgo_yylex()** peut être écrit par l'utilisateur et inclus dans le cadre de la spécification Bison.

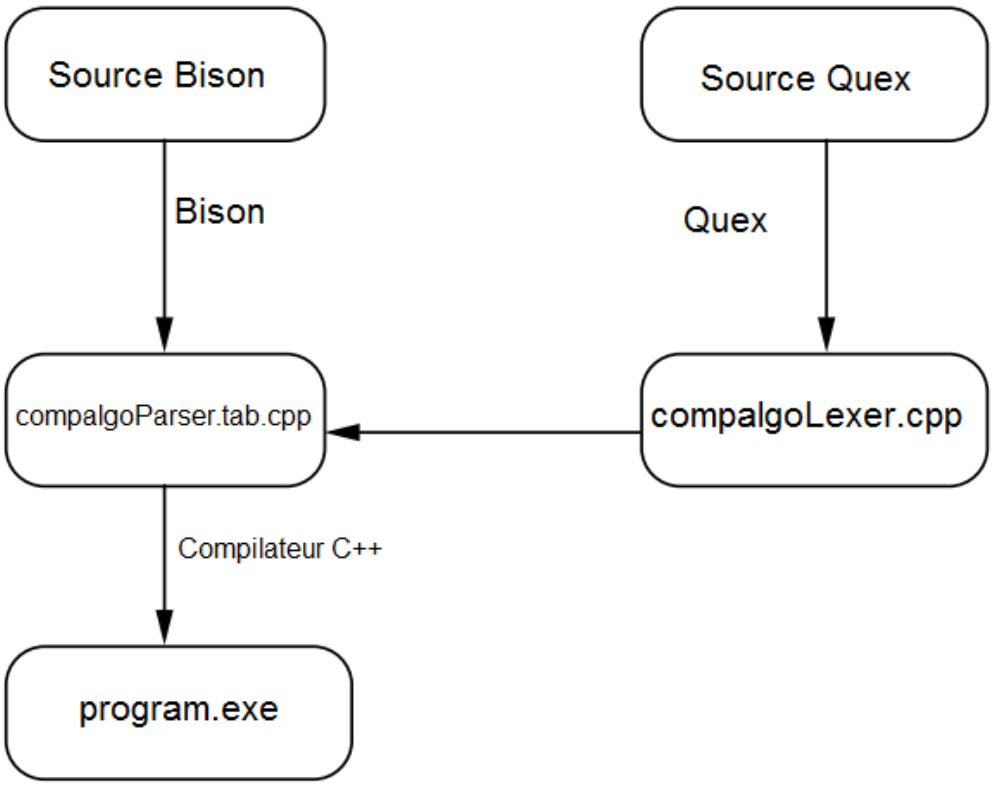

FIGURE 13 : GENERATION ET COMPILATION D'UNE APPLICATION AVEC QUEX ET BISON

3.4.1 STRUCTURE D'UN FICHIER SOURCE DE BISON

L'entrée de Bison est appelé le fichier source Bison. Il se compose de trois parties, qui sont séparées par le séparateur de %% :

```
Déclarations
%%
Règles
%%
Routines de support
```

La section Déclarations contient des déclarations des noms symboliques, le type de pile, et des informations de priorité qui peut être nécessaire par Bison. Il peut également contenir des énoncés de préprocesseur (#include ou #define) et les déclarations devant être inclus dans le fichier de sortie « compalgoParser.ypp ».

La section Règles est la grammaire de la langue étant précisé, comme Pascal. C'est la partie la plus importante du fichier source de Bison. Chaque règle est de la forme :

```
nonterminale:  α {action}
             | β {action}
             | γ {action}
             .
             .
             .
             ;
```

Où α, β, et γ sont des définitions du non-terminal. La barre verticale désigne définitions alternatives pour un non-terminal, comme dans BNF. Une action peut être associée à chacune des alternatives. Cette action est tout simplement une déclaration de C qui est invoqué lors de l'analyse d'une chaîne d'entrée lorsque la règle de grammaire correspondant est réduite. Les règles peuvent être écrites en format libre, et chaque règle est terminée par un point-virgule.

La troisième section du fichier source de Bison contient des routines de soutien, c'est à dire, les fonctions en C++ qui pourraient être appelés à partir des actions dans la section Règles. Par exemple, lors du traitement d'une instruction d'affectation, il peut être nécessaire de vérifier que le type de l'expression correspondant au type de la variable à laquelle il est affecté. Cela pourrait se faire par un appel à une fonction de vérification de type dans la troisième section du fichier source de Bison.

