Université Moulay Ismail

Faculté des Sciences et Techniques

Département d'Informatique

LST : Génie Logiciel

Mini-Projet :

Création d'un langage interpréter et compiler en utilisant PLY (Python Lex-Yacc)

Réaliser par :

Othmane Haiba

Amine Hamouchi

Encadrer par :

N. Mouhni

A.U : 2024/2025

I. Introduction :

Rexi est un langage de programmation expérimental qui combine simplicité syntaxique et typage fort. Il est conçu pour être facile à apprendre tout en maintenant une structure claire et des types expressifs, Parmi les caractéristiques de Rexi on a :

- Typage fort et statique
- Syntaxe claire et intuitive
- Types de données expressifs (IN, IR, STR, BINARY, TAB)
- Support natif des structures de contrôle
- Interpréteur complet écrit en Python

II. Le Compilateur:

Le compilateur Rexi transforme le code source en une suite d'instructions interprétables, Voici une explication détaillée des étapes impliquées :

1.1 Analyse Lexicale :

L'analyse lexicale consiste à identifier les composants syntaxiques appelés tokens dans le code source.
Elle utilise des expressions régulières pour reconnaître les éléments comme les opérateurs, les mots-clés, et les identifiants.

Voici un extrait du code :

```
tokens = (
    # Keywords
    'IF', 'THEN', 'ELSE', 'END', 'WHILE', 'FOR', 'FUNCTION', 'RETURN', 'OUTPUT',
    # Types
    'TYPE',
    # Operators
    'PLUS', 'MINUS', 'MULTIPLY', 'DIVIDE', 'ASSIGN',
    'GT', 'LT', 'GTE', 'LTE', 'EQUALS', 'NOTEQUALS',
    # Delimiters
    'LPAREN', 'RPAREN', 'LBRACE', 'RBRACE', 'LBRACKET', 'RBRACKET', 'SEMICOLON', 'COMMA',
    # Values
    'NUMBER', 'STRING', 'BOOLEAN', 'ID'
)
```

1.2 Analyse Syntaxique :

L'analyse syntaxique construit un arbre syntaxique abstrait (AST) qui structure les éléments lexicaux en représentations compréhensibles.

Voici un extrait du code :

```
# --- Règles d'analyse syntaxique complètes --- #

def p_program(p):
    """program : declarations"""
    p[0] = Program(p[1])

def p_declarations(p):
    """declarations : declaration
                    | declarations declaration"""
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = p[1] + [p[2]]

def p_declaration(p):
    """declaration : var_declaration
                   | function_declaration
                   | statement"""
    p[0] = p[1]

def p_var_declaration(p):
    """var_declaration : TYPE ID ASSIGN expression SEMICOLON
                       | TYPE ID LBRACKET NUMBER RBRACKET SEMICOLON"""
    if len(p) == 6:
        p[0] = VarDeclaration(p[1], p[2], p[4])
    else:
        p[0] = ArrayDecl(p[1], p[2], p[4])
```

1.3 Génération de Code :

La génération de code traduit l'AST en une suite d'instructions exécutables. Les instructions incluent les affectations, les conditions, et les boucles.

Voici un extrait du code :

```python
class CodeGenerator:
    def __init__(self):
        self.code = []
        self.temp_counter = 0
        self.label_counter = 0

    def generate_temp(self):
        self.temp_counter += 1
        return f"t{self.temp_counter}"

    def generate_label(self):
        self.label_counter += 1
        return f"L{self.label_counter}"

    def emit(self, op, arg1=None, arg2=None, result_=None):
        instruction = (op, arg1, arg2, result_)
        self.code.append(instruction)
        return result_

    def generate_code(self, node):
        method_name = f'generate_{type(node).__name__.lower()}'
        if hasattr(self, method_name):
            return getattr(self, method_name)(node)
        raise Exception(f"No visitor for {type(node).__name__}")

    def generate_program(self, node):
        for decl in node.declarations:
            self.generate_code(decl)
        return self.code
```

2. L'Interpréteur :

L'interpréteur exécute directement le code source Rexi sans passer par une étape intermédiaire de compilation.
Voici les principales étapes :

2.1 Analyse Lexicale et Syntaxique :

Comme pour le compilateur, l'interpréteur commence par une analyse lexicale et syntaxique faite par une Classe Lexer.
Cette Classe identifie les tokens et construit un AST adapté pour une exécution directe.

