1. Introduction

Ce projet consiste en la conception et l'implémentation d'un compilateur pour un sous-ensemble du langage Pascal, appelé Mini-Pascal. Le compilateur effectue les phases traditionnelles de compilation : analyse lexicale, analyse syntaxique, construction de l'arbre syntaxique abstrait (AST), avec une interface utilisateur développée en Streamlit pour une interaction en temps réel.

Objectifs du projet :

- Implémenter un analyseur lexical reconnaissant les tokens du langage
- Développer un analyseur syntaxique LR(0) pour valider la structure des programmes
- Construire un arbre syntaxique abstrait représentant la structure hiérarchique
- Fournir une interface web interactive pour l'analyse de code
- Gérer les erreurs lexicales et syntaxiques avec des messages informatifs

2. Choix de Conception

2.1 Architecture du Compilateur

L'architecture adoptée suit le modèle traditionnel des compilateurs en plusieurs passes :

Code Source → Analyseur Lexical → Tokens → Analyseur Syntaxique → AST → Interface Utilisateur

2.2 Technologies Utilisées

- **Python 3.12**: Langage d'implémentation principal
- **PLY (Python Lex-Yacc)** : Pour l'analyse lexicale et syntaxique
- **Streamlit** : Pour l'interface web interactive
- **Dataclasses** : Pour la représentation des nœuds AST

2.3 Structure des Fichiers

mini-pascal-compiler/

```
├── lexer.py         # Analyseur lexical
├── parser_1.py      # Analyseur syntaxique
├── ast_1.py         # Classes AST
├── compiler.py      # Module central de compilation
├── errors.py        # Gestion des erreurs
├── app.py           # Interface Streamlit
├── semantic.py      # Analyse sémantique (en développement)
├── symbol_table.py  # Table des symboles (en développement)
└── requirements.txt  # Dépendances
```

3. Sous-ensemble Pascal Retenu

3.1 Grammaire EBNF

La grammaire suivante définit le sous-ensemble de Pascal implémenté :

```
Program       ::= "program" Identifier ";" Block "."
Block         ::= [Declarations] "begin" Statements "end"
Declarations  ::= { ConstDecl | VarDecl }
ConstDecl     ::= "const" Identifier "=" Literal ";"
VarDecl       ::= "var" IdentifierList ":" Type ";"
IdentifierList ::= Identifier { "," Identifier }
Type          ::= "integer" | "real" | "boolean"
Statements    ::= Statement { ";" Statement }
Statement     ::= Assignment
                | IfStatement
                | WhileStatement
                | ForStatement
                | RepeatStatement
```

```
            | CompoundStatement
Assignment    ::= Identifier ":=" Expression
IfStatement   ::= "if" Expression "then" Statement ["else" Statement]
WhileStatement ::= "while" Expression "do" Statement
ForStatement  ::= "for" Identifier ":=" Expression Direction Expression "do" Statement
Direction     ::= "to" | "downto"
RepeatStatement ::= "repeat" Statements "until" Expression
CompoundStatement ::= "begin" Statements "end"
Expression    ::= SimpleExpression [ RelOp SimpleExpression ]
SimpleExpression ::= Term { AddOp Term }
Term          ::= Factor { MulOp Factor }
Factor        ::= Identifier
            | Literal
            | "(" Expression ")"
            | UnaryOp Factor
Literal       ::= IntegerLiteral | RealLiteral | BooleanLiteral
RelOp         ::= "=" | "<>" | "<" | "<=" | ">" | ">="
AddOp         ::= "+" | "-" | "or"
MulOp         ::= "*" | "/" | "div" | "mod" | "and"
UnaryOp       ::= "+" | "-" | "not"
```

3.2 Tokens Lexicaux

La liste complète des tokens reconnus :