3.4.2 LA DEFINITION DU SYNTAXE AVEC BISON

Le fichier **« compalgoParser.ypp »** contient le code de l'analyseur syntaxique, sémantique et le générateur de code sont tous, donc les actions syntaxiques sont attachées avec les actions sémantiques, et en même temps on génère le code d'assemblage propre à chaque niveau d'analyse.
Tout d'abord, nous allons déclarer une instance de l'objet **"CSymbolTable"** et **"CCodeGenerator"** :

```
extern char outfile[256];
CSymbolTable ST;
CCodeGenerator codeGenerator(outfile);
```

« outfile » est le chemin du fichier de sortie du code d'assemblage final.

Nous allons définir la fonction **« setupSym »**, cette fonction sera installer les variables déclarées dans le tableau de symboles, si un variable est déjà déclaré, un message d'erreur apparaît, se il n'y a pas d'erreur de la déclaration, le code d'assemblage approprié sera généré :

```
int setupSym(std::string sym_name,int buffer_size,bool isConst)
{
 if((ST.current_type>4)&&(buffer_size<1)){
        errorsNumber++;
        fresult<<"Erreur: La taille de la table ("<<sym_name<<") doit etre
superieur a 0 : ligne "<<line_number<<std::endl;
 }
 else if(ST[sym_name]==NULL){
       ST.putSym(sym_name,buffer_size, isConst);
       char *buffer_size_string = new char[4];
       itoa(buffer_size, buffer_size_string, 10);
       if(ST.current_type == _int)
       {
              codeGenerator.addVar((char*)sym_name.c_str());
              codeGenerator.addVar("\tSDWORD\t6 dup(?)\n");
       }
       else if(ST.current_type == _float)
       {
              codeGenerator.addVar((char*)sym_name.c_str());
              codeGenerator.addVar("\tSDWORD\t6 dup(?)\n");
       }
       else if(ST.current_type == _chr)
```

```
        {
                codeGenerator.addVar((char*)sym_name.c_str());
                codeGenerator.addVar("\tdb\t?\n");
        }
        else if(ST.current_type == _str)
        {
                codeGenerator.addVar((char*)sym_name.c_str());
                codeGenerator.addVar("\tdb\t256 dup(?)\n");
        }
        else if(ST.current_type == _int_tab)
        {
                codeGenerator.addVar((char*)sym_name.c_str());

        codeGenerator.addVar("\tdw\t");codeGenerator.addVar(buffer_size_strin
g);codeGenerator.addVar(" dup(?)\n");
        }
        else if(ST.current_type == _float_tab)
        {
                codeGenerator.addVar((char*)sym_name.c_str());

        codeGenerator.addVar("\tdw\t");codeGenerator.addVar(buffer_size_strin
g);codeGenerator.addVar(" dup(?)\n");
        }
 }
 else
 {
        errorsNumber++;
        fresult<<"Erreur: la variable "<<sym_name<<" est deja declare: ligne
"<<line_number<<std::endl;
        return 0;
 }
 return 1;
}
```

Une autre fonction devrait être définie, "**symCheck**", cette fonction sera appelée à chaque fois une variable utilisée, pour le vérifier, déclarée ou non:

```
int symCheck(std::string sym_name)
{
 if(ST[sym_name]==NULL) {
        errorsNumber++;
        fresult<<"Erreur:   variable   inconnue   "<<sym_name<<"   :   ligne
"<<line_number<<std::endl;
        return 0;
 }
 return 1;
}
```

Puis, nous allons définir tous les jetons dont nous avons besoin lorsque nous représentons la grammaire de la langue :

```
%token                  TKN_QUESTION_MARK          TKN_BRACKET_O
TKN_BRACKET_C TKN_CORNER_BRACKET_O TKN_CORNER_BRACKET_C
%token                  TKN_OP_ASSIGNMENT TKN_PLUS TKN_MINUS
%token                  TKN_MULT        TKN_DIV        TKN_MODULO
TKN_ASSIGN_PLUS TKN_ASSIGN_MINUS
%token                  TKN_ASSIGN_MULT   TKN_ASSIGN_DIV   TKN_EQ
TKN_NOT_EQ TKN_GREATER TKN_GR_EQ
```

```
%token                  TKN_LESS   TKN_LE_EQ   TKN_OR   TKN_COLON
TKN_SEMICOLON TKN_DOT
%token                  TKN_COMMA  TKN_EXPONENTIATION  TKN_REEL
TKN_BOOLEEN TKN_ENDFOR TKN_ENDWHILE TKN_ENDIF
%token      <TYPE>      TKN_IDENTIFIER   TKN_NUMBER   TKN_STRING
TKN_QUOTED_CHAR TKN_INT TKN_CHAR TKN_TRUE TKN_FALSE
%token                  TKN_IF    TKN_ELSE    TKN_FOR    TKN_DO
TKN_PRINTLINE
%token                  TKN_WHILE       TKN_PRINT       TKN_SCAN
TKN_ALGORITHME   TKN_FIN   TKN_VAR   TKN_CONST   TKN_A   TKN_THEN
TKN_DEBUT
%type <TYPE>      Expr Type
%type <TYPE>      Donnee Comparaison
```

