Rapport de mini-projet Technique de compilation

Par :

Roua Ben Emna, Joueme Ben Said et Mohamed Rayen Khlifi

Réalisation d'un mini-compilateur Python

Année universitaire : 2022/2023

TABLE DES MATIÈRES

TABLE DES FIGURES

INTRODUCTION GÉNÉRALE

Ce projet intitulé Réalisation d'un mini-compilateur python "nimble#" s'inscrit dans le cadre de la matière Technique de compilation, élaboré par : Roua Ben Emna, Joueme Ben Said et Mohamed Rayen Khlifi étudiants en 1re ING 1 à l'Institut supérieur d'informatique à Ariana. Dans ce rapport, nous allons présenter en détail ce projet en commençant par une introduction qui mettra en évidence l'importance de la réalisation d'un mini-compilateur. Nous décrirons également l'environnement logiciel que nous avons utilisé pour la conception du projet, ainsi que les étapes nécessaires pour son installation. Nous discuterons également de l'architecture du mini-compilateur, qui comprendra l'analyseur lexical, syntaxique et sémantique. Nous aborderons ensuite les fonctionnalités que nous avons implémentées. Ensuite, nous présenterons la partie de réalisation, où nous exposerons les différentes étapes de développement Finalement nous clôturons par une conclusion.

CHAPITRE 1

INTRODUCTION

Introduction

Pendant ce premier chapitre introductif, on va s'intéresser à la définition d'un compilateur tout en citant ces besoins et avantages.

1 Idée du projet

Le but de ce projet est de créer un compilateur simple pour le peuple tunisien en appliquant les différentes methodes étudiées et utilisées tel que les différentes étapes d'analyse, les tables de symboles, les mots clés personnalisés.

2 Qu'est ce qu'un compilateur

Les compilateurs sont utilisés pour traduire le code source écrit dans un langage de programmation (tel que C, C++, Java, Python, etc.) en un code exécutable compréhensible par la machine (par exemple, le langage de la machine ou le bytecode pour une machine virtuelle). Les programmes écrits en langage de programmation sont généralement constitués de codes source compréhensiblescompréhensibles par les programmeurs, mais qui doivent être traduits en un format compréhensible par la machine avant de pouvoir être exécutés.

L'utilisation d'un compilateur permet donc de transformer le code source d'un programme en code binaire qui peut être compris et exécuté par l'ordinateur. Cela permet aux développeurs de créer des programmes informatiques plus facilement et rapidement, sans avoir à s'inquiéter de détails de bas niveau tel que les registres de la CPU ou la gestion de la mémoire.

3 Les besoins des compilateurs

Le compilateur est un outil qui permet de convertir un code source écrit dans un langage de programmation de haut niveau en un code binaire exécutable par une machine. Les besoins de compilateur sont multiples, notamment :

- **Optimisation** : Le compilateur peut optimiser le code source pour le rendre plus rapide et plus efficace, en utilisant des techniques telles que la réorganisation du code, la suppression de code inutile et la simplification d'expressions.

- **Portabilité** : Le compilateur permet de rendre le code source portable, c'est-à-dire qu'il peut être exécuté sur différentes plateformes matérielles et logicielles. Pour cela, le compilateur doit générer du code binaire qui soit compatible avec les différents systèmes d'exploitation et architectures de processeurs.
- **Détection d'erreurs** : Le compilateur vérifie la syntaxe du code source et signale les erreurs éventuelles. Cela permet de détecter et de corriger les erreurs avant l'exécution du programme.
- **Traduction de langages de programmation** : Le compilateur permet de traduire un langage de programmation en un autre. Par exemple, un compilateur peut convertir du code source écrit en langage C en code binaire exécutable par une machine.

CHAPITRE 2

ENVIRONEMENT LOGICIEL

Introduction

Au cours de cette section, nous présentons les outils et logiciels utilisés tout au long de notre projet.

1 Python

Python est un langage de programmation interprété, ce qui signifie que le code est exécuté directement par un interpréteur, plutôt que d'être compilé en langage machine comme certains autres langages de programmation. Cela permet une approche plus interactive et flexible pour développer des logiciels. Python est également un langage multi-paradigme, ce qui signifie qu'il prend en charge plusieurs styles de programmation, notamment la programmation orientée objet, la programmation fonctionnelle et la programmation impérative.

La syntaxe de Python est conçue pour être simple, lisible et expressive, ce qui signifie qu'il est relativement facile à apprendre pour les débutants en programmation. La plupart des programmes Python peuvent être écrits avec moins de code que d'autres langages, ce qui permet également de développer des applications plus rapidement. Python est également connu pour sa grande bibliothèque standard, qui contient de nombreuses fonctions et modules pour une grande variété de tâches, allant de l'analyse de données à la création d'interfaces utilisateurs graphiques.

Enfin, python est un langage open source, ce qui signifie que son code source est disponible gratuitement et peut-être utilisé, modifié et distribué par quiconque. Il est largement utilisé dans l'industrie, l'enseignement, la recherche scientifique et de nombreux autres domaines

Figure 2.1 : Python

2 Git

Git est un système de contrôle de versions distribué open source qui permet de gérer efficacement les modifications apportées à des fichiers, des dossiers et des projets. Git permet de suivre les modifications apportées aux fichiers au fil du temps, de récupérer des versions antérieures d'un fichier, de collaborer avec d'autres développeurs sur un projet et de gérer les conflits de fusion lorsqu'il y a des modifications concurrentes. Git est devenu un outil de gestion de version très populaire dans l'industrie du développement logiciel et est largement utilisé pour le développement collaboratif de logiciels.

FIGURE 2.2 : Git

3 Visual Studio Code

Visual Studio Code (ou VS Code) est un éditeur de code source léger et gratuit, développé par Microsoft. Il est conçu pour fonctionner sur toutes les principales plates-formes, y compris Windows, macos et Linux. VS code est conçue pour être hautement configurable et extensible, avec une grande variété d'extensions disponibles pour ajouter de nouvelles fonctionnalités ou améliorer le flux de travail de développement.

VS code est équipé de fonctionnalités telles que la coloration syntaxique, l'indentation automatique, la complétion de code, le débogage, la gestion de versions, l'intégration avec les outils de développement de la ligne de commande, et bien plus encore. Il est également conçu pour être facile à utiliser, avec une interface utilisateur intuitive et personnalisable.

En tant qu'éditrice de code source, VS code est souvent utilisé pour le développement de logiciels, notamment pour des langages de programmation tels que Javascript, Python, Java, PHP, Ruby et bien plus encore. Grâce à son extensibilité, VS code est également utilisée pour d'autres

tâches telles que l'écriture de scripts, la configuration système et la gestion de projets.