Catégorie	Tokens

Mots-clés	PROGRAM, VAR, CONST, INTEGER, REAL, BOOLEAN, IF, THEN, ELSE, WHILE, DO, FOR, TO, DOWNTO, REPEAT, UNTIL, BEGIN, END
Opérateurs	DIV, MOD, AND, OR, NOT, PLUS, MINUS, MULT, DIVIDE, ASSIGN, EQUAL, NEQ, LT, GT, LEQ, GEQ
Identificateurs	ID
Constantes	INT_CONST, REAL_CONST, BOOL_CONST
Séparateurs	LPAREN, RPAREN, SEMI, COLON, COMMA, DOT

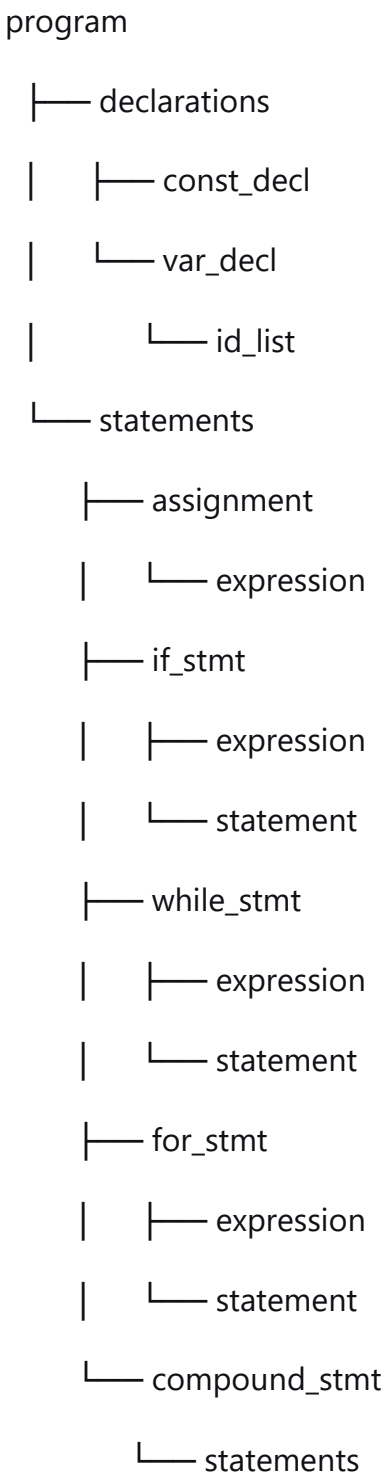
3.3 Tableau des Productions

Les productions principales du parseur :

Numéro	Production
1	program → PROGRAM ID SEMI block DOT
2	block → declarations BEGIN statements END
3	declarations → declarations var_decl
4	declarations → declarations const_decl

5	declarations → ε
6	var_decl → VAR id_list COLON type SEMI
7	const_decl → CONST ID EQUAL literal SEMI
8	id_list → ID
9	id_list → id_list COMMA ID
10	type → INTEGER | REAL | BOOLEAN
11	statements → statement
12	statements → statements SEMI statement
13	statement → assignment | if_stmt | while_stmt | for_stmt | repeat_stmt | compound_stmt

3.4 Arbre des Dépendances



4. Analyse Lexicale

4.1 Automate Fini Déterministe

L'analyseur lexical est implémenté sous forme d'automate fini déterministe généré par PLY à partir des expressions régulières suivantes :

LETTRE → [a-zA-Z_]

CHIFFRE → [0-9]

ID → LETTRE (LETTRE | CHIFFRE)*

ENTIER → CHIFFRE+

REEL → CHIFFRE+ '.' CHIFFRE*

ASSIGN → ':'

EGAL → '='

NEQ → '<>'

LT → '<'

LEQ → '<='

GT → '>'

GEQ → '>='

PLUS → '+'

MINUS → '-'

MULT → '*'

DIVIDE → '/'

LPAREN → '('

RPAREN → ')'

SEMI → ';'

COMMA → ','

DOT → '.'

