Compiler Design Lab Report

University of Chittagong

Compiler Design Lab, 2023

Group 8

Authors:
Md. Masud Mazumder
Tonmoy Chandro Das
Tareq Rahman Likhon Khan
Md. Siam
Rabbi Hasan
Imtiaz Ahamad Imon
Md. Mustak Ahmed
Abdullah Al Faruque
Md. Hasan Mia
Arif Hasan
Abu Noman Shawn Shikdar

Instructor:
Nasrin Sultana
Assistant Professor
Dept. of Computer Science & Engineering
University of Chittagong, Chittagong - 4331

Date: September 19, 2023

Contents

1 Introduction

Compiler design is a fundamental aspect of computer science and software engineering. It plays a pivotal role in the software development process by translating high-level programming languages into low-level machine code that can be executed by a computer's central processing unit (CPU). The development of compilers is essential to bridge the gap between human-readable code and machine-executable instructions, enabling programmers to express complex algorithms and logic in a more abstract and understandable manner.

The primary objective of this lab report is to explore the key concepts and components involved in the design and construction of a compiler. We will delve into the various phases of compilation, including **lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, and code generation**. Understanding these phases is crucial for anyone involved in compiler development or interested in gaining a deeper insight into how programming languages are processed and executed.

In this lab, we will also examine specific topics and tasks related to compiler design, such as regular expressions and finite automata for lexical analysis, context-free grammars and parsing techniques for syntax analysis, and symbol tables for managing program identifiers and their attributes. Moreover, we will discuss how compilers can optimize code for better execution performance.

The knowledge and skills gained from this lab are not only valuable for compiler developers but also for software engineers, as they provide a deeper understanding of the internal workings of programming languages and their translation into machine code. A well-designed and efficient compiler can significantly impact the performance of software systems, making compiler design an integral part of software engineering education.

As our lab task is to design a parser for **Complex Bengali Sentences**, this lab report will document our exploration of concepts of compiler design based on a sentence parser for the assigned kind of sentences, providing insights into the intricate world of compiler design and its importance in the realm of computer science.

1.1 Phases of Compiler

A compiler is a complex software tool that translates high-level programming code into low-level machine code or an intermediate representation. To achieve this transformation, compilers are typically divided into several well-defined phases (Figure 1), each responsible for a specific aspect of the compilation process. These phases work together systematically to ensure that the source code is analyzed, transformed, and generated into an executable form. In this section, we will explore the key phases of a compiler.

- **Lexical Analysis:** The first phase of a compiler is lexical analysis, also known as scanning. This phase reads the source code and breaks it into a stream of tokens, which are the basic units of the programming language. The tokens are then passed on to the next phase for further processing.
- **Syntax Analysis:** The second phase of a compiler is syntax analysis, also known as parsing. This phase takes the stream of tokens generated by the lexical analysis phase and checks whether they conform to the grammar of the programming language. The output of this phase is usually an Abstract Syntax Tree (AST).
- **Semantic Analysis:** The third phase of a compiler is semantic analysis. This phase checks whether the code is semantically correct, i.e., whether it conforms to the language's type system and other semantic rules.
- **Intermediate Code Generation:** The fourth phase of a compiler is intermediate code generation. This phase generates an intermediate representation of the source code that can be easily translated into machine code.
- **Code Optimization:** The fifth phase of a compiler is optimization. This phase applies various optimization techniques to the intermediate code to improve the performance of the generated machine code.
- **Code Generation:** The final phase of a compiler is code generation. This phase takes the optimized intermediate code and generates the actual machine code that can be executed by the target hardware.

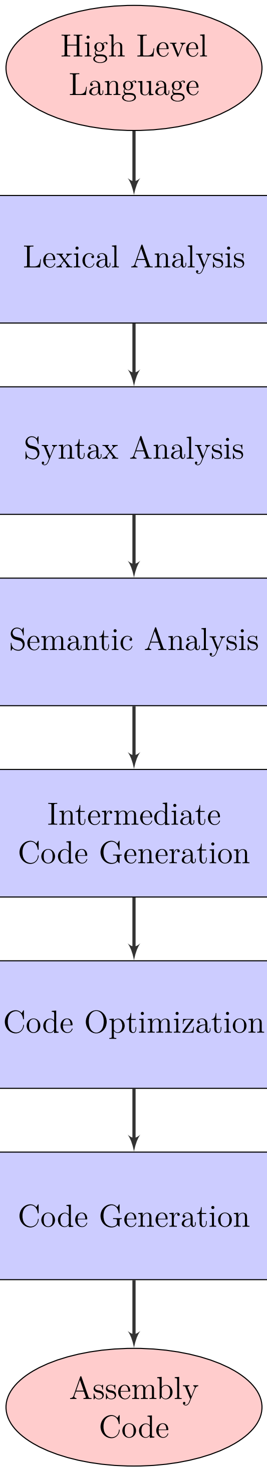

Figure 1: Phases of a compiler

2 Task Description

2.1 Task Overview

The task at hand involves the design and implementation of a sentence parser and compiler for handling complex Bengali sentences. Natural language processing (NLP) and compiler design are two distinct but interconnected fields. Combining them presents a unique challenge, as it requires us to bridge the gap between the rich and often intricate linguistic structures of Bengali and the formalized, executable code of a compiler.