FIGURE 2.3 : Visual Studio Code

CHAPITRE 3

INSTALLATION

Introduction

Au cours de ce chapitre, on va documenter les étapes d'installation du language sur les différentes plateformes.

1 Étapes

- **Installation Python** :

FIGURE 3.1 : Installation Python

- **Installation Git** :

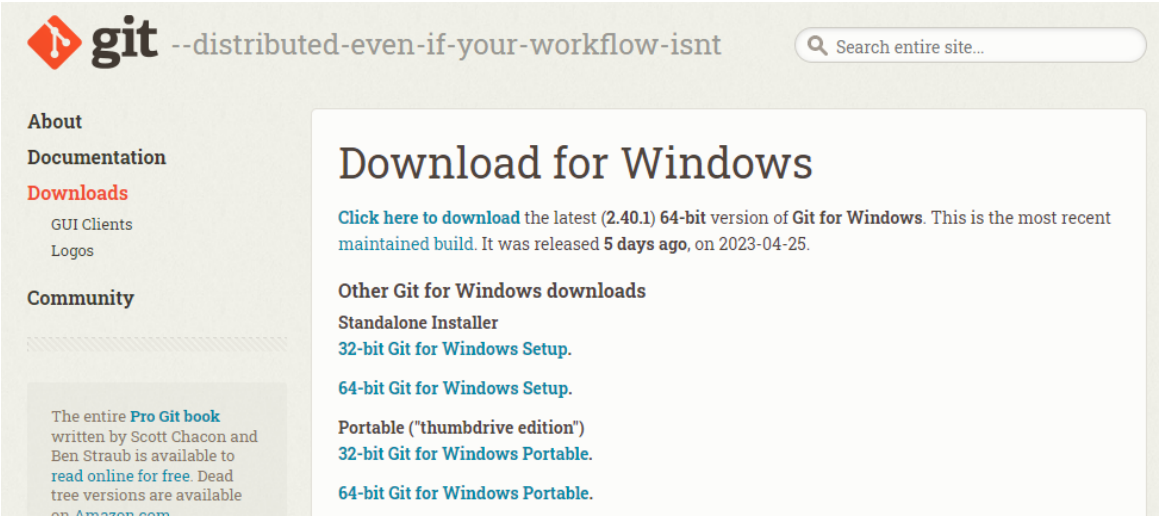

FIGURE 3.2 : Installation Git

- **Installation du dépôt sur GitHub** :

