



Texas College of Management & IT Bachelor of Information Technology (BIT)

**“Design, Construction and Simulation of Compiler Components: From
Lexical Analysis to Code Generation”**

Submitted By:

Name: Sajani Poudel

LCID: LC00017002770

Semester/year: 4th semester 2nd year

Submitted To:

Instructor Name: Mr. Saroj Ghimire

Course: Compiler Design and Construction

Date of Submission: 31st July, 2025

Texas College of Management & IT, Kathmandu

Abstract

This assignment explores the fundamental concepts and practical implementation of compiler design and construction. It covers the phases of a compiler from lexical analysis to code generation, focusing on syntax-directed translation, LL(1) grammar design, and parser construction. By applying formal grammar rules and automata theory, the study aims to deepen understanding of compiler components. The implementation section includes the development of lexical analysers and recursive descent parsers, complemented by a mini compiler case study. Overall, this work highlights both theoretical knowledge and hands-on experience essential for building efficient compilers.

Table of Contents	
Abstract.....	2
Introduction.....	4
Objectives	5
Section A: Theory Answers.....	6
Compiler Phases.....	6
Syntax Directed Translation.....	8
Parser Design	9
Section B: Numerical Solutions.....	12
Section C: Implementation/Programming-Based	17
Lexical Analysis Implementation.....	17
Recursive Descent Parser.....	18
Discussion and Challenges Faced	21
Conclusion	22

Introduction

Compiler design is a fundamental area of computer science focused on converting high-level programming languages into machine-level code that computers can execute. This process involves multiple phases, including lexical analysis, syntax analysis, semantic analysis, code optimization, and code generation. Each phase plays a vital role in ensuring that the source program is correctly translated into efficient and executable code. Understanding these phases and how they work together is crucial for creating reliable and effective compilers. This assignment explores the theory and practical implementation of these core compiler components, providing a comprehensive overview of the compilation process.

Objectives

The main objectives of this assignment are :

- To understand and describe the various phases of a compiler and the functions they perform.
- To study syntax-directed translation and the application of attribute grammars.
- To learn how to design grammars, convert them into LL(1) form, and build predictive parsing tables.
- To develop lexical analysers and parsers using programming languages Python.
- To produce intermediate code representations and gain the concepts of code optimization and generation.

Section A: Theory Answers

Compiler Phases

Explain the different phases of a compiler with a neat diagram. Briefly describe the role of each phase and give an example to illustrate how source code progresses through the compiler.

Ans: A compiler is a special program that converts code written in high-level programming languages like C or Java into machine language, which a computer can understand and execute. This process is done in several steps known as phases. These phases are grouped into two main parts:

Analysis Part (Front-End)

This part focuses on understanding and checking the source code written by the programmer. It ensures that the code is correct in terms of structure and logic but does not yet produce the final machine code. The analysis part includes the following four phases:

a. Lexical Analysis:

The lexical analysis phase is the first step of the compilation process. Its main role is to scan the entire source code and divide it into meaningful sequences known as tokens. Tokens are the smallest elements such as keywords, identifiers, operators, literals, and punctuation. During this phase, unnecessary elements like whitespaces and comments are removed. Additionally, the lexical analyzer maintains a symbol table that keeps track of identifiers and their attributes. This phase ensures that only valid characters and token patterns are passed to the next phase.

Example : `total = price * quantity + tax;`

The lexical analyzer produces tokens: `id1 = id2 * id3 + id4 ;`

Where `id1 = total`, `id2 = price`, `id3 = quantity`, and `id4 = tax`

b. Syntax Analysis (Parser)

The syntax analysis phase checks whether the tokens follow the correct grammatical structure of the programming language. It uses a set of grammar rules (such as BNF or CFG) to verify that the sequence of tokens forms valid statements. If valid, the compiler constructs a parse tree or abstract syntax tree (AST) to represent the hierarchical structure of the code. This phase is crucial for identifying missing symbols or incorrect nesting of expressions, such as unmatched parentheses or misplaced operators.