La grammaire est représenté exactement comme ci-dessus, avec une syntaxe qui est si proche de la BNF :

```
Program    :    TKN_ALGORITHME    TKN_IDENTIFIER    TKN_SEMICOLON
Declarations TKN_DEBUT ListeInstrs TKN_FIN TKN_DOT {}
;
Declarations: {}
      | TKN_VAR DecListe {}
;
.
.
ListeInstrs : {}
      | Instr ListeInstrs {}
;
Instr :PourInstr {}
      | TantqueInstr {}
      | AffectInstr {}
      | SiInstr {}
      | Lire {}
      | Ecrire {}
      | EcrireLn {}
;
PourInstr  :  TKN_FOR  TKN_IDENTIFIER  TKN_OP_ASSIGNMENT  Expr
TKN_A Expr TKN_DO ListeInstrs TKN_ENDFOR {}
;
.
.
/* etc */
```

Comme nous le voyons, des actions sémantiques sont entre { }, y compris les instructions de génération de code, par exemple :

```
Expr : Expr TKN_PLUS {codeGenerator.mathOperation = PLUS;} Expr
 {
      if(($1.type != _int) && ($1.type != _float)){
            errorsNumber++;
```

```
                fresult <<"Erreur : Cette operation ne est
applicable que sur les variables du type entier ou reel. ligne:
"<<line_number<<std::endl;
        }
        if(($4.type != _int) && ($4.type != _float)){
                errorsNumber++;
                fresult <<"Erreur : Cette operation ne est
applicable que sur les variables du type entier ou reel. ligne:
"<<line_number<<std::endl;
        }
        $$.type = ($1.type == _float)?_float:$4.type;
        codeGenerator.mathOperation = NONE;
 }
```

si il y a une erreur de syntaxe, ou l'ordre de l'entrée ne respecte pas la grammaire (ou l'automate correspondante), la fonction **compalgo_yyerror()** affichera un message d'erreur:

```
void compalgo_yyerror(quex::compalgoLexer *qlex, const
std::string& m)
{
        errorsNumber ++;
        fresult <<"Erreur : "<<m<<" : ligne
"<<line_number<<std::endl;
}
```

la fonction **compalgo_yylex()** est responsable de la liaison Bison-Quex, en demandant –à chaque fois- la prochaine Jeton (Token) de Quex:

```
int compalgo_yylex(YYSTYPE *yylval, quex::compalgoLexer
*qlex)
{
        quex::Token* token;
        qlex->receive(&token);
        line_number = token->line_number();
        if (token->get_text().length()>0)
        {
                yylval->TYPE.str = new std::string((char
*)token->get_text().c_str());
        }
        return (int)token->type_id();
}
```

3.4.3 Execution de bison

Afin de générer le code source finale en C++ de l'analyseur syntaxique/sémantique et le générateur de code, en exécuter cette commande :

C:\> bison -d compalgoParser.ypp

3.4.4 L'ANALYSE SEMANTIQUE

L'analyse sémantique promenade main à la main avec l'analyse de la syntaxe, ainsi, dans toutes les règles de syntaxe, nous allons vérifier la sémantique et afficher les messages d'erreurs, par exemple :

```
Expr : Expr TKN_PLUS Expr    {
     if(($1.type != _int) && ($1.type != _float)){
     errorsNumber++;
     fresult <<"Erreur : Cette operation ne est applicable
que sur les variables du type entier ou reel. ligne:
"<<line_number<<std::endl;
     }
}
```

L'exemple ci-dessus montre que l'opération "Addition" arithmétique mais n'est pas applicable que sur des variables et des valeurs de type "entier" ou "réel", et ainsi de suite.

3.5 GENERATION DU CODE D'ASSEMBLAGE

Jusqu'à ce point, nous avons ignoré l'architecture de machine pour laquelle nous construisons le compilateur, c'est à dire la machine cible, en architecture, on entend la définition de l'unité centrale de traitement de l'ordinateur tel que vu par un programmeur de langage machine. Spécifications des opérations de jeu instructions, formats d'instruction, les modes d'adressage, les formats de données, les registres du CPU,les instructions d'entrée / sortie, etc.

Nous allons générer un code d'assemblage pour les déclarations initiales, un assembleur prend le relais pour générer du code exécutable pour nous.