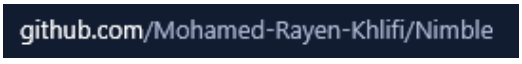

FIGURE 3.3 : Installation Dépôt Nimble sur Github 1

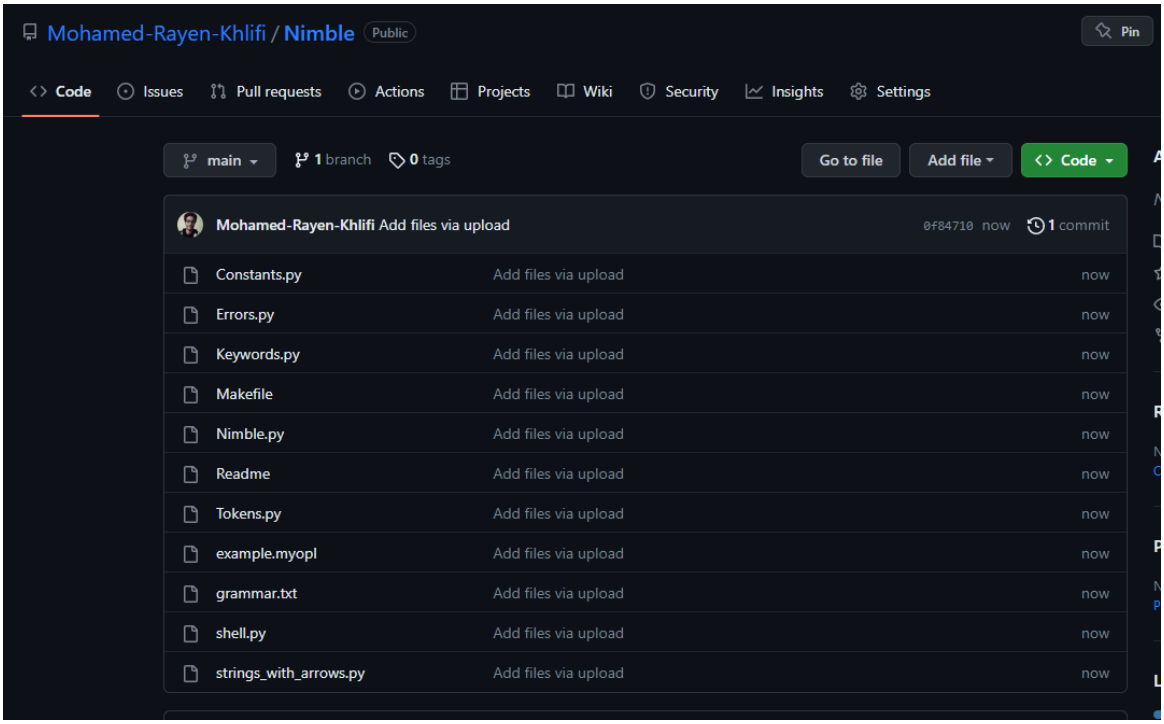

FIGURE 3.4 : Installation Dépôt Nimble sur Github 2

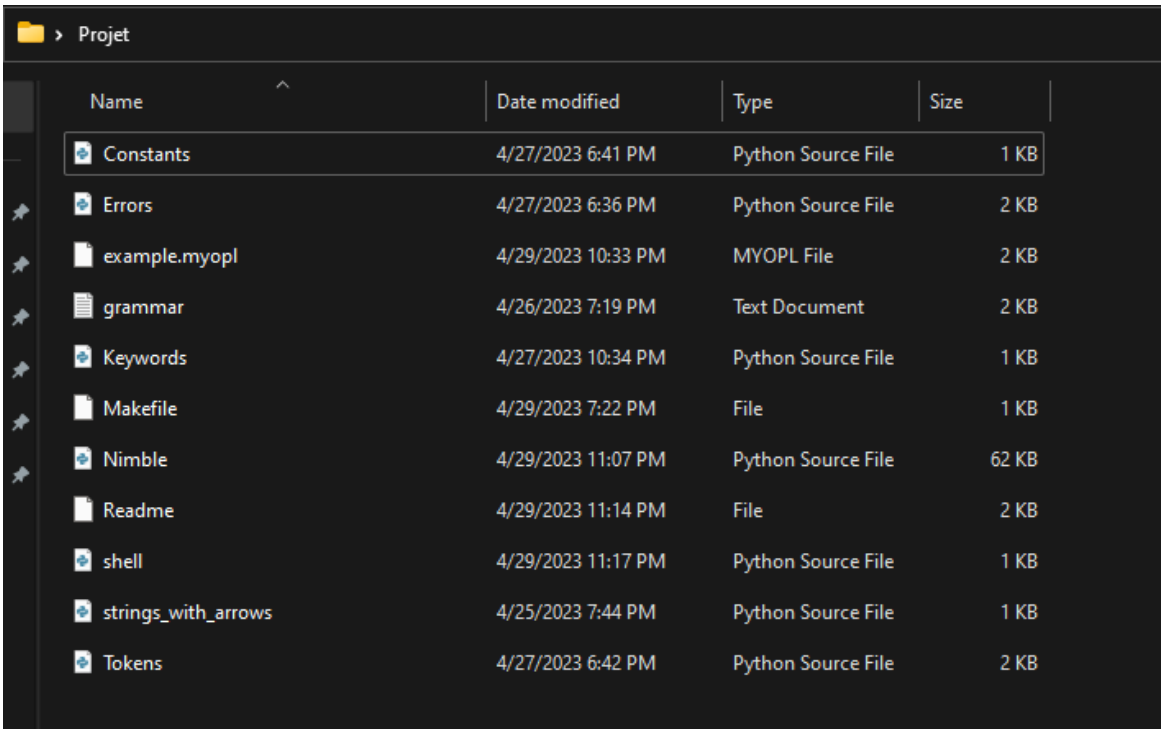

FIGURE 3.5 : Installation Dépôt Nimble sur Github 3

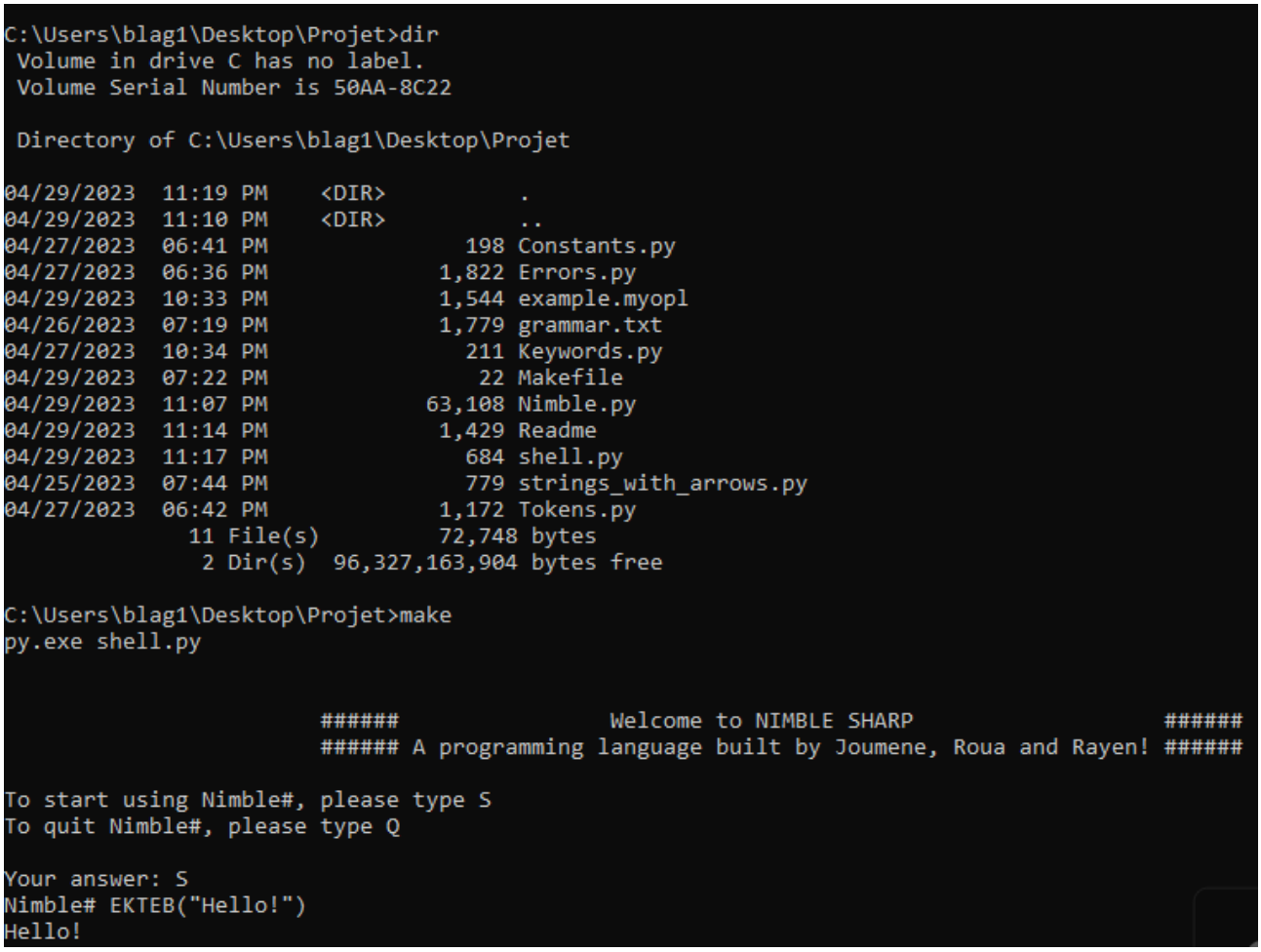

FIGURE 3.6 : Compilation du projet

CHAPITRE 4

ARCHITECTURE

Introduction

Dans cette section, nous allons nous intéresser aux fonctionnalités pour réaliser notre projet.

1 Analyseur Lexicale

L'analyse lexicale est la première étape de l'analyse du code, qui consiste à transformer une chaîne de caractères en une liste d'unités lexicales avec des types et des valeurs associées. Cette étape est essentielle pour que l'analyse syntaxique puisse comprendre la structure du code et générer un arbre syntaxique.