Example

```

      i. =
    /  \
total  +
      ii. /\
tax
    /\
price quantity
  
```

c. Semantic Analysis:

The semantic analysis phase ensures that the program has meaningful correctness. It goes beyond syntax to check for proper usage of data types, declarations, and scope of variables. It also performs type checking (e.g., ensuring integer is not assigned to a float variable without conversion) and identifies semantic errors like undeclared variables or function misuse. If any such inconsistencies are found, the compiler reports them as semantic errors.

Example: It confirms that total, price, quantity, and tax are declared before use and have compatible data types like all float or all int.

d. Intermediate Code Generation:

This phase converts the verified source code into a simplified, platform-independent intermediate representation. The goal is to make the next stages of optimization and code generation easier. The intermediate code is typically represented in the form of three-address code, quadruples, or postfix notation. Each expression is broken down into smaller and manageable operations using temporary variables.

Example:

```
t1 = price * quantity
t2 = t1 + tax
total = t2
```

Synthesis Part (Back-End)

The synthesis part uses the processed information from the analysis part to produce the final machine code. It also tries to make the code more efficient. This part includes two phases:

a. Code Optimization:

In this phase, the compiler attempts to improve the efficiency of the intermediate code without changing its output. The optimization may include removing redundant instructions, minimizing memory access, simplifying expressions, and reusing variables when possible. The result is a more efficient and compact version of the code that can execute faster and use fewer resources

Example

```
t1 = price * 10
t2 = t1 + tax
total = t2
```

b. Code Generation:

The final phase is code generation, where the optimized intermediate code is translated into machine-level instructions or assembly language for the target hardware. This phase maps high-level operations to low-level instructions, allocates memory/registers, and ensures that the final output is executable by the system's processor.

Example

MOV R1, price

MUL R1, quantity

ADD R1, tax

MOV total, R1

Syntax Directed Translation

Define syntax-directed definitions. Differentiate between synthesized and inherited attributes with examples. How is the dependency graph constructed for attribute evaluation?

Ans: A Syntax Directed Definition (SDD) is a method used to associate semantic information with the syntactic structure of a programming language. It extends the grammar rules by attaching semantic rules, which are used by the compiler to compute values, perform type checking, and ensure the correctness of the source code. In simple terms, an SDD connects the syntax (structure) of a program with its semantics (meaning) through the use of attributes. These attributes are attached to grammar symbols such as variables, expressions, or operators, and can hold important data like types, values, or names. The semantic rules specify how these attributes should be computed, thereby guiding the translation and analysis phases of the compiler.

The difference between synthesized and inherited attributes are:-

Synthesized Attribute	Inherited Attribute
A synthesized attribute is computed using the values of attributes from a non-terminal's child nodes in the parse tree.	An inherited attribute is determined based on the attributes of a non-terminal's parent or sibling nodes.
It is evaluated in a bottom-up manner, moving from the leaves of the parse tree toward the root.	It is evaluated in a top-down or left-to-right manner, flowing from parent to child or from left sibling to right sibling.
Synthesized attributes are commonly used to compute values such as types, results of expressions, or structure information.	Inherited attributes are used to pass context-sensitive information, such as scope, expected types, or formatting rules.
They depend only on the attributes of the child nodes in a production rule.	They depend on the attributes of parent or sibling nodes in a production rule.

In the rule $E \rightarrow E1 + T$, the attribute $E.val = E1.val + T.val$ is synthesized from the children $E1$ and T .	In the rule $T \rightarrow int$, the attribute $T.type$ may be inherited from the expected type set by the parent node.
---	--

A dependency graph is a visual tool used in syntax-directed translation to represent how attributes of grammar symbols in a parse tree depend on each other. It helps determine the correct order for evaluating attributes by clearly showing which attributes must be computed first. This ensures that all dependencies are met before evaluation, and also helps identify and prevent circular dependencies that could make evaluation impossible.