2.2 Exécution Dynamique :

L'AST est parcouru pour exécuter les instructions dynamiquement. Les structures comme « if », « while », et les opérations mathématiques sont exécutées en mémoire.

2.3 Gestion des Erreurs :

L'interpréteur gère les erreurs de syntaxe et de type, fournissant des retours immédiats en cas de problème.

3. L'Interface Graphique :

L'interface graphique du projet Rexi Language Editor a été conçue pour offrir une expérience utilisateur fluide et intuitive aux utilisateurs souhaitant écrire, éditer et exécuter du code écrit dans le langage Rexi. Cette interface repose sur le module **CustomTkinter**, qui combine l'esthétique moderne et minimaliste avec les capacités établies de Tkinter.

3.1 Explorateur de Fichiers:

Un explorateur de fichiers situé à gauche de l'interface permet aux utilisateurs de naviguer dans un répertoire, de visualiser les fichiers disponibles et d'ouvrir le fichier sélectionné dans l'éditeur de texte principal.

3.2 Éditeur de Texte Principal:

Un champ de texte central permet à l'utilisateur de visualiser et modifier le contenu des fichiers. Cet éditeur est personnalisé pour correspondre au thème visuel de l'application.

Voici un extrait du code :

```
# Main Text Editor
text_editor = ctk.CTkTextbox(app, width=550, height=390, corner_radius=10)
text_editor.pack(pady=10, padx=20, side="top", expand=True, fill="both")

# Console/Output Area
console_label = ctk.CTkLabel(app, text="Console Output")
console_label.pack(anchor="w", padx=20)

console = ctk.CTkTextbox(app, width=600, height=120, corner_radius=10, state="normal")
console.pack(pady=5, padx=20)
```

3.3 Exécution de Code:

Un bouton dédié permet d'exécuter le code présent dans l'éditeur et d'afficher le résultat dans la console.
Voici un extrait du code :

```
# Buttons
button_frame = ctk.CTkFrame(app, corner_radius=10 )
button_frame.pack( fill="x" ,padx=20, pady=5 )

mode_switch = ctk.CTkSwitch(button_frame,text = "", variable=switch_var, onvalue=True, offvalue=False, command=toggle_mode)
mode_switch.pack(side="left", padx=0)
on = ctk.CTkLabel(button_frame, textvariable = dynamic_text)
on.pack(side = "left" , padx=0)
run_button = ctk.CTkButton(button_frame, text="Run Code", command=run_code ,width=20 , height= 20)
run_button.pack(side="right", padx=5)

save_button = ctk.CTkButton(button_frame, text="Save File", command=save_file ,width=20 , height= 20)
save_button.pack(side="right", padx=5)

open_button = ctk.CTkButton(button_frame, text="Open File", command=open_file ,width=20 , height= 20)
open_button.pack(side="right", padx=5)
```

3.4 En-tête avec Logo et Nom du Langage :

Un en-tête est présent en haut de l'application, affichant le logo et le nom du langage "Rexi" pour renforcer l'identité visuelle.

- **Points Techniques :**

  - Affichage d'une image via « Pillow » (ImageTk).

- Utilisation d'un « CTkLabel » personnalisé pour afficher le nom du langage.

Pourquoi CustomTkinter???

- Esthétique Moderne : L'utilisation de CustomTkinter offre une interface propre et attrayante.

- Fonctionnalités Complètes : L'application couvre tous les besoins de base pour un éditeur de langage spécialisé.

- Extensibilité : Les modules sont conçus de manière à permettre des ajouts futurs, comme un débogueur ou un interprète intégré.