Voici les étapes de l'analyseur lexical que nous avons implémenté dans notre projet ;

- **Lecture du code source :** La première étape consiste à lire le code source en entrée, qui peut être un fichier ou une chaîne de caractères. Le code est lu caractère par caractère pour identifier les différentes unités lexicales.
  voici un exemple de l'automate :

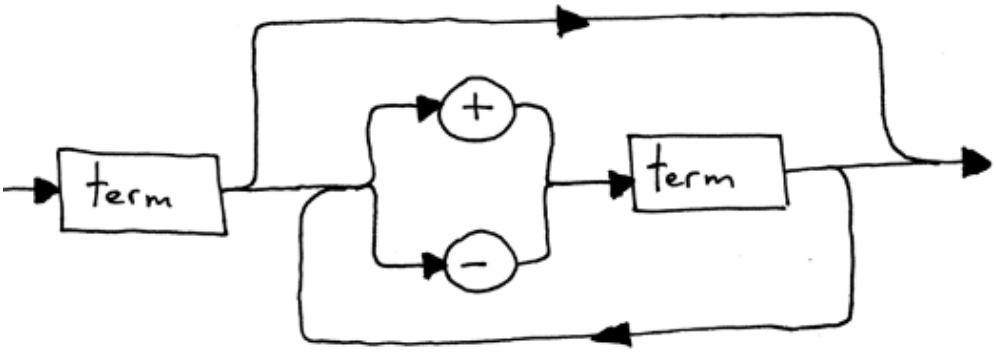

FIGURE 4.1 : Automate

- **Division en unités lexicales :** Chaque fois qu'un caractère est lu, le lexer essaie de trouver la plus longue séquence de caractères qui correspond à une unité lexicale valide. Par exemple, lors de la lecture de "KEN", le lexer générera un token du type "IF".

- **Attribution de types de token :** une fois que chaque unité lexicale a été identifiée, elle est associée à un type de token. Les types de token courants incluent "IDENTIFIER", "KEYWORD", "PLUS", "STRING", etc.
  Cette figure illustre un exemple de correspondance entre les lexèmes et les tokens :

Token	Sample lexemes
INTEGER	342, 9, 0, 17, 1
PLUS	+
MINUS	—

FIGURE 4.2 : Table Tokens

- **Attribution de valeurs :** Chaque token peut également avoir une valeur associée qui représente sa signification dans le programme. Par exemple, un token du type "INT" peut avoir une valeur de 42.
  Voici un exemple où chaque caractère est lu un par un pour identifier les différentes unités lexicales. Dans ce cas, le lexer a identifié que la séquence de caractères "+" correspondait à une unité lexicale valide, qui a été associée à un type de token (PLUS) et à une valeur ("+").

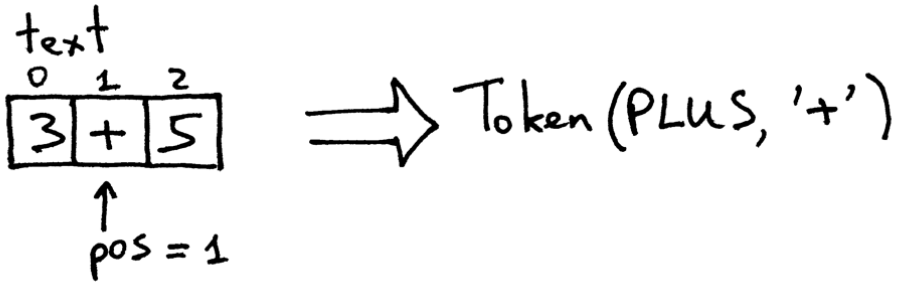

FIGURE 4.3 : Tokénisation des lexemes

- **Gestion des erreurs :** Si le lexer rencontre un caractère ou une séquence de caractères qui ne correspond à aucun token valide, il doit signaler une erreur lexicale.
- **Création d'une liste de tokens :** Une fois que tous les tokens ont été générés et associés à leurs types et valeurs respectifs, ils sont stockés dans une liste pour être utilisés dans l'analyse syntaxique ultérieure.

2 Analyseur Syntaxique

Notre analyseur syntaxique(parser) est responsable de la création d'un arbre syntaxique abstraite à partie des tokens créé par l'analyseur lexical.

FIGURE 4.4 : Analyseur Syntaxique(Parser)

Une notation largement utilisée pour spécifier la syntaxe d'un langage de programmation s'appelle les grammaires sans contexte (ou simplement les grammaires).

Une grammaire est constituée d'une séquence de règles, également appelées productions. elle définit un langage en expliquant quelles phrases elle peut former. Si la grammaire ne peut pas dériver une certaine expression arithmétique, alors elle ne la supporte pas et l'analyseur syntaxique générera une erreur de syntaxe lorsqu'il essaiera de reconnaître l'expression.

La figure ci-dessous représente la structure de base de notre grammaire.

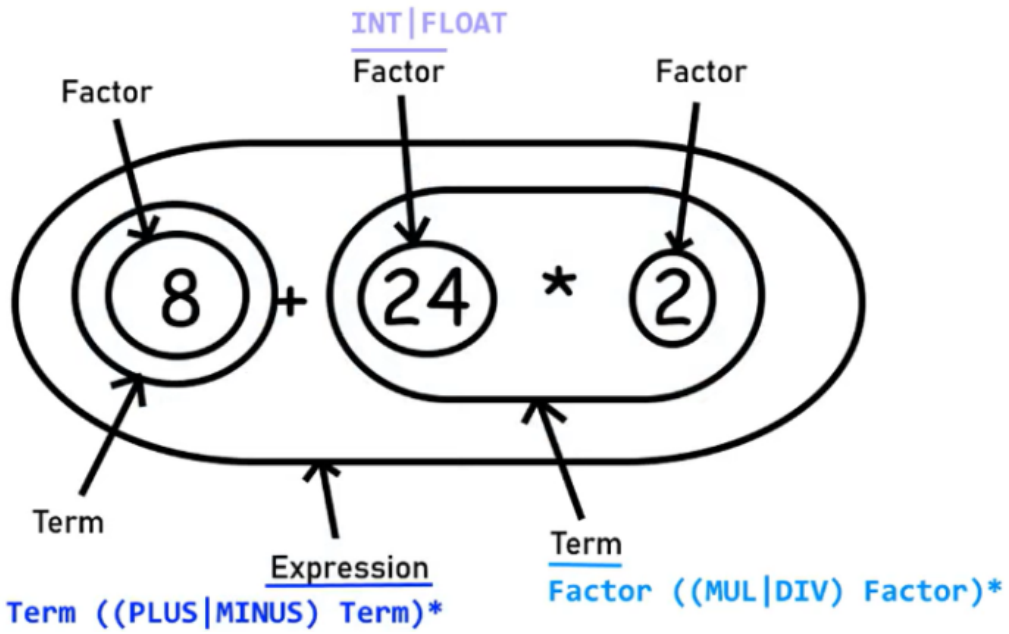

FIGURE 4.5 : Grammaire

- Chaque règle de la grammaire possède une méthode dans l'analyseur syntaxique, donc 3 méthodes : expr(), term(), factor().