The steps to construct the Dependency Graph are given below:

1. Parse Tree Construction: Begin with the parse tree of the input string based on the given grammar.
2. Attribute Identification: Identify all the attributes (synthesized and inherited) associated with each node (symbol) in the parse tree.
3. Edge Creation:
 - For every semantic rule in the syntax-directed definition, determine which attributes depend on others.
 - For each such dependency, draw a directed edge from the attribute it depends on to the attribute being computed.
 - These directed edges form the dependency graph.

Once the dependency graph is constructed, the compiler performs a topological sort of the graph. This sorting determines an appropriate order in which the attributes can be evaluated so that no attribute is computed before the attributes it depends on.

If the dependency graph contains a cycle, it implies that some attributes are mutually dependent in a circular way, making the evaluation order undefined and thus invalid. Such situations must be avoided in well-designed syntax-directed definitions.

Parser Design

What are LL(1) grammars? Explain the steps to convert a grammar to LL(1) and the construction of a predictive parsing table. Also describe how recursive descent parsing works.

Ans: An LL(1) grammar is a type of context-free grammar that is used in top-down parsing techniques, particularly in predictive parsers. The first L stands for scanning the input from left to right, the second L represents producing a leftmost derivation of the input, and the number 1 indicates that the parser makes decisions using only one lookahead symbol. LL(1) grammars are simple and efficient to parse, but not all grammars fall into this category. The grammar must be non-left-recursive and non-ambiguous, and it should be possible to decide the correct production by examining only the current input symbol.

The steps to convert a grammar to LL(1) are explained below:

1. **Remove Left Recursion:**
Eliminate both direct and indirect left recursion to avoid infinite loops in top-down parsing. This is done by introducing new non-terminals and rewriting recursive productions appropriately.
2. **Apply Left Factoring:**
If multiple productions for a non-terminal share a common prefix, factor it out. This simplifies the grammar and allows the parser to make decisions using a single lookahead symbol.
3. **Compute FIRST Sets:**
The FIRST set of a symbol includes all terminals that can appear at the beginning of strings derived from that symbol. It helps decide which production rule to apply during parsing.
4. **Compute FOLLOW Sets:**
The FOLLOW set of a non-terminal consists of terminals that can appear immediately after that non-terminal in some valid derivation. It is especially important when ϵ (empty string) is in the FIRST set.
5. **Check LL(1) Condition:**
Ensure that for each non-terminal, the FIRST sets of its productions are disjoint. If ϵ is in a FIRST set, then its FOLLOW set must also be disjoint from other FIRST sets. This confirms that the grammar is suitable for LL(1) predictive parsing.

Construction of Predictive Parsing Table

- Create a table with rows for non-terminals and columns for terminals, including the end-of-input marker \$
- For each production $A \rightarrow \alpha$, insert the rule into the table cell $M[A, a]$ for every terminal a in $\text{FIRST}(\alpha)$.
- If ϵ is in $\text{FIRST}(\alpha)$, then also insert the production $A \rightarrow \alpha$ into cells $M[A, b]$ for each terminal b in $\text{FOLLOW}(A)$.
- If any cell contains more than one production, the grammar is not LL(1), indicating ambiguity or the need for further transformation.

The predictive parsing table allows the parser to determine the correct production to apply, without backtracking, based solely on the current input symbol and the top of the parsing stack.

Example

$E \rightarrow T E'$

$E' \rightarrow + T E' \mid \epsilon$

$T \rightarrow \text{id}$

This grammar is already in LL(1) form.