La classe **CodeGenerator** -qui est définit dans le fichier « CodeGenerator.cpp »- contient les fonctions nécessaire pour faire cette tâche.

```
class CCodeGenerator
{
public:
       std::ofstream  fcode;
       char constSection[1024];
       char dataSection[1024];
       char varsSection[1024];
       char codeSection[8192];
       int Operation;
       int ControlFlowOp;
       int CmpOp;
       int CmpOpOperand;
       int mathOperation;
private:
       int _msg_x;
       int _label_x;
       char* GenStrIdent();
       char* GenLabel(char *);
public:
```

```
        CCodeGenerator(void);
        CCodeGenerator(char *outfile);
        ~CCodeGenerator(void);
        void initCode(char *);
        void addConst(char *);
        void addData(char *);
        void addVar(char *);
        void addCode(char *);
        void disposeCode();
        void generateCode(std::string str, ...);
};
```

FIGURE 14 : LA CLASSE CCODEGENERATOR

L'utilisation de cette classe est facile, tout d'abord, nous déclarons un objet instancié de lui nommé **« codeGenerator »** :

```
CCodeGenerator codeGenerator(outfile);
```

Le constructeur de la classe va ouvrir le chemin du fichier transmis par **"outfile",** après cela, à chaque pas fait lors de l'analyse syntaxique, le générateur de code génère le code d'assemblage approprié pour l'action.

Le Code de l'Assemblée (dans notre cas) a deux sections principales : la section Variables, section de constantes, la section de données et de la section de code, dans l'autre main, la classe de générateur de code **« CCodeGenerator »** a quatre fonctions qui sont responsable pour ajouter le code approprié pour chaque section**: addConst, addData, addVar, addCode.**
Par exemple, lorsque l'utilisateur déclare une variable dans le code de l'algorithme, un appel sera fait à **addVar** afin de générer le code assembleur appropriée :

```
if(ST.current_type == _int)

        {
                codeGenerator.addVar((char*)sym_name.c_str());
                codeGenerator.addVar("\tSDWORD\t6 dup(?)\n");
        }
```

Cela va convertir **"x: entier;" à "x 6 SDWORD dup (?)"**

Avancer de plus en plus, lorsque le scanner est livré avec la fonction "lire", il annoncera au début de la règle selon laquelle il est dans une fonction de lecture :

```
Lire: TKN_SCAN {codeGenerator.Operation = SCAN;} TKN_BRACKET_O LireArgs
TKN_BRACKET_C TKN_SEMICOLON {codeGenerator.Operation = NONE;};
```

Cela va forcer le générateur de code pour générer un code d'assemblage pour lire les variables, de sorte que lorsque la fonction **"generateCode"** est appelé, il va générer le code approprié.

```
case SCAN:
        switch(type){
                case _str:
                addCode("invoke StdIn,ADDR ");
                addCode((char*)str.c_str());addCode(",128\n");
                addCode("mov BYTE PTR [");
                addCode((char*)str.c_str());
                addCode("+eax], 0\n");
```

```
addCode("invoke StripLF,ADDR ");
addCode((char*)str.c_str());addCode("\n");
break;
```

En supposant que l'utilisateur a tapé :

```
var x:entier;
ecrire("Entrez x = ");
lire(x);
ecrire("x = ", x);
```

Le code assembleur généré sera :

```
.DATA?
x   SDWORD     6 dup(?)
.CODE
print chr$("Entrez x = ")
invoke StdIn,ADDR number,12
mov BYTE PTR [number+eax], 0
invoke StripLF,ADDR number
mov x, sval(ADDR number)
print chr$("x = ")
print str$(x)
```

3.6 Assemblage est Edition de liens

L'édition des liens est un processus qui permet de créer des fichiers exécutables ou des bibliothèques dynamiques ou statiques, à partir de fichiers objets.

La compilation en fichiers objets laisse l'identification de certains symboles à plus tard. Avant de pouvoir exécuter ces fichiers objets, il faut résoudre les symboles et les lier à une bibliothèque. Le lien peut être :

- statique : le fichier objet et la bibliothèque sont liés dans le même fichier exécutable ;
- dynamique : le fichier objet est lié avec la bibliothèque, mais pas dans le même fichier exécutable ; les liens sont établis lors du lancement de l'exécutable.

Les assembleurs sont généralement livrés avec un programme chargé de faire ce travail (appelé **linker** ou éditeur de liens).

Enfin, nous avons besoin d'assembler le code d'assemblage (généré dans le fichier **alg.asm**) pour obtenir un fichier exécutable, en utilisant masm32 avec ces commandes :

```
C:\> ml.exe /c /coff /Cp /nologo alg.asm
C:\> link.exe /SUBSYSTEM:CONSOLE /RELEASE /VERSION:4.0 alg.obj
```

L'opération d'édition des liens se fait implicitement par le programme link.exe, Après cela, nous allons obtenir un fichier exécutable nommé "**alg.exe**".

3.6 DEVELOPPEMENT DE L'INTERFACE UTILISATEUR

Il est très important de développer une interface utilisateur graphique (GUI), et se rendre compte que, nous avons décidé d'utiliser C # .NET avec l'environnement de développement Microsoft Visual Studio 2010.
L'interface fera la création de fichiers de l'algorithme, la compilation et l'exécution plus facile et plus rapide et d'autres fonctionnalités comme la coloration syntaxique, c'est une interface simple et élégante.