- Nous avons créé une classe "Operation" dont les paramètres sont "left", "op" et "right", où "left" et "right" pointent respectivement vers le nœud de l'opérande gauche et le nœud de l'opérande droit. "Op" contient un jeton pour l'opérateur lui-même : Token(PLUS, '+') pour l'opérateur d'addition, Token(MINUS, '-') pour l'opérateur de soustraction, et ainsi de suite.
- Le parser prend un objet Lexer comme argument lorsqu'il est créé.
- Le parser à un attribut current token qui est initialement défini sur le premier jeton renvoyé par le Lexer.
- Le parser à une méthode parse() qui renvoie l'arbre de syntaxe abstraite (AST) pour l'expression d'entrée.
- La méthode parse() appelle la méthode expr() pour analyser l'expression d'entrée.
- La méthode expr() appelle la méthode term() pour analyser le premier terme dans l'expression d'entrée.
- La méthode term() appelle la méthode factor() pour analyser le premier facteur dans l'expression d'entrée.
- La méthode factor() renvoie soit un objet Num représentant un entier, soit appelle récursivement la méthode expr() si elle rencontre une parenthèse gauche.
- Les deux méthodes term() et expr() utilisent une boucle while pour vérifier à plusieurs reprises si le jeton actuel est un opérateur, et si c'est le cas, elles créent un objet "Operation" représentant cette opération avec les expressions factorielles de gauche et de droite étant les expressions factorielles de chaque côté de l'opérateur.
- La méthode parse() renvoie l'AST résultant.

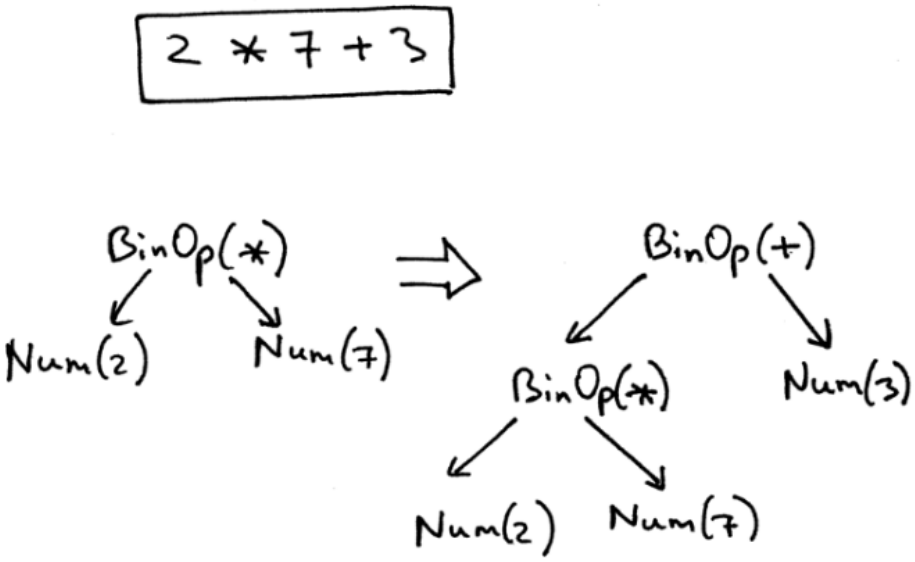

FIGURE 4.6 : Example d'AST

Dans la figure ci-dessus, l'analyseur lit le premier terme « 2x7 » et génère une classe d'opération binaire « C1 » composée de :

;Partie gauche : 2 Opération : multiplication Partie droite : 7 Ensuite, il lit le Token « + » et détermine qu'il s'agit d'une opération, créant ainsi une autre classe d'opération binaire « C2 » composée de : Partie gauche : C1 Opération : addition Partie droite : 3

Voici un exemple de résultats obtenus par notre compilateur

```
Nimble# 5 + 5
(INT:5, PLUS, INT:5)
Nimble# 5 * 5
(INT:5, MUL, INT:5)
Nimble# 5 - 5
(INT:5, MINUS, INT:5)
Nimble# 5 / 5
(INT:5, DIV, INT:5)
```

FIGURE 4.7 : Example Tokens

3 Analyseur Sémantique

Lorsque l'analyseur a fini de construire l'AST, nous savons que le programme est grammaticalement correct ; c'est-à-dire que sa syntaxe est correcte selon nos règles de grammaire et maintenant nous pouvons nous concentrer séparément sur la vérification des erreurs qui nécessitent un contexte et des informations supplémentaires que l'analyseur n'avait pas au moment de la construction de l'AST.

Fondamentalement, un analyseur sémantique est un processus pour nous aider à déterminer si un programme a un sens, et s'il a un sens, selon une définition du langage. C'est une autre étape après l'analyse de notre programme et la création d'un AST pour vérifier le programme source pour certaines erreurs supplémentaires que l'analyseur n'a pas pu détecter en raison d'un manque d'information supplémentaires (contexte).

Sur la figure ci-dessus ; structure de l'interpréteur Nimble, vous pouvez voir que notre lexer obtiendra le code source en entrée, le transformera en tokens que l'analyseur consommera et utilisera pour vérifier que le programme est grammaticalement correct, puis il générera un arbre de syntaxe abstraite que notre nouveau phase d'analyse sémantique utilisera pour appliquer les différentes exigences du langage Nimble. Lors de la phase d'analyse sémantique, l'analyseur sémantique construira et utilisera également la table des symboles. Après l'analyse sémantique, notre interprète prendra l'AST, évaluera le programme en parcourant l'AST et produira le résultat du programme