- $\text{FIRST}(E) = \{\text{id}\}$

- $\text{FIRST}(E') = \{+, \epsilon\}$
- $\text{FOLLOW}(E) = \{\$,)\}$
- $\text{FOLLOW}(E') = \{\$,)\}$

Using these sets, we construct the predictive parsing table as follows:

Non-Terminal	id	+	\$
E	$E \rightarrow T E'$		
E'		$E' \rightarrow + T E'$	$E' \rightarrow \epsilon$
T	$T \rightarrow \text{id}$		

Recursive descent parsing is a top-down parsing technique where each non-terminal in the grammar is implemented as a separate recursive function in code. These functions attempt to match the current portion of the input against the expected structure defined by the grammar rules. The parser uses a lookahead symbol to determine which production rule to apply and recursively calls functions for other non-terminals as needed. Terminals are matched directly with the input. This method is easy to implement and aligns well with LL(1) grammars because it avoids backtracking by making decisions using a predictive parsing table or conditional logic based on lookahead tokens.

Section B: Numerical Solutions

Section B (Numerical)

1. Lexical Analyser - Automata Construction.

Given the regular expressions:

$(a|b)^*abb$

- Construct a NFA using Thompson's construction.
- Construct the NFA to DFA using subset construction.

Ans

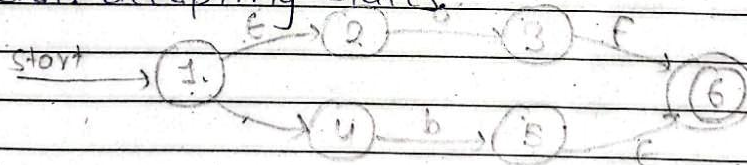
Given R.E is, $(a|b)^*abb$.

We have following steps to convert R.E to NFA.

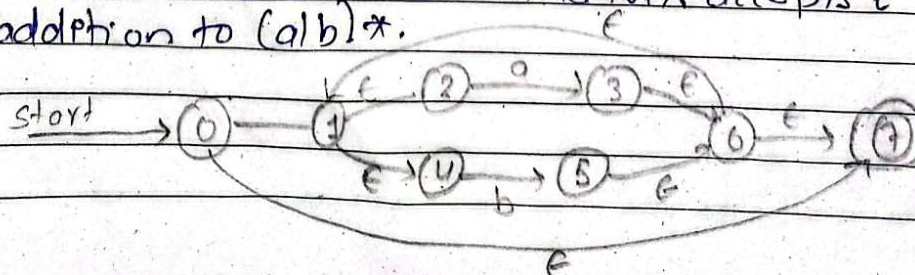
a) The NFA for single character regular expressions, ϵ, a, b



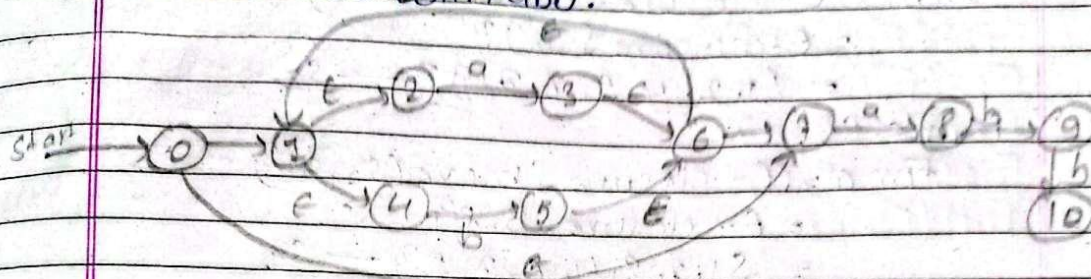
b) The NFA for union of 'a' & 'b': $a|b$ is constructed from the individual NFAs using the ϵ NFA as 'glue'. Remove the individual accepting state & replace with overall accepting states.



c) Kleene closure on $(a|b)^*$. The NFA accepts ϵ in addition to $(a|b)^*$.



d) Concatenate with abb.