3.6.1 A PROPOS DE MICROSOFT VISUAL C#.NET

Visual C# est un outil de développement édité par Microsoft, permettant de concevoir des applications articulées autour du langage C#.

Visual C# propose les outils pour développer des applications C# hautement performantes qui ciblent la plateforme nouvelle génération de Microsoft pour la programmation distribuée et compatible Internet. Ce langage de programmation est simple, de type sécurisé et orienté objet. Il a été conçu pour générer des applications d'entreprise. Le code écrit en C# est compilé en code managé exécuté sous le Framework .NET.

3.6.2 DESCRIPTION DE L'INTERFACE UTILISATEUR

Une interface d'utilisateur est un arrangement de conception logicielle pour permettre le couplage de composants. Pour une bibliothèque logicielle on parle d'interface de programmation ou API, permettant le couplage entre un programme et la librairie. Pour une classe, un objet ou un module logiciel, on parle simplement d'interface ; cette interface permet le couplage entre classes, objets ou modules. L'interface qui est présentée à l'utilisateur est nommée interface utilisateur. Elle donne accès aux fonctions du programme par le biais d'un clavier, d'une souris ou d'un écran tactile tout en les représentant d'une manière graphique (couplage entre l'homme et la machine).

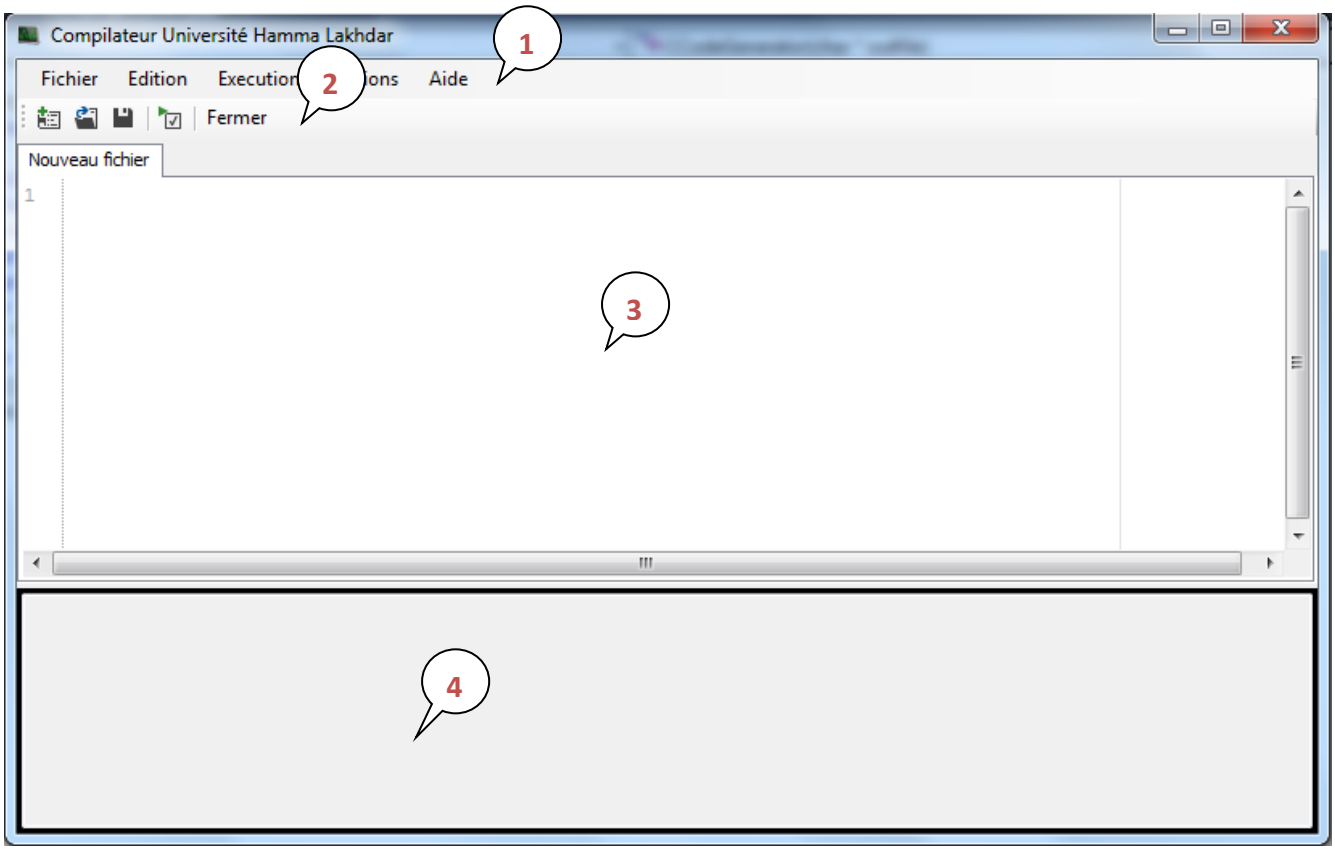

FIGURE 15 : L'INTERFACE DE L'UTILISATEUR

1. Le menu principal de l'application ;
2. La barre d'outils principale ;
3. Une zone de texte pour la saisie d'algorithmes ;
4. Une zone pour afficher les erreurs et le résultat de compilation ;