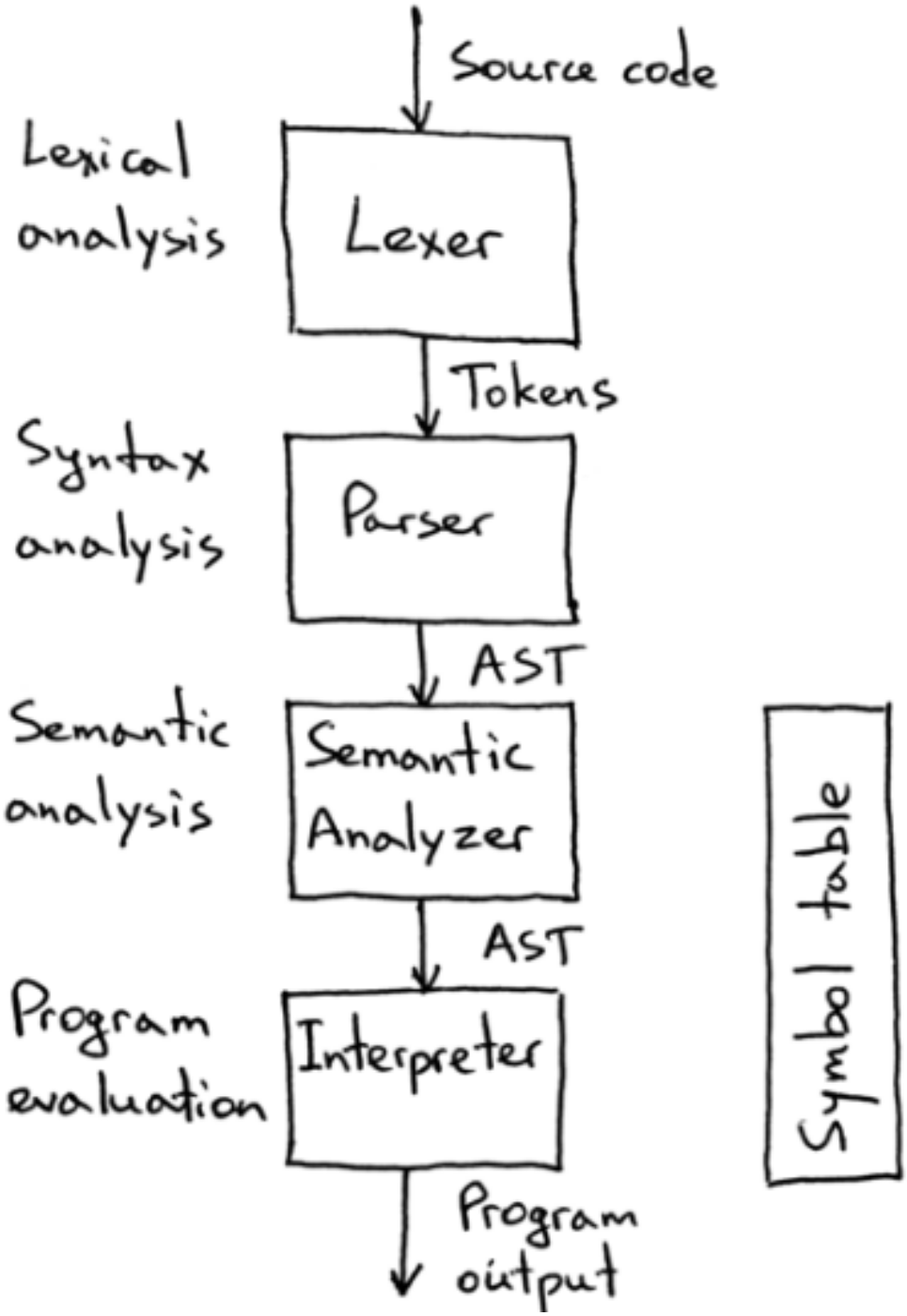

FIGURE 4.8 : Structure de l'interpréteur Nimble

.

4 Table de symbole

Une table de symboles est une structure de données utilisée par un compilateur pour stocker des informations sur des symboles comme :

- Le nom du symbole
- Le type de données associées
- L'emplacement de stockage
- La portée
- Les propriétés du symbole, telles que sa valeur par défaut ou sa visibilité.

Ces informations sont utilisées par le compilateur pour générer du code objet à partir du code source. La table des symboles est également utilisée pour détecter les erreurs telles que les identificateurs non déclarés, les identificateurs en double, les types de données incompatibles, etc. Elle permet aussi d'obtenir les valeurs des variables ou de les changer ainsi que les supprimer.

4.1 Table de symbole globale

La table de symbole global est similaire à la table de symbole mais celle-ci est utilisée pour gérer les variables, structure de données et les fonctions intégrées du language tels que ;

- La fonction PRINT(EKTEB)
- La variable PI
- La fonction INPUT(AkRA)

Ces attributs sont prédéfinis, par suite ils sont accessibles directement sans devoir les implementers.

CHAPITRE 5

FONCTIONNALITÉS

Introduction

Pour ce dernier chapitre, on va proposer des Exemple de gestion d'erreurs d'éxecution pour les fonctionnalités de Nimble.

1 Mot clés

VAR	SAJEL
IF	KEN
ELIF	WELLAKEN
ELSE	MAKENECH
END	KAHAW
FOR	MEN
THEN	AAMEL
FUN	APP
WHILE	MADEM
RETURN	RAJAA
CONTINUE	KAMEL
BREAK	KOS
PRINT	EKTEB
INPUT	AKRA
INPUT_INT	AKRAADAD
NOT	AAKS
OR	WELLA
AND	W
STEP	KADEM
TO	HATA
IS_FUN	FONCTION
RUN	LANCI

2 Operateurs

2.1 Arithmétiques

```
Nimble# 1+2
3
Nimble# 3-5
-2
Nimble# 3*5
15
Nimble# 16/4
4.0
Nimble# 2^2
4
```

FIGURE 5.1 : Exemple opérations arithmétiques

2.2 Logiques

```
Nimble# 1 W 0
0
Nimble# 1 WELLA 0
1
Nimble# AAKS 0
1
Nimble# AAKS 1
0
```

FIGURE 5.2 : Exemple opérations logiques

3 Variables

```
Nimble# SAJEL my_name="Ahmed"
"Ahmed"
Nimble# SAJEL my_age=23
23
Nimble# EKTEB(my_name)
Ahmed
0
Nimble# EKTEB(my_age)
23
0
```

FIGURE 5.3 : Exemple declaration des variables

4 If statement

```
Nimble# KEN 23<100 AAMEL EKTEB("Mezlet Sghir") MAKENECH EKTEB("KBERT")
Mezlet Sghir
```

FIGURE 5.4 : Exemple d'une structure conditionnelle

5 For

```
Nimble# MEN i=0 HATA 10 KADEM 2 AAMEL EKTEB(i)
0
2
4
6
8
```

FIGURE 5.5 : Exemple d'une boucle for

6 While

```
Nimble# MADEM SHIH AAMEL EKTEB("Infinite Loop")
```

FIGURE 5.6 : Exemple d'une boucle while 1

```
Infinite Loop
Infinite Loop
Infinite Loop
Infinite Loop
Infinite Loop
Infinite Loop
Infinite Loop
Infinite Loop
Infinite Loop
Infinite Loop
Infinite Loop
Infinite Loop
```