2.2 Motivation

The motivation behind this task is twofold. First, Bengali is a language of immense depth and complexity, with a rich syntax and extensive vocabulary. Developing tools that can analyze and process Bengali sentences is of significant linguistic and practical value. Second, the integration of compiler design principles into NLP opens up opportunities for automating linguistic analysis, which can be applied to various fields, including machine translation, sentiment analysis, and information retrieval.

2.3 Project Goals

- Develop a CFG for Complex Sentences in Bengali that accurately models complex sentence structures.

- Implement a parser that can analyze Complex Sentences in Bengali sentences based on the CFG.

3 Context Free Grammar (CFG)

A crucial step in this project is the development of a CFG that accurately captures the syntax and structure of complex Bengali sentences. The CFG will include rules for sentence formation, noun phrases, verb phrases, adjectives, and various sentence connectors. It must consider both grammatical correctness and semantic meaning.

For our task, we first chose 10 Bengali Complex sentences. These are as follows:

1. যদি তুমি আসো, তবে আমি যাবো।
2. যে পড়োশানা করেব, সে গিাড়েত চড়বে।
3. যত পড়েব, তত জানবে।
4. যেমনটা আমি চেয়িছলাম, তেমনটা পাইনি।
5. যারা দেশপ্রেমিক, তারা দেশেক ভোলাবোস।
6. যদিও লোকটি গিরব, তথািপ সে অতিথিপরায়ণ।
7. যতই পরিশ্রম করেব, ততই ফল পোব।
8. যে অপরাধ করেছে, সে শাস্তি পেয়েছে।
9. যখন বিপদ আসে, তখন দুঃখও আসে।
10. যে অন্ধ, তাকে আলো দাও।

To satisfy this sentence structure we then define a Context Free Grammar (CFG) as follows:

```
বাক্য --> সাপেক্ষ-সর্বনাম-শুরু + খন্ড-বাক্য + , + সাপেক্ষ-সর্বনাম-শেষ + খন্ড-বাক্য
খন্ড-বাক্য --> বিশেষ্য + খন্ড-বাক্য-শেষ । সর্বনাম + খন্ড-বাক্য-শেষ
খন্ড-বাক্য-শেষ --> ক্রিয়া । বিশেষণ
সাপেক্ষ-সর্বনাম-শুরু --> যদি । যে । যত । যেমনটা । যারা । যদিও । যতই । যখন
সাপেক্ষ-সর্বনাম-শেষ --> তবে । সে । তত । তেমনটা । তারা । তথাপি । ততই । তখন
বিশেষ্য --> বিপদ । দুঃখও । অপরাধ । শাস্তি । আলো । পড়াশোনা । গাড়িতে । e
বিশেষণ --> গরিব । অন্ধ । দেশপ্রেমিক । অতিথিপরায়ণ
সর্বনাম --> তুমি । আমি । আমার ভাই । লোকটি । সে । e
ক্রিয়া --> আসো । দাও । আসে । করেছে । পেয়েছে । করবে । পাবে । চড়বে । ভালোবাসে
যাবো । জানবে । পড়বে । এসেছিলো । চেয়েছিলাম । পাইনি । e
```

4 Lexical Analysis

4.1 Overview

Lexical analysis, also known as scanning or tokenization, is the first phase in the process of parsing and compiling complex Bengali sentences. It converts the High-level input program into a sequence of Tokens. In this phase, the raw input sentence, composed of characters, is broken down into meaningful units known as tokens. These tokens serve as the basic building blocks for subsequent phases of parsing and analysis.

`Tokens:` Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In the case of our task, Noun, Pronoun, Verb, Adjective, etc. can be considered as tokens. For example, the sentence “যত পড়বে, তত জানবে।" contains the tokens: “যত", ‘‘পড়বে", ‘‘তত", ‘‘জানবে".

4.2 Role of Lexical Analysis

The primary goals of lexical analysis in the context of Bengali sentence parsing and compilation are as follows:

- **Tokenization:** Identify and tokenize the input sentence into discrete elements, such as words, punctuation, and special symbols, that are relevant for linguistic analysis.
- **Normalization:** Normalize the tokens to a consistent format to facilitate further processing. This includes handling variations in case, removing diacritics, and ensuring consistent spacing.
- **Error Detection:** Detect and report lexical errors in the input sentence, such as unrecognized characters or syntax errors at the token level.
- **Token Classification:** Categorize tokens into different classes, such as nouns, verbs, adjectives, pronouns, and connectors, based on their linguistic roles.

4.3 Lexical Analysis Code

In the context of the project focused on designing a sentence parser and compiler for complex Bengali sentences, the lexical analysis phase plays a crucial role in processing raw input sentences. The Python code below demonstrates the lexical analysis process, which tokenizes Bengali sentences into meaningful units while handling common delimiters.