3.6.3 COMMENT COMPILER UN ALGORITHME ?

Tout d'abord, vous devez saisir le texte de l'algorithme, ou ouvrez un fichier d'algorithme **(extension .algo)** et appuyez sur "exécuter" sous le menu "Exécution", ou appuyez simplement sur **Ctrl+F9.**

Par exemple, cette capture d'écran montre un algorithme simple et son résultat de compilation.

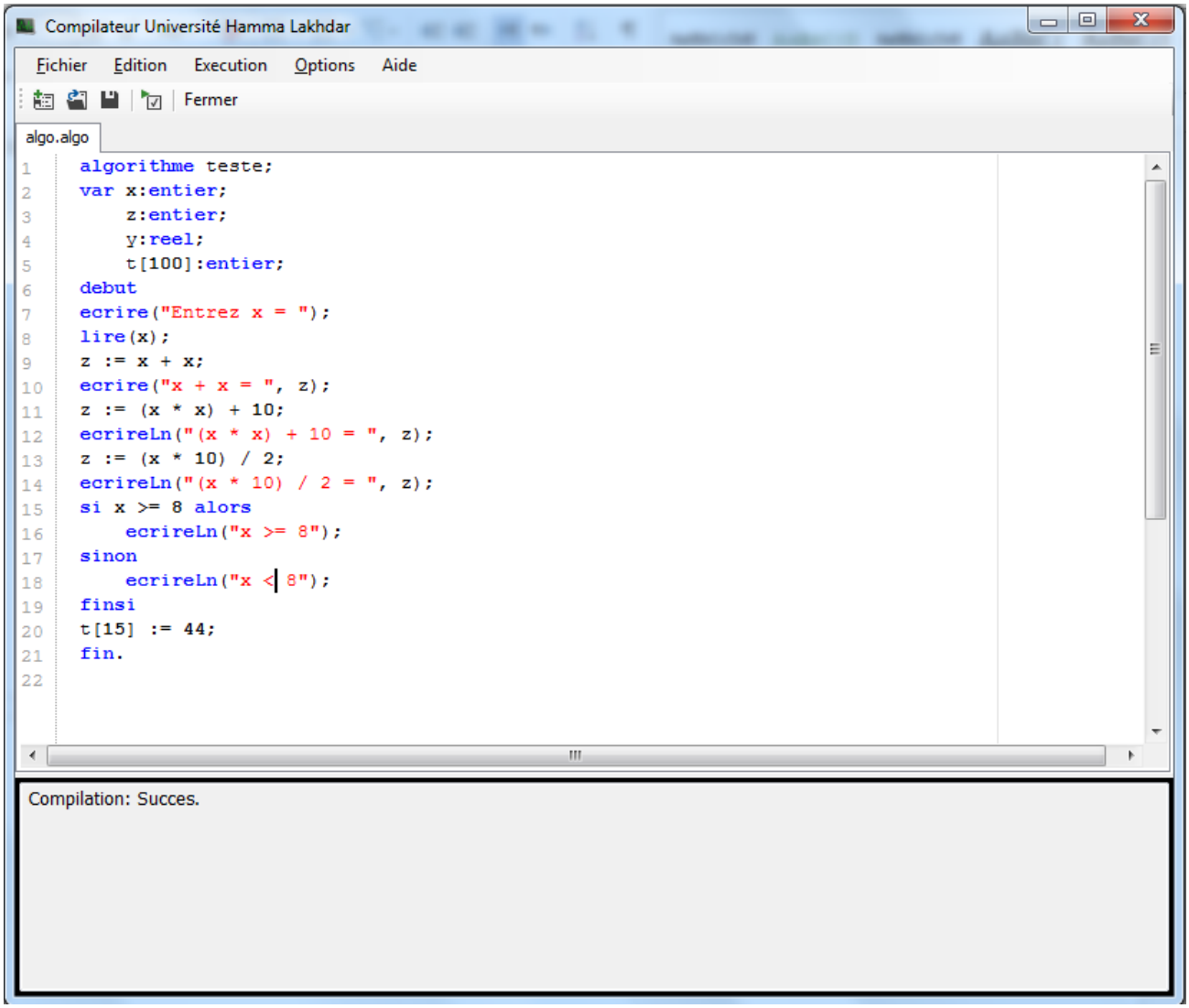

FIGURE 16 : EXAMLPE DE SAISIE D'ALGORITHME

FIGURE 17 : EXECUTION D'UN ALGORITHME

3.6.4 GESTION D'ERREUR DE LA COMPILATION

S'il y a des erreurs dans l'algorithme source, les messages d'erreur s'affiche dans la zone de texte en bas, par exemple, prenons l'algorithme précédent et faire quelques ajustements afin d'afficher les messages d'erreur :