To convert the above NFA to DFA using subset method, we have the following steps:-
For DFA, we have,

1) Starting state of DFA,

$$S_0 = \epsilon\text{-closure}\{\text{starting state of NFA}\}$$

$$= \epsilon\text{-closure}\{0\}$$

$$= \{0, 1, 2, 4, 7\}$$

$$\therefore S_0 = \{0, 1, 2, 4, 7\}$$

Mark S_0 ,

$$\text{For } a = \epsilon\text{-closure}\{\text{move}(S_0, a)\}$$

$$= \epsilon\text{-closure}\{3, 8\}$$

$$= \{1, 2, 3, 4, 6, 7, 8\} \rightarrow S_1$$

$$\text{For } b = \epsilon\text{-closure}\{\text{move}(S_0, b)\}$$

$$= \epsilon\text{-closure}\{5\}$$

$$= \{1, 2, 4, 5, 6, 7\} \rightarrow S_2$$

Mark S_1 ,

$$\text{For } a = \epsilon\text{-closure}\{\text{move}(S_1, a)\}$$

$$= \epsilon\text{-closure}\{3, 8\}$$

$$= \{1, 2, 3, 4, 6, 7, 8\} \rightarrow S_1$$

Date:
Page:

$$\begin{aligned}\text{For } h, &= \epsilon\text{-closure}\{\text{move}(s_1, b)\} \\ &= \epsilon\text{-closure}\{3, 9\} \\ &= \{1, 2, 4, 5, 6, 7, 9\} \rightarrow S_3\end{aligned}$$

Mark S_2 .

$$\begin{aligned}\text{For } a, &= \epsilon\text{-closure}\{\text{move}(s_2, a)\} \\ &= \epsilon\text{-closure}\{3, 8\} \\ &= \{1, 2, 3, 4, 6, 7, 8\} \rightarrow S_1\end{aligned}$$

$$\begin{aligned}\text{For } b, &= \epsilon\text{-closure}\{\text{move}(s_2, b)\} \\ &= \epsilon\text{-closure}\{5\} \\ &= \{1, 2, 4, 5, 6, 7\} \rightarrow S_2\end{aligned}$$

Mark S_3 ,

$$\begin{aligned}\text{For } a, &= \epsilon\text{-closure}\{\text{move}(s_3, a)\} \\ &= \epsilon\text{-closure}\{3, 8\} \\ &= \{1, 2, 3, 4, 6, 7, 8\} \rightarrow S_1\end{aligned}$$

$$\begin{aligned}\text{For } b, &= \epsilon\text{-closure}\{\text{move}(s_3, b)\} \\ &= \epsilon\text{-closure}\{5, 10\} \\ &= \{1, 2, 4, 5, 6, 7, 10\} \rightarrow S_4\end{aligned}$$

Mark S_4 ,

$$\begin{aligned}\text{For } a, &= \epsilon\text{-closure}\{\text{move}(s_4, a)\} \\ &= \epsilon\text{-closure}\{3, 8\} \\ &= \{1, 2, 3, 4, 6, 7, 8\} \rightarrow S_1\end{aligned}$$

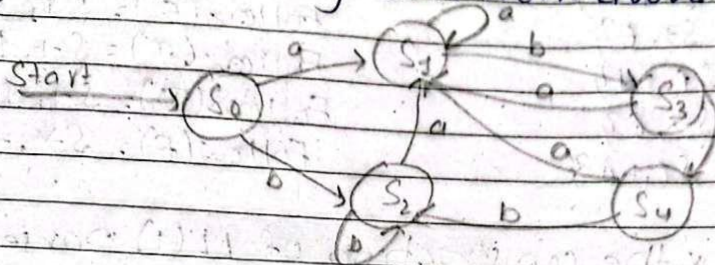
$$\begin{aligned}\text{For } b, &= \epsilon\text{-closure}\{\text{move}(s_4, b)\} \\ &= \epsilon\text{-closure}\{5\} \\ &= \{1, 2, 4, 5, 6, 7\} \rightarrow S_2\end{aligned}$$

\therefore Hence, S_0 is the starting state of DFA since 0 is a member of $S_0 = \{0, 1, 2, 4, 7\}$.
 S_4 is the accepting state of DFA since final state of NFA i.e., 10 is a member of S_4 .

Date:
Page:

$S_u = \{1, 2, 4, 5, 6, 7, 10\}$.