Listing 1: Code for Lexical analysis

```python
def lexical_analysis(self, sentence):
    print("Lexical analysis started...")

    # Initialize an empty token
    token = ""

    # Iterate through each character in the input sentence
    for c in sentence:
        # Check for common delimiters: comma and Bengali full stop
        if c == "," or c == "।":
            # If the token is not empty, append it to the list of tokens
            if len(token) > 0:
                self.tokens.append(token)
            # Append the delimiter as a separate token
            self.tokens.append(c)
            # Reset the token
            token = ""
        # Check for whitespace or newline characters
        elif c == " " or c == "\n":
            # If the token is not empty, append it to the list of tokens
            if len(token) > 0:
                self.tokens.append(token)
            # Reset the token
            token = ""
        else:
            # Add the character to the current token and remove leading/trailing whitespace
            token += c
            token = token.strip()

    print("Lexical analysis successful...")
```

Explanation:

- The **lexical_analysis** function takes an input sentence in Bengali as its parameter.

- It initializes an empty token to capture segments of the sentence.
- The function iterates through each character in the input sentence, checking for common delimiters like commas and Bengali full stops ( ).
- When a delimiter is encountered, the current token is appended to the list of tokens, and the delimiter is added as a separate token.
- Whitespace and newline characters are used to determine the boundaries between tokens. When encountered, they trigger the addition of the current token to the list of tokens.
- Characters are accumulated in the current token, stripping leading and trailing whitespace.
- Finally, the function prints messages to indicate the start and successful completion of the lexical analysis process.

This code snippet represents the initial step in processing Bengali sentences, breaking them down into tokens that can be further analyzed and compiled. It demonstrates the essential functionality required for the lexical analysis phase of the project.

4.4 Lexical Analysis Result

The result of lexical analysis is the tokenization of the input sentence. After performing lexical analysis, the tokens are stored in the **self.tokens** list. These tokens represent the individual elements of the input sentence, separated based on the lexical rules you defined.

For example, if we pass the sentence "যে অন্ধ, তাকে আলো দাও ।" to the **Complex_sentence_parser**, the lexical analysis will break it down into tokens as follows:
["যে", "অন্ধ", ",", "তাকে", "আলো", "দাও", " ।"] Here's a breakdown of these tokens:

- "যে" (Bengali word)
- "অন্ধ" (Bengali word)
- "," (Comma punctuation)
- "তাকে" (Bengali word)
- "আলো" (Bengali word)
- "দাও" (Bengali word)
- " ।" (full stop (dari) punctuation)

These tokens represent the individual elements of the input sentence, where Bengali words are separated by commas and spaces, and punctuation marks (the comma in this case) are treated as separate tokens. These tokens are the result of the lexical analysis and serve as the input for the subsequent syntax analysis phase of your parser.

4.5 Conclusion

The lexical analysis phase serves as the foundational step in the complex task of parsing and compiling Bengali sentences. By breaking down sentences into manageable tokens, normalizing them, and classifying them based on their linguistic roles, we pave the way for subsequent phases of syntactic and semantic analysis. The success of this phase greatly influences the overall accuracy and effectiveness of the sentence parser and compiler for complex Bengali sentences.

5 Syntax Analysis

5.1 Overview

The syntax analysis phase in the context of our project plays a pivotal role in dissecting and understanding the structural arrangement of Bengali sentences. It is the second crucial step after lexical analysis and focuses on examining how words and phrases are structured to convey meaning.

5.2 Role of Syntax Analysis

The primary objectives of the syntax analysis phase are as follows:

1. **Parse Tree Generation:** Construct a parse tree or syntax tree that represents the hierarchical structure of the input sentence. This tree illustrates how words and phrases relate to one another in terms of syntactic rules.

2. **Grammar Compliance:** Validate the input sentence against a predefined Context-Free Grammar (CFG) for Bengali. Detect and report any syntax errors or violations of grammatical rules.

3. **Semantic Analysis Preparation:** Prepare the groundwork for semantic analysis by providing a structured representation of the sentence's syntactic elements. This paves the way for the subsequent phase of semantic analysis.

5.3 Parsing Techniques

In the context of our project, parsing Bengali sentences involves the use of context-free grammars and parsing techniques. Some of the commonly used parsing techniques include:

- **Top-Down Parsing:** Starting from the root of the parse tree and working down to the leaves, this technique is often used for LL(k) grammars.

- **Bottom-Up Parsing:** Beginning with the leaves and working upwards to construct the parse tree, this approach is suitable for LR(k) grammars.

- **Chart Parsing:** Chart parsing is employed for grammars with potentially ambiguous structures. It generates multiple parse trees, allowing for the exploration of various interpretations of the same sentence.

5.4 Syntax Analysis Code

In the context of our project, the syntax analysis phase is responsible for validating the structural correctness of Bengali complex sentences based on predefined syntactic rules. The Python code provided below implements the syntax analysis process using a top-down parsing approach known as **Recursive Descent** parsing. Recursive descent parsing involves creating a set of recursive functions, each corresponding to a non-terminal in the grammar, to recursively parse the input sentence to verify the sentence's grammatical structure.