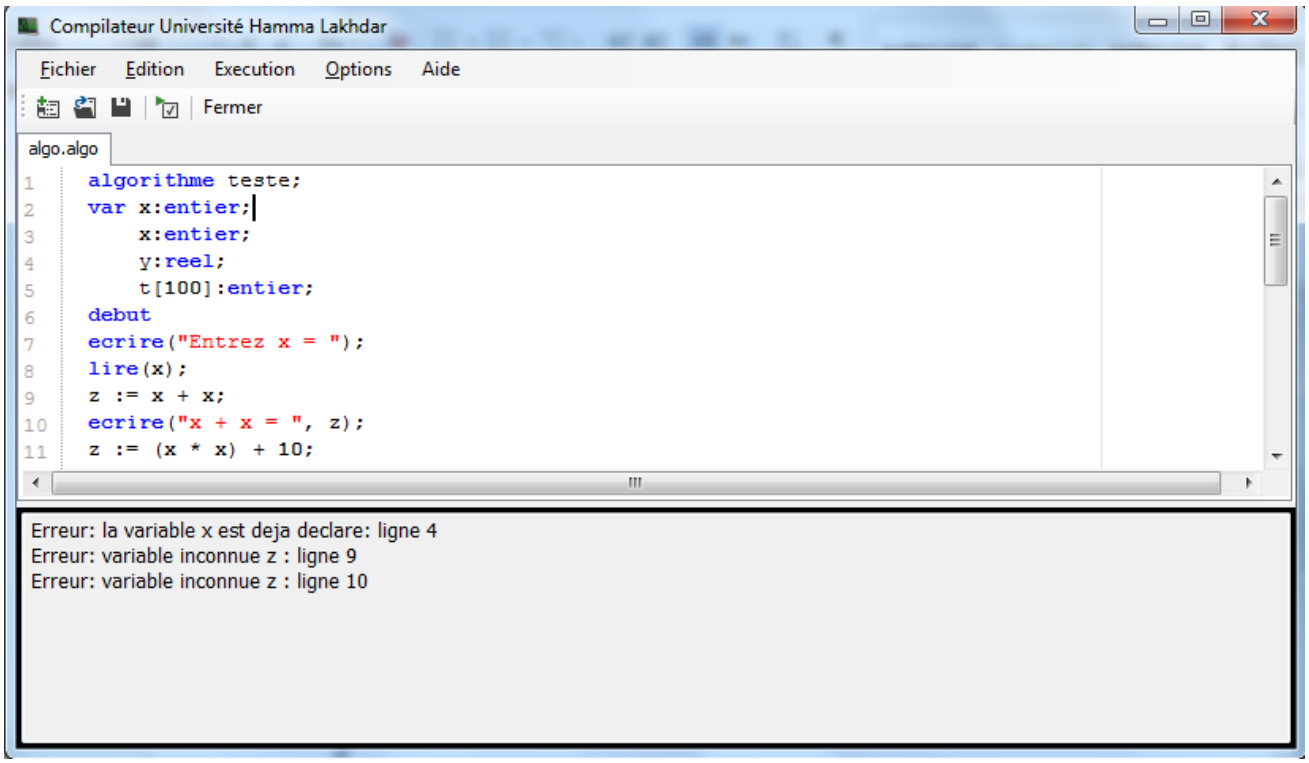

FIGURE 18 : LES MESSAGE D'ERREURS

La partie de code algorithme erroné est :

```
algorithme teste;
var   x:entier;
      x:entier;
      y:reel;
      t[100]:entier;
debut
ecrire("Entrez x = ");
lire(x);
z := x + x;
ecrire("x + x = ", z);
z := (x * x) + 10;
```

Le variable x est déclarée deux fois, ce qui est une erreur, de l'autre côté, la variable z est utilisée sans avoir été déclarée, la ligne 9 et la ligne 10 comme il est indiqué.