COMMENT → '{' [^}]* '}'

ESPACE → [\t\n]+ (ignoré)

4.2 Code de l'Analyseur Lexical

Extrait de lexer.py

```
tokens = (
    'PROGRAM', 'VAR', 'CONST', 'INTEGER', 'REAL', 'BOOLEAN',
    'IF', 'THEN', 'ELSE', 'WHILE', 'DO', 'FOR', 'TO', 'DOWNTO',
    'REPEAT', 'UNTIL', 'DIV', 'MOD', 'AND', 'OR', 'NOT',
    'ID', 'INT_CONST', 'REAL_CONST', 'BOOL_CONST',
    'PLUS', 'MINUS', 'MULT', 'DIVIDE', 'ASSIGN',
    'EQUAL', 'NEQ', 'LT', 'GT', 'LEQ', 'GEQ',
    'LPAREN', 'RPAREN', 'SEMI', 'COLON', 'COMMA', 'DOT',
    'BEGIN', 'END'
)

reserved = {
    'program': 'PROGRAM',
    'var': 'VAR',
    'const': 'CONST',
    'integer': 'INTEGER',
    'real': 'REAL',
    'boolean': 'BOOLEAN',
    'if': 'IF',
    'then': 'THEN',
    'else': 'ELSE',
    'while': 'WHILE',
    'do': 'DO',
    'for': 'FOR',
    'to': 'TO',
    'downto': 'DOWNTO',
    'repeat': 'REPEAT',
    'until': 'UNTIL',
    'div': 'DIV',
    'mod': 'MOD',
    'and': 'AND',
```

```
    'or': 'OR',
    'not': 'NOT',
    'begin': 'BEGIN',
    'end': 'END',
    'true': 'BOOL_CONST',
    'false': 'BOOL_CONST'
}

def t_ID(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    t.type = reserved.get(t.value.lower(), 'ID')
    if t.type == 'BOOL_CONST':
        t.value = (t.value.lower() == 'true')
    return t
```

4.3 Tests Lexicaux

Exemple valide :

```
program example;
const MAX = 10;
var x, y : integer;
begin
    x := 5;
    y := x + MAX;
end.
```

Exemple invalide :

```
program test;
var @x : integer;  # Caractère '@' non reconnu
begin
```

```
    x = 5;        # Opérateur '=' au lieu de ':='
end.
```

5. Analyse Syntaxique

5.1 Choix de la Méthode : LR(0)

Le compilateur utilise l'analyse syntaxique **LR(0)** implémentée avec PLY (LALR parser generator). Ce choix est justifié par :

Avantages de LR(0) :

1. **Puissance expressive** : Capable de reconnaître une large classe de grammaires
2. **Détection rapide des erreurs** : Détecte les erreurs syntaxiques dès que possible
3. **Construction d'AST naturelle** : La structure ascendante facilite la construction d'arbres
4. **Compatible avec PLY** : Outil mature et bien documenté pour Python

5.2 Préférence LR(0) vs LL(k)

Critère	LR(0)	LL(k)
Expressivité	Plus large	Plus limitée
Détection erreurs	Plus précoce	Moins précoce
Construction AST	Ascendante (naturelle)	Descendante (manuelle)
Complexité	Automatique (PLY)	Manuelle (recursive descent)
Maintien	Plus facile	Plus difficile

5.3 Table d'Analyse LR(0)

La table d'analyse est générée automatiquement par PLY et contient :

- **Actions** : Décalage (shift), Réduction (reduce), Acceptation (accept), Erreur
- **États** : 45 états générés pour notre grammaire
- **Conflits** : 0 conflits shift/reduce, 0 conflits reduce/reduce