Listing 2: Code for Syntax analysis

```
def get_next_token(self):
    # Move to the next token in the list
    self.index += 1

    # Check if the end of the token list is reached
    if self.index == len(self.tokens):
        return '#'

    return self.tokens[self.index]

def Noun(self, token):
    # Check if the token is a noun
    if token not in noun:
        return self.SS2(token)
    else:
        return self.SS2(self.get_next_token())

def Pronoun(self, token):
    # Check if the token is a pronoun
    if token not in pronoun:
        return self.SS2(token)
    else:
        return self.SS2(self.get_next_token())

def Verb(self, token):
    # Check if the token is a verb
    if token not in verb:
        return False
    else:
        return True

def Adjective(self, token):
    # Check if the token is an adjective
    if token not in adjective:
        return False
    else:
        return True

def PS(self, token):
    # Check if the token is a pronoun start
    if token not in pronoun_start:
        raise Exception('Pronoun start does not exist.')
    else:
```

```python
        return True

def PE(self, token):
    # Check if the token is a pronoun end
    if token not in pronoun_end:
        raise Exception('Pronoun end does not exist.')
    else:
        return True

def SS(self, token):
    # Check if the token represents a valid sentence segment
    if self.Noun(token) or self.Pronoun(token):
        return True
    else:
        raise Exception('Invalid sentence segment.')

def SS2(self, token):
    # Check if the token represents a valid secondary sentence segment
    if self.Verb(token) or self.Adjective(token):
        return True
    else:
        return False

def Has_comma(self, token):
    # Check if the token is a comma
    if token == ',':
        return True
    else:
        raise Exception('Comma does not exist.')

def Has_dari(self, token):
    # Check if the token is a Bengali full stop (dari)
    if token == '।':
        return True
    else:
        raise Exception('Dari (full stop) does not exist.')

def parse(self):
    try:
        # Start the parsing process
        self.PS(self.get_next_token())
        self.SS(self.get_next_token())
        self.Has_comma(self.get_next_token())
        self.PE(self.get_next_token())
        self.SS(self.get_next_token())
        self.Has_dari(self.get_next_token())

        # Check if there are additional tokens remaining
        if self.index + 1 < len(self.tokens):
            raise Exception('This parser cannot handle this sentence parsing.')

        # Reset the index for future parsing
        self.index = -1

        print('Parsing successful.')
    except Exception as e:
        print('Parsing Error: ', str(e))
```

Explanation:

- The provided Python code outlines a syntax analysis process that verifies the structural correctness of Bengali complex sentences based on a set of predefined grammatical rules.
- The code employs a series of functions, each dedicated to checking specific aspects of sentence structure, including nouns, pronouns, verbs, adjectives, and more.
- The **parse** function serves as the main entry point, orchestrating the sequence of checks required for valid sentence parsing. It raises exceptions when encountering structural inconsistencies.
- The code is designed to handle Bengali complex sentence structures that conform to the defined rules, and it provides detailed error messages for any deviations.

This code represents a critical component of the syntax analysis phase in our project. It ensures that Bengali sentences adhere to grammatical rules before proceeding to further stages of analysis and compilation.

5.5 Parsing Result

Successful syntax analysis results in the generation of a parse tree or syntax tree that visually represents the sentence's syntactic structure. This tree can be instrumental in subsequent stages of semantic analysis and code generation.
The code included in the above notifies about the error or success result of parsing a sentence.

5.6 Conclusion

The syntax analysis phase is a critical component of our project's goal to design a sentence parser and compiler for complex Bengali sentences. By generating parse trees and validating sentences against a CFG, we gain insights into the structural composition of sentences. This phase prepares the groundwork for subsequent stages, including semantic analysis and code generation, bringing us closer to achieving our project's objectives.

6 Parsing Using ANTLR

6.1 Overview

ANTLR, or ANother Tool for Language Recognition, is a powerful and widely used tool for generating parsers, lexers, and compilers. It is known for its flexibility, ease of use, and strong support for creating domain-specific languages (DSLs) and interpreters. ANTLR is often used in the fields of programming language design, compiler construction, and other areas where parsing and code generation are essential tasks.

- **Parsing and Lexing**
  - ANTLR primarily focuses on parsing and lexing (tokenizing) input text based on a specified grammar. It can generate parsers and lexers for a wide range of languages and file formats.
- **Grammar Specification**
  - ANTLR grammars are specified using a custom syntax that is similar to EBNF (Extended Backus-Naur Form). This grammar definition is used to describe the syntax of the language or format you want to parse.
- **Lexer and Parser Generation**
  - ANTLR generates both lexers and parsers from a single grammar specification. The lexer recognizes individual tokens, while the parser builds a parse tree or abstract syntax tree (AST) based on the recognized tokens.
- **LL(*) Parsing Algorithm**
  - ANTLR uses a variation of the LL(k) parsing algorithm, known as LL(*), which allows for more expressive and context-sensitive grammars. This flexibility is particularly useful for parsing complex languages.
- **Target Languages**
  - ANTLR can generate parsers and lexers in various programming languages, including Java, C#, Python, JavaScript, and more. This makes it versatile and compatible with a wide range of development environments.
- **Tree Parsing**
  - ANTLR can generate tree walkers or visitors for traversing and processing parse trees or ASTs. This is essential for performing semantic analysis, optimization, and code generation in compilers and interpreters.
- **Error Handling**

    - ANTLR provides mechanisms for handling syntax errors and generating meaningful error messages, making it easier to diagnose and correct issues in the input code.