Now constructing DFA from above information,



Q. LL(1) parsing table construction for grammar,

$E \rightarrow TE'$

$E' \rightarrow +TE | E$

$T \rightarrow FT'$

$T' \rightarrow *FT' / \epsilon$

$F \rightarrow (E) \text{ id.}$

- Compute the First & Follow sets for all non-terminals.
- Construct the LL(1) parsing table.

Ans LL(1) parsing table or LL(1) parser is a type of top-down parser that takes or reads input from left to right in a compiler design. Here, '1' indicates that there is one symbol look ahead.

Now,

For the given grammar, we have,

- a) The given grammar contains all properties so, no need to remove it for left-factoring & left recursion. So, we can directly find out First & Follow.

Date:
Page:

For First,

$$\text{First}(E) = \{ \epsilon, id \}$$

$$\text{First}(E') = \{ +, \epsilon \}$$

$$\text{First}(T) = \{ \epsilon, id \}$$

$$\text{First}(T') = \{ *, \epsilon \}$$

$$\text{First}(F) = \{ \epsilon, id \}$$

For Follow,

$$\text{Follow}(E) = \{ \$, \epsilon \}$$

$$\text{Follow}(E') = \{ \$, \epsilon \}$$

$$\text{Follow}(T) = \{ +, \$, \epsilon \}$$

$$\text{Follow}(T') = \{ +, \$, \epsilon \}$$

$$\text{Follow}(F) = \{ *, +, \$, \epsilon \}$$

Q. Now, for the construction of LL(1) parse table, we have to take Non-terminal terms & follows of terminal terms.

Non-terminal	Terminals					
	id	*	+	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow TE'$		$E' \rightarrow E$	$E' \rightarrow E$	
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow FT'$	$T' \rightarrow E$		$T' \rightarrow E$	$T' \rightarrow E$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Section C: Implementation/Programming-Based

Lexical Analysis Implementation

Write a program (in C, Java, or Python) to simulate a simple lexical analyzer that takes an input string and identifies tokens such as keywords, identifiers, operators, and delimiters. Provide code, sample input, and output.

Ans:

Source Code in Python:

```
import re

def lexical_analyzer(input_string):
    keywords = {'if', 'else', 'while', 'for', 'int', 'float', 'return', 'main', 'print'}
    operators = {'+', '-', '*', '/', '=', '==', '!=', '<', '>', '<=', '>='}
    delimiters = {';', '(', ')', '{', '}', ',', '[', ']'}

    # Defining regex for each type
    identifier_regex = r'[a-zA-Z_][a-zA-Z0-9_]*'
    number_regex = r'\d+(\.\d+)?'
    operator_regex = r'==|!=|<=>|[\+\-\*/=<>]'
    delimiter_regex = r';(){}\[,\]'
    whitespace_regex = r'\s+'

    # List of (type, pattern) in order
    token_patterns = [
        ('OPERATOR', operator_regex),
        ('DELIMITER', delimiter_regex),
        ('NUMBER', number_regex),
        ('IDENTIFIER', identifier_regex),
        ('WHITESPACE', whitespace_regex)
    ]

    tokens = []
    i = 0
    while i < len(input_string):
        match_found = False

        for token_type, pattern in token_patterns:
            match = re.match(pattern, input_string[i:])
            if match:
                value = match.group(0)

                if token_type == 'WHITESPACE':
                    pass # skip spaces
```

```

        elif token_type == 'IDENTIFIER':
            if value in keywords:
                tokens.append((value, 'KEYWORD'))
            else:
                tokens.append((value, 'IDENTIFIER'))
        else:
            tokens.append((value, token_type))

        i += len(value)
        match_found = True
        break

    if not match_found:
        tokens.append((input_string[i], 'UNRECOGNIZED'))
        i += 1

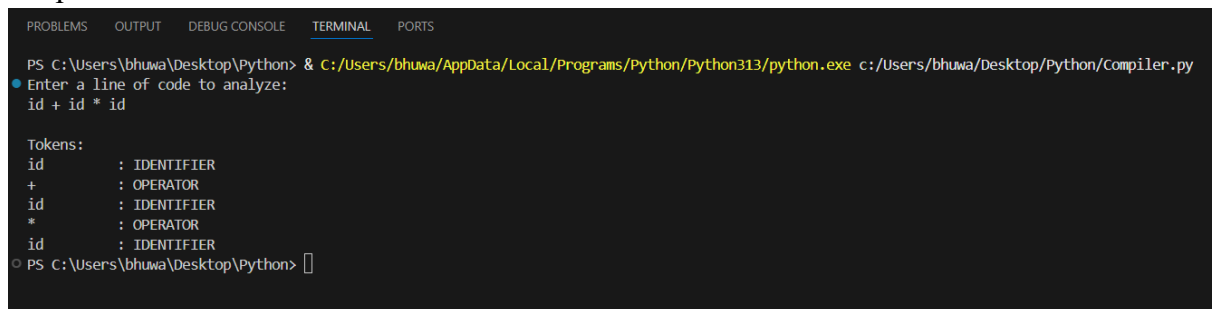
    return tokens

user_input = input("Enter a line of code to analyze:\n")
print("\nTokens:")
for token, type_ in lexical_analyzer(user_input):
    print(f'{token:<10} : {type_}')