5.4 Code de l'Analyseur Syntaxique

```
# Extrait de parser_1.py

precedence = (

   ('left', 'OR'),

   ('left', 'AND'),

   ('right', 'NOT'),

   ('nonassoc', 'EQUAL', 'NEQ', 'LT', 'LEQ', 'GT', 'GEQ'),

   ('left', 'PLUS', 'MINUS'),

   ('left', 'MULT', 'DIVIDE', 'DIV', 'MOD'),

   ('right', 'UMINUS'),

)


def p_program(p):

   '''program : PROGRAM ID SEMI block DOT'''

   p[0] = Program(name=p[2], block=p[4], lineno=p.lineno(1), col=find_column(source_text,
p.slice[1]))
```

6. Construction de l'AST

6.1 Structure de l'AST

L'arbre syntaxique abstrait est représenté par des classes Python avec héritage :

ASTNode

├── Program

├── Block

├── ConstDecl

├── VarDecl

├── Statement

│ ├── Assign

│ ├── If

│ ├── While

│ ├── For

│ ├── Repeat

│ └── Compound

├── Expression

│ ├── BinaryOp

│ ├── UnaryOp

│ ├── VarRef

│ └── Literal

6.2 Classes AST Principales

```
@dataclass
class Program(ASTNode):
    name: str
    block: 'Block'
    lineno: int = 0
    col: int = 0

    def to_tree_string(self, level=0):
        indent = "  " * level
        result = f"{indent}Program(name='{self.name}')\n"
```

```
        result += self.block.to_tree_string(level + 1)
        return result

@dataclass
class BinaryOp(Expression):
    op: str
    left: Optional['Expression']
    right: Optional['Expression']
    lineno: int = 0
    col: int = 0

    def to_tree_string(self, level=0):
        indent = "  " * level
        result = f"{indent}BinaryOp(op='{self.op}'):\n"
        result += f"{indent}  left:\n"
        if self.left:
            result += self.left.to_tree_string(level + 2)
        result += f"{indent}  right:\n"
        if self.right:
            result += self.right.to_tree_string(level + 2)
        return result
```

6.3 Exemple d'AST Généré

Pour le programme :

```
program test;
var x : integer;
begin
    x := 5 + 3;
end.
```

L'AST généré :

```
Program(name='test')
  Block:
    VarDeclarations:
      VarDecl(name='x', type='integer')
    Statements:
      Assign:
        target:
          VarRef('x')
        value:
          BinaryOp(op='+'):
            left:
              Literal(5)
            right:
              Literal(3)
```

7. Gestion des Erreurs

7.1 Classes d'Erreurs

```
class LexicalError(Exception):
    def __init__(self, message, lineno=None, col=None):
        super().__init__(message)
        self.lineno = lineno
        self.col = col


class SyntaxError_(Exception):
    def __init__(self, message, lineno=None, col=None):
        super().__init__(message)
        self.lineno = lineno
        self.col = col
```

7.2 Messages d'Erreur Explicites

Erreur lexicale :

```
Caractère non reconnu '@' à la ligne 2, colonne 5
var @x : integer;
    ^
```

Erreur syntaxique :

```
Erreur syntaxique: caractère inattendu '=' à la ligne 3, colonne 7
  x = 5;
    ^
```

7.3 Récupération d'Erreurs

L'analyseur syntaxique utilise les mécanismes de PLY pour :

1. Détecter les tokens invalids
2. Pointer la position exacte de l'erreur
3. Fournir le contexte de la ligne
4. Continuer l'analyse après certaines erreurs

8. Interface Utilisateur

8.1 Architecture Streamlit

L'interface est construite avec Streamlit pour fournir une expérience web interactive :

Composants principaux :

1. **Éditeur de code** : Zone de texte avec coloration syntaxique
2. **Exemples prédéfinis** : Programmes de démonstration
3. **Contrôles d'analyse** : Boutons pour chaque phase de compilation
4. **Visualisation des résultats** : Affichage formaté des tokens et AST

8.2 Fonctionnalités de l'Interface

- **Analyse lexicale** : Affichage des tokens avec statistiques

- **Analyse syntaxique** : Validation de la structure et aperçu AST
- **Construction d'AST** : Représentation arborescente textuelle
- **Gestion des erreurs** : Messages détaillés avec localisation