- **Integrated Development Environment (IDE)**
    - ANTLR offers a graphical IDE called ANTLRWorks, which provides a user-friendly environment for developing, testing, and debugging ANTLR grammars.

- **Community and Ecosystem**
    - ANTLR has a large and active community of users and contributors. It has been widely adopted in both academia and industry, and there are extensive resources, tutorials, and libraries available to support ANTLR-based projects.

- **Use Cases**
    - ANTLR is used in various domains, including programming language design, compiler construction, static analysis tools, configuration file parsing, and more. It is versatile and can be applied to a wide range of parsing tasks.

ANTLR is a powerful tool for projects that involve parsing and language recognition. It simplifies the process of creating parsers and lexers for various languages and formats, making it an invaluable resource for software developers, compiler engineers, and researchers in the field of formal language processing.

6.2 ANTLR Version 4

- We are utilizing ANTLR version 4 for our specific task.
- ANTLR 4 is the latest major version of ANTLR, offering significant improvements and enhancements over its predecessors.
- Improved Performance: ANTLR 4 is designed for improved parsing performance, making it suitable for large-scale language processing tasks.
- Target Independence: With ANTLR 4, generated parsers and lexers are more target-independent, allowing you to easily retarget them to different programming languages.
- Integrated Lexer Modes: ANTLR 4 introduces lexer modes, enabling lexers to switch between different tokenization modes based on context, which is particularly useful for languages with complex lexical structures.

- Support for Left-Recursive Grammars: ANTLR 4 has built-in support for left-recursive grammars, simplifying the grammar specification for languages with left-recursive rules.

- Improved Error Reporting: ANTLR 4 provides enhanced error reporting capabilities, making it easier to pinpoint and understand syntax errors in input code.

- Community and Documentation: ANTLR 4 benefits from an active user community and extensive documentation, including tutorials and resources to help users get started quickly.

6.3 Requirements for ANTLR Version 4

To utilize ANTLR version 4 effectively, you will need the following:

- **ANTLR 4 Runtime Library**: ANTLR provides runtime libraries for various programming languages. You should have the appropriate ANTLR 4 runtime library for your chosen target language. These libraries enable you to execute the parsers and lexers generated by ANTLR.

- **Java Runtime (Optional)**: If you are using the Java target, you will need a Java Runtime Environment (JRE) to run ANTLR and execute Java-based parsers and lexers.

- **ANTLRWorks (Optional)**: ANTLRWorks is an integrated development environment (IDE) for ANTLR grammars. While optional, it can greatly simplify the process of developing and debugging ANTLR grammars.

- **Operating System Compatibility**: ANTLR 4 is compatible with various operating systems, including Windows, macOS, and Linux. Ensure that your development environment supports your chosen target platform.

- **Text Editor or Integrated Development Environment (IDE)**: You'll need a text editor or an integrated development environment to write and edit ANTLR grammar files.

6.4 Setup ANTLR Version 4 and Usage

6.4.1 Install ANTLR 4

- Follow the instructions given at this link: https://github.com/antlr/antlr4-tools

- Otherwise you can follow the instructions given at https://antlr.org

6.4.2 Write Context-Free Grammar (CFG)

We first write our Context-Free Grammar(CFG) in ANTLR format where the file name is **CFG.g4**.

Listing 3: CFG in ANTLR format

```
grammar CFG;

sentence        :       pronoun_start ss ',' pronoun_end ss '।';

ss              :       noun ss2
                |       pronoun ss2
                ;

ss2             :       verb
                |       pronoun
                ;

pronoun_start   :       'যদি' | 'যে' | 'যত' | 'যেমনটা' | 'যারা' | 'যদিও'
                |       'যতই' | 'যখন';

pronoun_end     :       'তবে' | 'সে' | 'তত' | 'তেমনটা' | 'তারা'
                |       'তথাপি' | 'ততই' | 'তখন';

noun            :       'বিপদ' | 'দুঃখও' | 'অপরাধ' | 'শাস্তি' | 'আলো'
                |       'পড়াশোনা' | 'গাড়িতে';

adjective       :       'গরিব' | 'অন্ধ' | 'দেশপ্রেমিক' | 'অতিথিপরায়ণ';

pronoun         :       'তুমি' | 'আমি' | 'আমার ভাই' | 'লোকটি' | 'সে';

verb            :       'আসো' | 'দাও' | 'আসে' | 'করেছে' | 'পেয়েছে'
                |       'করবে' | 'পাবে' | 'চড়বে' | 'ভালোবাসে' | 'যাবো'
                |       'জানবে' | 'পড়বে' | 'এসেছিলো' | 'চেয়েছিলাম'
                |       'পাইনি';
```

6.4.3 Compile Grammar

The final part is to compile the CFG and generate a parse tree for a sentence that is valid for the grammar. In this regard, first of all, create a .txt (test.txt) file and write a test string to parse using the CFG. Then run the following commands sequentially:

```
class
antlr CFG.g4
javac CFG*.java
```

6.4.4 Run and Generate Parse Tree

Finally, run the parser and provide input to parse using the following command:

```
grun CFG sentence test.txt -gui
```

This will generate a graphical view of a parse tree according the sentence in the *test.txt* file.