```

Input: id + id * id

Output:



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\bhuwa\Desktop\Python> & C:/Users/bhuwa/AppData/Local/Programs/Python/Python313/python.exe c:/Users/bhuwa/Desktop/Python/Compiler.py
Enter a line of code to analyze:
id + id * id

Tokens:
id      : IDENTIFIER
+       : OPERATOR
id      : IDENTIFIER
*       : OPERATOR
id      : IDENTIFIER
PS C:\Users\bhuwa\Desktop\Python>

```

Recursive Descent Parser

1. Implement a recursive descent parser for the grammar:
 $E \rightarrow TE'$

$$E' \rightarrow +TE' \mid \epsilon$$
$$T \rightarrow \text{id}$$

Ans:

Source Code using Python:

class RecursiveDescentParser:

def __init__(self, input_str):

Split input into tokens by spaces

self.tokens = input_str.split()

self.pos = 0 # current position in tokens

def parse(self):

Parsing the expression and ensuring all tokens are consumed

tree = self.E()

if self.pos != len(self.tokens):

raise SyntaxError(f"Unexpected token: {self.tokens[self.pos]}")

return tree

def E(self):

E -> T E'

left = self.T()

while self.current() == '+':

self.consume('+')

right = self.T()

left = ('+', left, right)

return left

def T(self):

T -> id

token = self.current()

if token == 'id':

self.consume('id')

return ('id',)

else:

raise SyntaxError("Expected 'id'")

def current(self):

Return current token or None if done

if self.pos < len(self.tokens):

return self.tokens[self.pos]

return None

def consume(self, expected):

Consume expected token or raise error

if self.current() == expected:

```

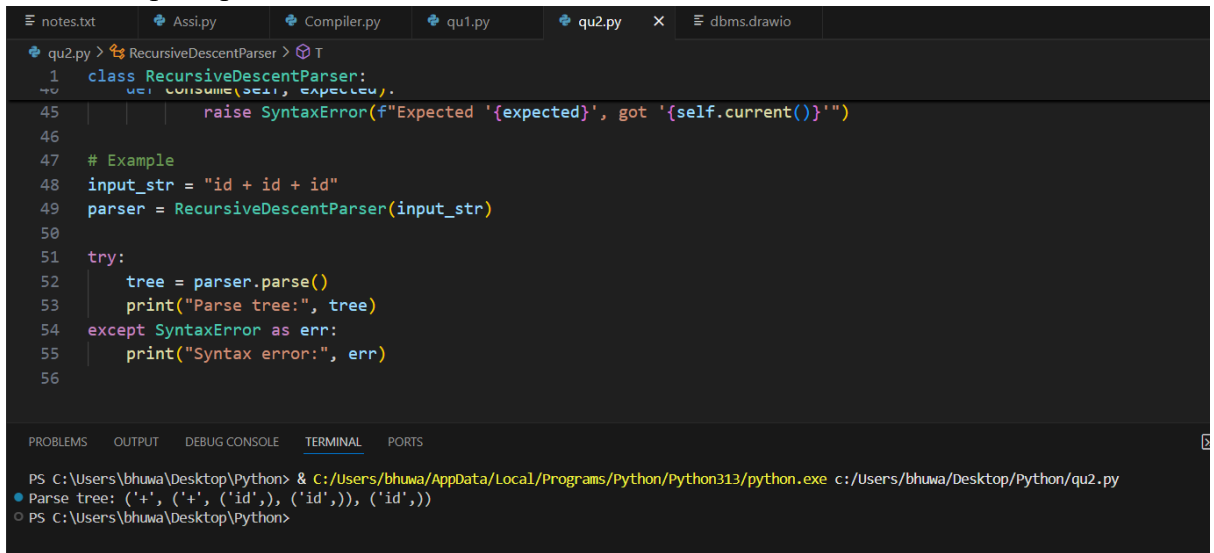
        self.pos += 1
    else:
        raise SyntaxError(f"Expected '{expected}', got '{self.current()}'")

# Example
input_str = "id + id + id"
parser = RecursiveDescentParser(input_str)

try:
    tree = parser.parse()
    print("Parse tree:", tree)
except SyntaxError as err:
    print("Syntax error:", err)

```

Parser accepts input like: id + id + id



The screenshot shows a code editor with a file named `qu2.py` containing the following Python code:

```

1  class RecursiveDescentParser:
2      def consume(self, expected):
3          ...
45     raise SyntaxError(f"Expected '{expected}', got '{self.current()}'")
46
47     # Example
48     input_str = "id + id + id"
49     parser = RecursiveDescentParser(input_str)
50
51     try:
52         tree = parser.parse()
53         print("Parse tree:", tree)
54     except SyntaxError as err:
55         print("Syntax error:", err)
56

```

Below the code editor, the terminal output is displayed:

```

PS C:\Users\bhuwa\Desktop\Python> & C:/Users/bhuwa/AppData/Local/Programs/Python/Python313/python.exe c:/Users/bhuwa/Desktop/Python/qu2.py
● Parse tree: ('+', ('+', ('id',)), ('id',)), ('id',))
○ PS C:\Users\bhuwa\Desktop\Python>

```

Discussion and Challenges Faced

This assignment provided valuable insights into the intricate process of compiler design, covering both theoretical concepts and practical implementations. Through the study of compiler phases, syntax-directed translation, and parsing techniques, a comprehensive understanding of how source code is translated into machine code was developed.

Implementing lexical analyzers and parsers enhanced practical skills in programming language translation, while the mini compiler case study allowed for the application of these concepts in a real-world scenario.

However, several challenges were encountered during the assignment which are mentioned below:

- Writing regular expressions for the lexical analyzer was tricky and took time to get right.
- Coding the recursive descent parser was difficult, especially avoiding infinite loops.
- Building the syntax tree from grammar rules was harder during implementation than in theory.
- Generating correct intermediate code was confusing at first.

Conclusion

In conclusion, this assignment greatly improved the understanding of compiler design and its practical aspects. It provided clear insights into how source code is processed and converted into machine code through various phases of compilation. The study of syntax-directed definitions, LL(1) grammar conversion, and predictive parsing helped build a strong foundation in theory. At the same time, implementing lexical analyzers and recursive descent parsers strengthened hands-on coding skills. Through the mini compiler case study, students were able to see how different components of a compiler come together in a working system. Overall, the assignment enhanced problem-solving abilities, emphasized the role of formal grammar in language translation, and laid the groundwork for more advanced learning in compiler construction.