Rapport du Projet de Compilateur Micro-Assembleur

Nom: OTERO Clément Groupe: ING1

January 17, 2025

Contents

1 Introduction **2**

2 Grammaire **2**

3 Conception du Compilateur **2**

4 Description des Modules **2**
4.1 Lexer 2
4.2 Parser 3
4.3 Semantic Analyzer 3
4.4 Génération de Code C 3

5 Gestion des Erreurs **3**

6 Programme Exemple **4**

7 Conclusion **4**

1 Introduction

Ce rapport documente la conception, l'implémentation et les fonctionnalités d'un compilateur pour un langage micro-assembleur. Le projet comprend les étapes suivantes :

- Définition de la grammaire du langage micro-assembleur.
- Conception et implémentation des différentes étapes du compilateur.

2 Grammaire

La grammaire du langage micro-assembleur est définie comme suit :

```
Programme -> VarSection InstructionsSection
VarSection -> "Var" Declarations
Declarations -> Declaration {"," Declaration} ";"
Declaration -> IDENTIFIER ":" Type
Type -> "byte" | "Array" "[" NUMBER "]"
InstructionsSection -> "Instructions" Instruction+
Instruction -> NUMBER ":" Operation ";"
Operation -> MOV | ADD | SUB | MULT | DIV | JMP | JZ | ...
Operand -> IDENTIFIER | NUMBER | IDENTIFIER "[" Expression "]"
```

3 Conception du Compilateur

Le compilateur est conçu en plusieurs modules :

- **Lexer** : Analyse lexicale pour transformer le code source en une liste de tokens.
- **Parser** : Analyse syntaxique pour construire un AST (Abstract Syntax Tree).
- **Semantic Analyzer** : Vérification des règles sémantiques, comme les types et les accès aux tableaux.
- **C Compiler** : Génération de code en langage C.

4 Description des Modules

4.1 Lexer

Le Lexer analyse les caractères pour identifier les mots-clés, identifiants, opérateurs, et autres composants. Voici un extrait de code :

Listing 1: Extrait du Lexer

```
class Lexer:
    def tokenize(self):
        tokens = []
        while self.current_char is not None:
```

```
            if self.current_char.isspace():
                self.skip_whitespace()
            elif self.current_char.isdigit():
                tokens.append(Token(TokenType.NUMBER, str(self.get_number()
            elif self.current_char.isalpha():
                identifier = self.get_identifier()
                if identifier in KEYWORDS:
                    token_type = KEYWORDS[identifier]
                else:
                    token_type = TokenType.IDENTIFIER
                tokens.append(Token(token_type, identifier, self.line, self
            ...
```

4.2 Parser

Le Parser construit un AST en respectant la grammaire définie. Exemple de méthode :

Listing 2: Extrait du Parser

```
def parse_declarations(self):
    declarations = []
    self.match(TokenType.VAR)
    while self.current_token.type != TokenType.INSTRUCTIONS:
        decl = self.parse_declaration()
        declarations.append(decl)
        if self.current_token.type == TokenType.COMMA:
            self.advance()
    return declarations
```

4.3 Semantic Analyzer

Vérifie les erreurs comme l'utilisation de variables non déclarées ou l'accès hors limites d'un tableau.

Listing 3: Extrait du Semantic Analyzer

```
def check_array_index(self, index, array_symbol):
    if isinstance(index, Number):
        if index.value < 0 or index.value >= array_symbol.size:
            raise Exception(f"Erreur: Index hors limites pour le tableau '{
```

4.4 Génération de Code C

Le compilateur n'étant pas fini ne fonctionne pas encore. Certaines fonctionnalité sont defectueuse dans le compilateur ce qui l'empeche de génère du code C structuré et exécutable.

5 Gestion des Erreurs

Le compilateur détecte les erreurs lexicales, syntaxiques et sémantiques. Par exemple :

- Variable non déclarée.

6 Programme Exemple

```
Var
    x: byte,
    y: Array[10],
    sum: byte;
Instructions
    0: mov sum, 0;
    1: input(y[0]);
    2: add sum, y[0];
    3: print(sum);
```

7 Conclusion

Ce projet non fini par faute de temps met en oeuvre un compilateur pour un langage micro-assembleur.