7 Parsing Using Bison

7.1 Overview

Bison, a robust parser generator, simplifies the creation of parsers for programming languages and structured input. It operates by taking a formal grammar description and automating the parser's construction, aided by the efficient LALR(1) parsing algorithm. This tool is instrumental in compiler design, interpreters, and systems requiring comprehension of structured input languages. Bison-generated parsers, usually in C or C++, facilitate seamless integration into projects in these languages. Beyond parsing, Bison supports error handling and offers a strong community with extensive documentation and tutorials, enhancing its accessibility and utility in language processing and compiler construction.

Here are some other necessarry information about Bison:

- **Parsing and Lexing**
  - Bison primarily focuses on parsing input text based on a specified context-free grammar. It requires a separate lexer (e.g., Flex) to tokenize the input text.
- **Grammar Specification**
  - Bison grammars are specified using a custom syntax that defines the syntax of the language you want to parse. It uses context-free grammar rules.
- **Lexer and Parser**
  - Bison generates parsers that work in conjunction with a lexer (e.g., Flex). The lexer tokenizes the input, and the parser processes these tokens based on the grammar rules.
- **LALR Parsing Algorithm**
  - Bison uses a variation of the LALR(1) parsing algorithm, which is efficient and commonly used for parsing context-free grammars.
- **Error Handling**
  - Bison-generated parsers typically include error recovery and reporting mechanisms to handle syntax errors in the input.
- **Target Languages**
  - Bison generates parsers in C and C++, making it suitable for integration into projects written in these languages.
- **Use Cases**

  - Bison is used in various domains, including compiler construction, interpreters, code analysis tools, and any application that requires parsing and processing structured input.

In short, Bison simplifies parser creation, proving invaluable for software developers, compiler engineers, and language processing researchers. Its efficiency in generating parsers for diverse languages and formats accelerates project development. Bison's impact spans compiler design, text analysis tools, and beyond, driving innovation in computer science and software engineering.

7.2 Requirements for Bison Parsing in Windows

To utilize Bison for parsing in a Windows environment, you will need the following:

- **Windows Operating System**: Bison can be used on various versions of Windows, including Windows 10, 8, and 7. Ensure you have a working Windows environment to run Bison and related tools.
- **Bison Installer for Windows**: Download and install a Bison distribution for Windows. Bison is part of the GNU Compiler Collection (GCC). You can download it from the following official website.
- **Flex for Windows (Optional)**: If you also need a lexer, you might want to install Flex, which is often used alongside Bison. Flex is also part of the GNU Compiler Collection. You can download it from:
https://sourceforge.net/projects/winflexbison
- **C Compiler for Windows**: Bison generates C code for parsers, so you will need a C compiler for Windows to compile the generated code. Popular options include Microsoft Visual C++ (MSVC) or GCC for Windows.
- **Text Editor or Integrated Development Environment (IDE)**: You'll need a text editor or an integrated development environment to write and edit Bison grammar files and related code.
- **Command Prompt or Terminal**: You'll use the command prompt or terminal to run Bison, Flex (if needed), and compile the generated parser and lexer.

7.3 Requirements for Bison Parsing in Ubuntu

To utilize Bison for parsing in an Ubuntu environment, you will need the following:

- **Ubuntu Operating System**: Ensure you have Ubuntu installed and running on your system, providing the necessary Linux environment.
- **Bison Installation**: Install Bison, a parser generator, using the package manager for Ubuntu. Use the following command:

```
sudo apt-get install bison
```

- **Flex Installation (Optional)**: If you need a lexer alongside Bison, you can install Flex, a lexer generator. Use the following command:

  ```
  sudo apt-get install flex
  ```

- **C Compiler**: Ensure you have a C compiler installed, such as GCC, to compile the generated parser.
- **Text Editor or IDE**: Use a text editor or an integrated development environment to write and edit Bison grammar files and related code.
- **Terminal**: Use the terminal to run Bison, Flex (if needed), and compile the generated parser and lexer.

7.4 Bison Setup and Usage

To set up Bison and create a parser using Bison, follow these steps:

7.4.1 Install Bison

First, make sure Bison is installed on your system. You can download it from the official website.

7.4.2 Write the Bison Grammar

Create a Bison grammar file (e.g., `parser.y`) specifying the grammar rules for your language. Here's a simple example:

Listing 4: Bison grammar

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern int yylex();
extern FILE* yyin;
extern char* yytext;

void yyerror(const char* s);

#define MAX_PARAMS 10

extern int yynrhs;
```

```

%}



%union {
    int numval;
    char* strval;
}

%token PRINT
%token IF
%token ELSE
%token IDENTIFIER
%token NUMBER
%token ADD SUB MUL DIV
%token EQ NE LT LE GT GE
%token ASSIGN SEMICOLON
%token LPAREN RPAREN COMMA
%token LKARLI RKARLI