Aussi le compilateur de détecter l'incompatibilité de type :

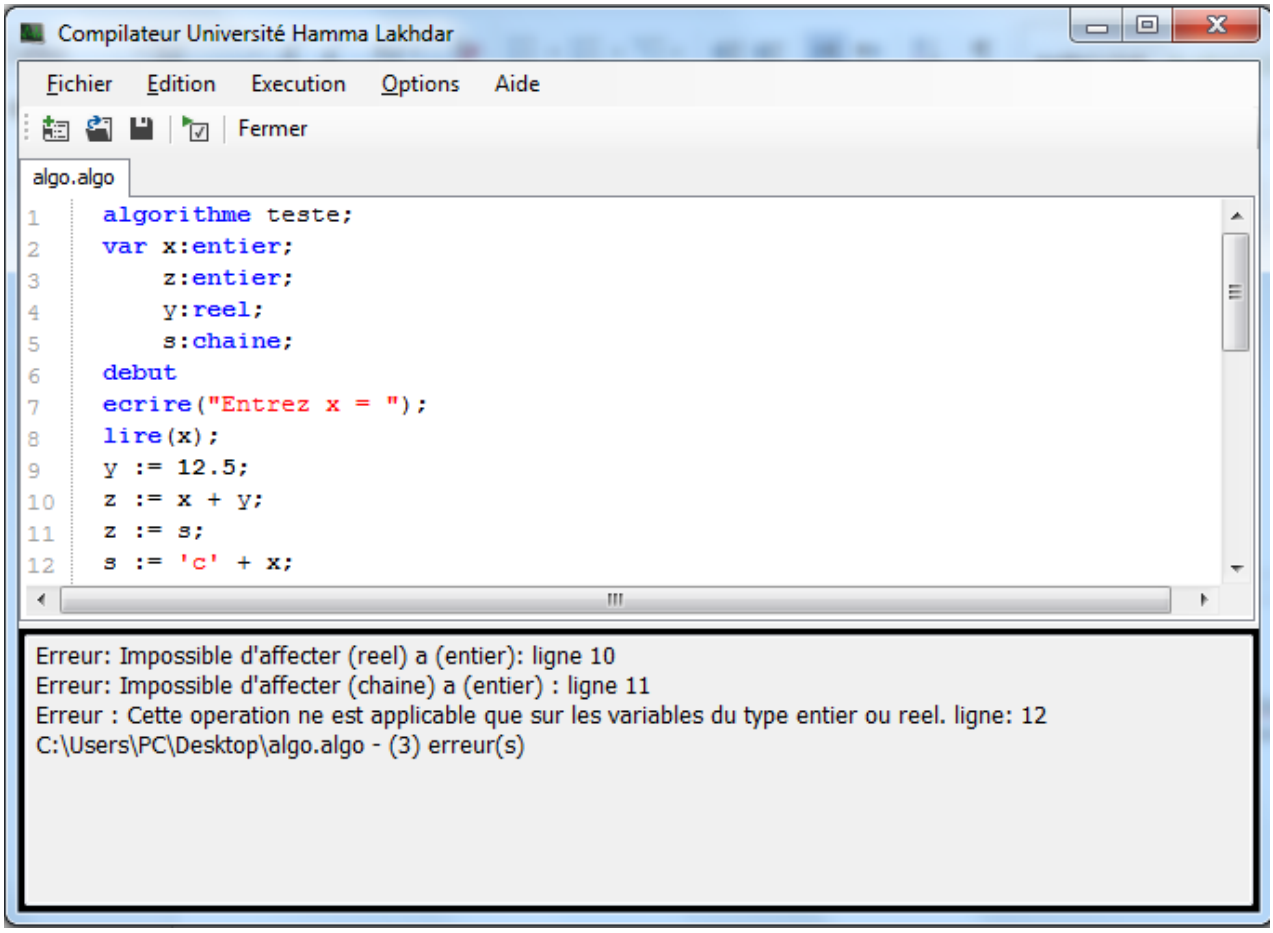

FIGURE 19 : ERREURS D'INCOMPATIBILITE DE TYPE

Après une compilation de succès, le compilateur génère un programme exécutable de l'algorithme d'origine, nommé **[Fichier_algo].exe,** par exemple, algo.exe compte tenu de l'exemple précédent.

CONCLUSION GENERALE

Nous avons développé un compilateur pour compiler les codes sources algorithmiques en utilisant les outils : **Quex** et **Bison** et les langages de programmation C++ et Microsoft Visual C# (.NET).

Nous avons obtenus des résultats plausibles de point de vue objectif et subjectif, algorithmes peuvent être éditées et exécutées facilement.

Comme perspective, nous proposons d'améliorer notre compilateur afin de gérer plus syntaxe algorithmique, la suggestion de soutien et de correction automatique d'erreurs et plus.

BIBLIOGRAPHIES

Alfred V. Aho, Monica S. Lam, Ravi Sethi, & Jeffrey D. Ullman. (2007). Compilers : principles, techniques, and tools. Boston: PEARSON Addison Wisley.

Bergmann, S. D. (2010). Compiler Design: Theory, Tools, and Examples. Rowan: Rowan University.

Ferber, J. (1997). Cours de compilation. Montpellier: Université de Montpellier II.

Yacine, K. (2010). Comment programmer un compilateur (Edition Numérique en Arabe). Ouargla, Algerie.

Yunlin Su, & Song Y. Yan. (2011). Principles of Compilers, A New Approach to Compilers Including the Algebraic Method. Java Timur, Indonesia: Springer.