FIGURE 5.7 : Exemple d'une boucle while 2

7 Structure de données : List

```
EKTEB("##############List Operations ##############")
SAJEL liste1 = [1,2]
SAJEL liste2 = [4,5]
EKTEB("Liste1:")
EKTEB(liste1)
EKTEB("Liste2:")
EKTEB(liste2)
liste1 + 3
EKTEB("Liste1 with an added 3:")
EKTEB(liste1)
EKTEB("Liste1 concat Liste2:")
EKTEB(liste1*liste2)
EKTEB("First element of Liste1:")
EKTEB(liste1/0)
EKTEB("Last element of liste1:")
EKTEB(liste1/(TOUL(liste1)-1))
EKTEB("Liste1 without the first element:")
EKTEB(liste1-0)
EKTEB("The length of Liste1 is:")
EKTEB(TOUL(liste1))
```

FIGURE 5.8 : Exemple d'opérations sur les listes 1

```
##############List Operations ##############
Liste1:
1, 2
Liste2:
4, 5
Liste1 with an added 3:
1, 2, 3
Liste1 concat Liste2:
1, 2, 3, 4, 5
First element of Liste1:
1
Last element of liste1:
5
Liste1 without the first element:
2, 3, 4, 5
The length of Liste1 is:
4
```

FIGURE 5.9 : Exemple d'opérations sur les listes 2

8 Fonctions

```
EKTEB("#############Function Operations #############")
APP somme(x,y,z) -> x+y+z

SAJEL res = somme(1,0,10)
EKTEB(res)
```

FIGURE 5.10 : Exemple d'une arrow function 1

```
##############Function Operations ##############
11
```

FIGURE 5.11 : Exemple d'une arrow function 2

9 Return

```
EKTEB("#############Function Operations #############")
APP somme(x,y,z)
    RAJAA x+y+z
KAHAW

SAJEL res = somme(10,12,10)
EKTEB(res)
```

FIGURE 5.12 : Exemple du mot-clé Return 1

```
##############Function Operations ##############
32
```

FIGURE 5.13 : Exemple du mot-clé Return 2

10 Break

```
EKTEB("#############Function Operations #############")
APP somme(x,y,z)
    SAJEL zero = (x==0) WELLA (y==0)
    KEN zero WELLA z==0 AAMEL
        EKTEB("Il ya un zero")
        KOS
    KAHAW
    RAJAA x+y+z
KAHAW

SAJEL res = somme(1,0,10)
EKTEB(res)
```

FIGURE 5.14 : Exemple du mot-clé Break 1

```
##############Function Operations #############
Il ya un zero
```

FIGURE 5.15 : Exemple du mot-clé Break 2

11 Gestion d'erreurs

```
Nimble# 5 +
Invalid Syntax: Expected int, float, identifier, '+', '-', '(', '[', KEN', 'MEN', 'MADEM', 'APP'
File <stdin>, line 1

5 +
  ^
```

FIGURE 5.16 : Exemple de gestion d'erreurs 1

```
Nimble# 123 123 +
Invalid Syntax: Token cannot appear after previous tokens
File <stdin>, line 1

123 123 +
    ^^^
```

FIGURE 5.17 : Exemple de gestion d'erreurs 2

```
Nimble# 10/(6-6)
Traceback (most recent call last):
  File <stdin>, line 1, in <program>
Runtime Error: Division by zero

10/(6-6)
    ^^^
```

FIGURE 5.18 : Exemple de gestion d'erreurs 3

```
Nimble# APP add(a,b
Invalid Syntax: Expected ',' or ')'
File <stdin>, line 1

APP add(a,b
          ^
```

FIGURE 5.19 : Exemple de gestion d'erreurs 4

```
Nimble# APP add(a,b) -> a+b
<function add>
Nimble# APP add(a,b
Invalid Syntax: Expected ',' or ')'
File <stdin>, line 1

APP add(a,b
          ^
Nimble# EKTEB(add(1,2,3))
Traceback (most recent call last):
  File <stdin>, line 1, in <program>
Runtime Error: 1 too many args passed into <function add>

EKTEB(add(1,2,3))
      ^^^^^^^^^
```

FIGURE 5.20 : Exemple de gestion d'erreurs 5

```
Nimble# 2*d
Traceback (most recent call last):
  File <stdin>, line 1, in <program>
Runtime Error: 'd' is not defined

2*d
  ^
```

FIGURE 5.21 : Exemple de gestion d'erreurs 6

```
Nimble# MEN i = 0 HATA 10
Invalid Syntax: Expected 'AAMEL'
File <stdin>, line 1

MEN i = 0 HATA 10
                ^
```

FIGURE 5.22 : Exemple de gestion d'erreurs 7

```
Nimble# KEN SHIH
Invalid Syntax: Expected 'AAMEL'
File <stdin>, line 1

KEN SHIH
        ^
```

FIGURE 5.23 : Exemple de gestion d'erreurs 8

```
Nimble# MADEM GHALET
Invalid Syntax: Expected 'AAMEL'
File <stdin>, line 1

MADEM GHALET
            ^
```