%left '+' '-'
%left '*' '/'
%left EQ NE LT LE GT GE
%right ASSIGN

%%

program:
    statement_list
    ;

statement_list:
    statement
    | statement statement_list
    ;

statement:
    declaration_or_assignment SEMICOLON
    | function_call SEMICOLON
    | print_statement SEMICOLON
    | if_statement
    ;

declaration_or_assignment:
    IDENTIFIER ASSIGN expression
    ;
```

```

function_call:
    IDENTIFIER LPAREN arg_list RPAREN
    ;

arg_list:
    expression
    | expression COMMA arg_list
    ;

print_statement:
    PRINT LPAREN expression RPAREN
    ;

expression:
    term
    | expression ADD term
    | expression SUB term
    ;

term:
    factor
    | term MUL factor
    | term DIV factor
    ;

factor:
    IDENTIFIER
    | NUMBER
    | LPAREN expression RPAREN
    | comparison
    ;

comparison:
    expression EQ expression
    | expression NE expression
    | expression LT expression
    | expression LE expression
    | expression GT expression
    | expression GE expression
    ;

if_statement:
    IF LPAREN expression RPAREN block
    | IF LPAREN expression RPAREN block ELSE block
    ;

```

```
block:
    LKARLI statement RKARLI
    ;

%%

int main(int argc, char* argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <input_file>\n", argv[0]);
        return 1;
    }

    FILE* input_file = fopen(argv[1], "r");
    if (!input_file) {
        perror("Error opening input file");
        return 1;
    }

    yyin = input_file;
    if(!yyparse()){
        printf("Parsing successful!\n");
    }

    fclose(input_file);
    return 0;
}


void yyerror(const char* s) {
    fprintf(stderr, "Error: %s\n", s);
    return;
}
```

7.4.3 Generate the Lexer using Flex

Write a Flex lexer (e.g., `lexer.l`) to tokenize the input. Here's a simple example:

Listing 5: Flex lexer

```
%{
#include <string.h>
#include "parser.tab.h"
%}

%%

"print"          { return PRINT; }
"if"             { return IF; }
```

```
"else"              { return ELSE; }
"@"[a-zA-Z][a-zA-Z0-9]{0,19} { yylval.strval = strdup(yytext); return IDENTIFIER; }
[0-9]+              { yylval.numval = atoi(yytext); return NUMBER; }
"+"                 { return ADD; }
"-"                 { return SUB; }
"*"                 { return MUL; }
"/"                 { return DIV; }
"=="                { return EQ; }
"!="                { return NE; }
"<"                 { return LT; }
"<="                { return LE; }
">"                 { return GT; }
">="                { return GE; }
"="                 { return ASSIGN; }
";"                 { return SEMICOLON; }
"("                 { return LPAREN; }
")"                 { return RPAREN; }
"{"                 { return LKARLI; }
"}"                 { return RKARLI; }
","                 { return COMMA; }
[ \t\n]             ;  // Skip whitespace and newline characters
.                   { fprintf(stderr, "Unexpected character: %s\n", yytext); }

%%

int yywrap(void) {
    return 1;
}
```

7.4.4 Compile the Lexer and Parser

Generate the lexer and parser using Flex and Bison by running the following commands sequentially:

```
flex lexer.l
bison -d parser.y
gcc -o my_parser lex.yy.c parser.tab.c
```

7.4.5 Run the Parser

Run the parser and provide input to parse using the following command:

```
./my_parser input_file.txt
```

8 Comparison between ANTLR4 and Bison

ANTLR4 and Bison are both popular parser generators, but they have some differences in terms of features, use cases, and implementation. Here's a comparison between the two:

1. **Language Support**:
   - **ANTLR4**: ANTLR4 is known for its broad language support. It can generate parsers for a wide range of languages, including C, C++, Java, Python, and more.
   - **Bison**: Bison is primarily used for generating parsers in C and C++.
2. **Grammar Language**:
   - **ANTLR4**: ANTLR4 uses a custom grammar specification language, which is highly readable and can represent complex grammar concisely. It supports both lexer and parser rules in the same grammar file.
   - **Bison**: Bison uses the YACC-style grammar specification language. Lexer rules are typically defined separately using tools like Flex.
3. **Lexer Integration**:
   - **ANTLR4**: ANTLR4 has integrated lexer and parser generation. Lexer rules can be defined alongside parser rules in the same grammar file.
   - **Bison**: Bison typically requires a separate lexer generator like Flex for defining lexer rules. This means two separate files for lexer and parser.
4. **Lexer Rules**:
   - **ANTLR4**: ANTLR4 lexer rules are more flexible and powerful. They can match complex patterns and perform custom actions.
   - **Bison**: Lexer rules in Bison are simpler and may require additional tools like Flex for more advanced lexer patterns.
5. **Parsing Algorithm**:
   - **ANTLR4**: ANTLR4 uses a recursive descent parsing algorithm. It can handle LL(*) grammars, which include left-factored and left-recursive rules.
   - **Bison**: Bison generates LALR(1) parsers. It cannot directly handle left-recursive rules and requires manual elimination.
6. **Error Handling**:
   - **ANTLR4**: ANTLR4 provides detailed error messages with information about expected tokens. It has better support for error recovery.

- **Bison**: Bison's error messages are typically less informative, and custom error recovery can be more challenging.

7. **Tooling**:

   - **ANTLR4**: ANTLR4 comes with a powerful IDE called ANTLRWorks for grammar development. It has good tooling support for various languages.
   - **Bison**: Bison is often used with other development environments or code editors, and it may not have the same level of integrated tooling.

8. **Community and Ecosystem**:

   - **ANTLR4**: ANTLR4 has a large and active community. It's widely used in various domains, including domain-specific language development.
   - **Bison**: Bison also has an active community, but it may be more common in traditional compiler development.

9. **License**:

   - **ANTLR4**: ANTLR4 is under the BSD license, which is more permissive.
   - **Bison**: Bison is typically used with code generated under the GPL license, which has stricter requirements for code distribution.

10. **Performance**:

    - **ANTLR4**: ANTLR4 parsers are generally slower than Bison parsers because of their LL(*) parsing strategy. However, performance can vary depending on the grammar and optimizations applied.
    - **Bison**: Bison generates LALR(1) parsers, which are often faster but may struggle with more complex grammar.

In summary, the choice between ANTLR4 and Bison depends on your specific project requirements and language preferences. ANTLR4 is more versatile and user-friendly for grammar development but may have slightly lower parsing performance. Bison is well-suited for traditional compiler development in C/C++ and offers good performance but may require more manual effort for grammar development.

9 Semantic Analysis

To be added...

A Appendix

A.1 Full Code

Listing 6: Your Code Title