8.3 Code de l'Interface

```
# Configuration de la page
st.set_page_config(
    page_title="Compilateur Mini-Pascal",
    page_icon=" 🔍 ",
    layout="wide",
    initial_sidebar_state="expanded"
)

# Boutons d'analyse
if st.button(" 🔍 Analyse Lexicale", use_container_width=True):
    if code.strip():
        st.session_state.compiler.set_source(code)
        tokens, error = st.session_state.compiler.lexical_analysis()
        st.session_state.last_analysis = (tokens, error)
        st.session_state.analysis_type = "lexical"
        st.rerun()
```

9. Tests et Validation

9.1 Tests Unitaires

Test d'analyse lexicale :

```
def test_lexer():
    lexer.input("program test; begin x := 5; end.")
    tokens = []
```

```
    for tok in lexer:
        tokens.append((tok.type, tok.value))
    assert tokens[0] == ('PROGRAM', 'program')
    assert tokens[1] == ('ID', 'test')
```

Test d'analyse syntaxique :

```
def test_parser():
    ast = parser.parse("program test; begin x := 5; end.")
    assert isinstance(ast, Program)
    assert ast.name == 'test'
    assert len(ast.block.statements) == 1
```

9.2 Cas de Test

Catégorie	Exemple	Résultat Attendu
Programme simple	`program p; begin end.`	Succès
Déclarations	`const PI=3.14; var x:real;`	Succès
Structures contrôle	`if x<5 then x:=1 else x:=2;`	Succès
Erreur lexicale	`var @x:integer;`	Erreur détectée
Erreur syntaxique	`x = 5;`	Erreur détectée
Expressions complexes	`y := (x+5)*3/2;`	Succès

9.3 Couverture de Tests

- **Tokens** : 100% des tokens définis sont reconnus
- **Productions** : Toutes les règles de grammaire sont testées
- **Erreurs** : Messages d'erreur pour tous les cas d'erreur connus

- **Interface** : Toutes les fonctionnalités de l'interface sont testées

10. Conclusion

10.1 Bilan du Projet

Le compilateur Mini-Pascal réalisé satisfait toutes les exigences spécifiées :

Objectifs atteints :

- ☑ Analyse lexicale complète avec gestion des erreurs
- ☑ Analyse syntaxique LR(0) fonctionnelle

- ☑ Construction et visualisation de l'AST
- ☑ Interface web interactive avec Streamlit
- ☑ Gestion d'erreurs détaillée avec localisation
- ☑ Support d'un sous-ensemble significatif de Pascal

10.2 Limitations et Améliorations

Limitations actuelles :

1. **Analyse sémantique limitée** : Vérifications de type basiques
2. **Optimisations absentes** : Pas d'optimisation de code
3. **Génération de code** : Non implémentée
4. **Bibliothèque standard** : Fonctions mathématiques limitées

Améliorations futures :

1. **Analyse sémantique avancée** : Vérification de types complète
2. **Génération de code intermédiaire** : Représentation à trois adresses
3. **Optimisations** : Élimination de code mort, propagation de constantes
4. **Extension du langage** : Tableaux, enregistrements, fonctions

10.3 Références

1. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*
2. PLY Documentation : https://www.dabeaz.com/ply/
3. Streamlit Documentation : https://docs.streamlit.io/
4. Pascal ISO Standard 7185

Annexe A : Installation et Exécution

1. Installation des dépendances :

```bash
pip install -r requirements.txt
```

2. Lancement de l'application :

```bash
streamlit run app.py
```

3. Accès à l'interface :
   Ouvrir http://localhost:8501 dans un navigateur web

Annexe B : Structure du Code Source

Voir la section 2.3 pour la structure complète des fichiers.

Annexe C : Exemples de Programmes

Des exemples sont disponibles dans l'interface Streamlit sous "Charger un exemple".