FIGURE 5.24 : Exemple de gestion d'erreurs 9

```
Nimble# LANCI("fichier_inexistant")
Traceback (most recent call last):
  File <stdin>, line 1, in <program>
  File <stdin>, line 1, in run
Runtime Error: Failed to load script "fichier_inexistant"
[Errno 2] No such file or directory: 'fichier_inexistant'

LANCI("fichier_inexistant")
^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

FIGURE 5.25 : Exemple de gestion d'erreurs 10

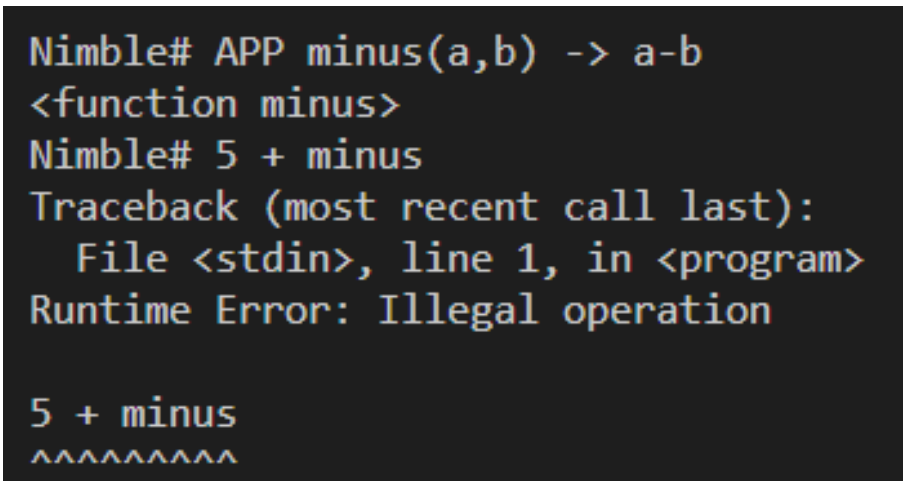

FIGURE 5.26 : Exemple de gestion d'erreurs 11

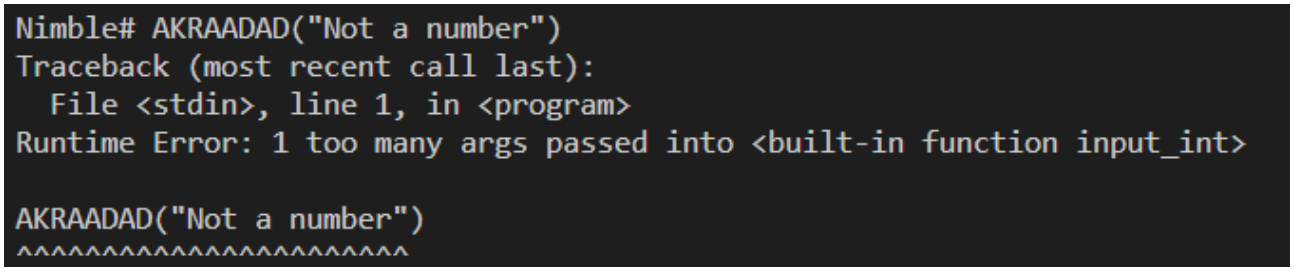

FIGURE 5.27 : Exemple de gestion d'erreurs 12

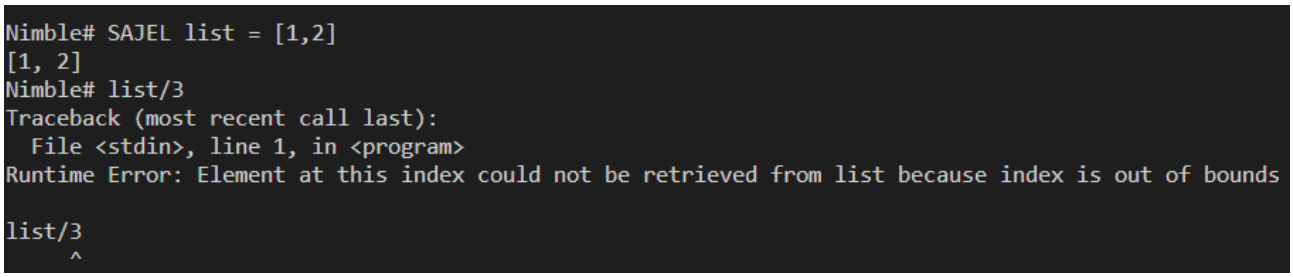

FIGURE 5.28 : Exemple de gestion d'erreurs 13

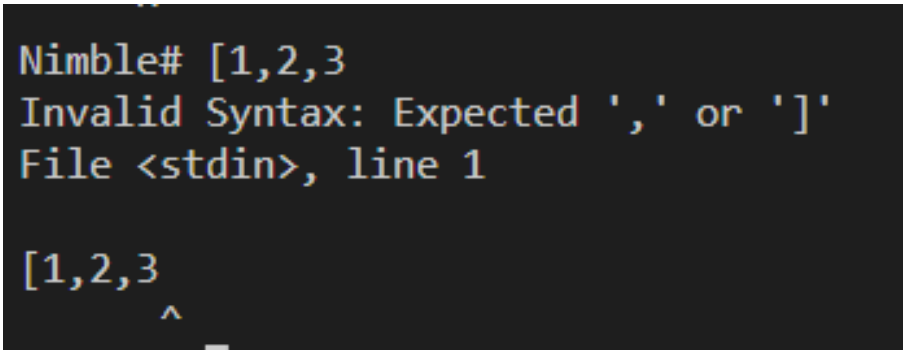

FIGURE 5.29 : Exemple de gestion d'erreurs 14

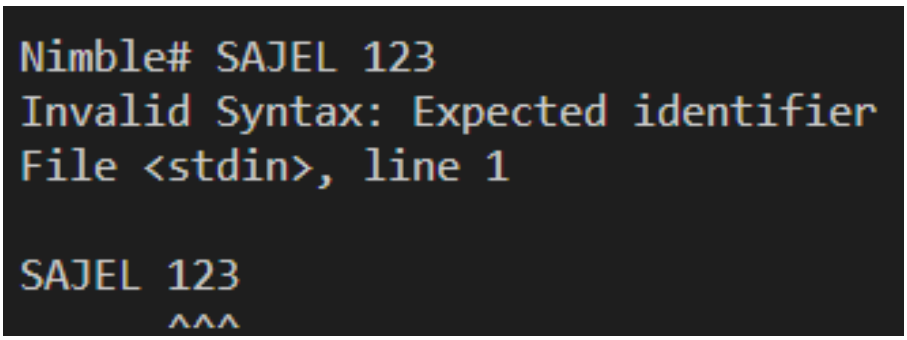

FIGURE 5.30 : Exemple de gestion d'erreurs 15

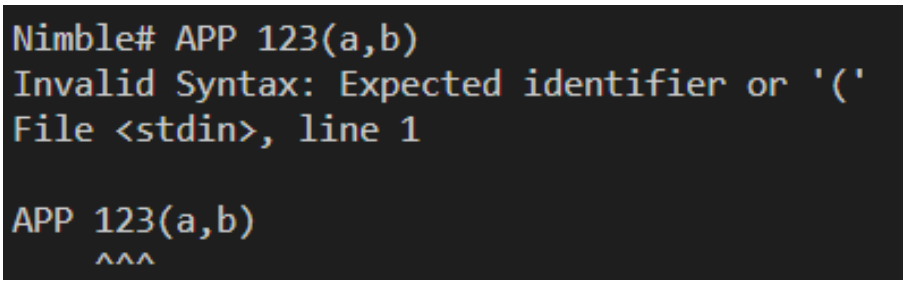

FIGURE 5.31 : Exemple de gestion d'erreurs 16

CONCLUSION GÉNÉRALE

En conclusion, ce rapport a présenté de manière détaillée le projet de réalisation d'un mini-compilateur Python "Nimble#" . Nous avons souligné l'importance de la réalisation d'un tel projet et présenté les différentes étapes nécessaires à sa conception. Nous avons également décrit l'architecture complète du mini-compilateur, . De plus, nous avons mis en avant les différentes fonctionnalités que nous avons implémentées. Enfin, nous avons inclus des captures d'écran pour illustrer le projet.

Ce projet nous a permis d'acquérir de nouvelles connaissances et compétences dans le domaine de la compilation, ainsi que de mettre en pratique les notions théoriques apprises en cours.