```python

class Complex_sentence_parser:
    def __init__(self, sentence):
        # Initialize the index for token retrieval
        self.index = -1
        # Initialize a list to store tokens
        self.tokens = []
        # Perform lexical analysis on the input sentence
        self.lexical_analysis(sentence)

    def lexical_analysis(self, sentence):
        print("Lexical analysis started...")
        # Initialize an empty token
        token = ""
        # Iterate through each character in the sentence
        for c in sentence:
            # Check for common delimiters: comma and Bengali full stop (dari)
            if(c == "," or c == "।"):
                # If the token is not empty, append it to the list of tokens
                if len(token) > 0:
                    self.tokens.append(token)
                # Append the delimiter as a separate token
                self.tokens.append(c)
                # Reset the token
                token = ""
            # Check for whitespace or newline characters
            elif(c == " " or c == "\n"):
                # If the token is not empty, append it to the list of tokens
                if len(token) > 0:
                    self.tokens.append(token)
                # Reset the token
                token = ""
            else:
                # Add the character to the current token and remove leading/trailing whitespace
                token += c
                token = token.strip()
        print("Lexical analysis successful...")

    def get_next_token(self):
        # Move to the next token in the list
        self.index += 1
        # Check if the end of the token list is reached
        if(self.index == len(self.tokens)):
            return '#'
        # Return the next token
        return self.tokens[self.index]

    def Noun(self, token):
        # Check if the token is not in the set of known nouns
        if token not in noun:
            return self.SS2(token)
        else:
            return self.SS2(self.get_next_token())

    def Pronoun(self, token):
        # Check if the token is not in the set of known pronouns
        if token not in pronoun:
            return self.SS2(token)
        else:
            return self.SS2(self.get_next_token())

    def Verb(self, token):
        # Check if the token is in the set of known verbs
        if token not in verb:
            return False
        else:
            return True

    def Adjective(self, token):
        # Check if the token is in the set of known adjectives
        if token not in adjective:
            return False
        else:
            return True

    def PS(self, token):
        # Check if the token is not in the set of known pronoun starts
        if token not in pronoun_start:
            raise Exception('Pronoun start does not exist.')
        else:
            return True

    def PE(self, token):
```

```python
        # Check if the token is not in the set of known pronoun ends
        if token not in pronoun_end:
            raise Exception('Pronoun end does not exist.')
        else:
            return True

    def SS(self, token):
        # Check if the token is a valid noun or pronoun
        if self.Noun(token) or self.Pronoun(token):
            return True
        else:
            raise Exception('Invalid sentence segment.')

    def SS2(self, token):
        # Check if the token is a valid verb or adjective
        if self.Verb(token) or self.Adjective(token):
            return True
        else:
            return False

    def Has_comma(self, token):
        # Check if the token is a comma
        if token == ',':
            return True
        else:
            raise Exception('Comma does not exist.')

    def Has_dari(self, token):
        # Check if the token is a Bengali full stop (dari)
        if token == '।':
            return True
        else:
            raise Exception('Dari (full stop) does not exist.')

    def parse(self):
        try:
            # Start parsing with a pronoun start
            self.PS(self.get_next_token())
            # Expect a valid sentence segment
            self.SS(self.get_next_token())
            # Expect a comma
            self.Has_comma(self.get_next_token())
            # Expect a pronoun end
            self.PE(self.get_next_token())
            # Expect another valid sentence segment
            self.SS(self.get_next_token())
            # Expect a Bengali full stop (dari)
            self.Has_dari(self.get_next_token())

            # If there are more tokens, raise an exception
            if self.index + 1 < len(self.tokens):
                raise Exception('This parser cannot handle this sentence parsing.')

            # Reset the index for future parsing
            self.index = -1
            print('Parsing successful.')
        except Exception as e:
            print('Parsing Error: ', str(e))
```