BAHIRDAR INSTUETE OF TECHNOLOGY

FACULITY OF COMPUTING

DEPARTMENT OF SOFTWARE ENGINEERING

PRINCIPLE OF COMPILER DESIGN

INDIVUDUAL PROJECT

ERMIYAS MISGANAW…………………………………………………………………..1505942

QUESTION -50

1. **(Theory):** What is **syntax error recovery** in compilers? Give an example of one method.
2. **(C++):** Implement a **recursive descent parser** in C++ for strings with equal number of 'a's followed by 'b's.
3. **(Problem-solving):** Grammar:

   S → aSbS | ε

   Draw the **parse tree** for "aab".

1. SYNTAX ERROR RECOVERY IN COMPILERS - DETAILED EXPLANATION

Definition and Purpose

Syntax error recovery is a crucial mechanism in compiler design that enables the parser to continue analyzing source code after encountering syntax errors, rather than terminating at the first error. This allows compilers to detect and report multiple errors in a single compilation pass, significantly improving developer productivity.

Core Objectives:

- **Error Detection**: Identify syntax violations in the source code
- **Meaningful Reporting**: Generate clear, informative error messages
- **Recovery and Continuation**: Resume parsing to find additional errors
- **Minimal Disruption**: Skip or correct minimal portions of code to maintain context

Common Recovery Strategies:

1. Panic Mode Recovery (Most Widely Used)

When an error is detected, the parser discards input tokens until it encounters a synchronizing token from a predefined set (typically statement/block terminators like ;, }, end, etc.).

Example in C++:

cpp

```
int x = 10      // Missing semicolon
float y = 20.5; // Valid statement
```

Recovery Process:

1. Parser encounters float after 10 and expects ;
2. Enters panic mode and discards tokens until it finds ;
3. Continues parsing from float y = 20.5;
4. Reports: "Error: Expected ';' after statement at line 1"

Advantages:

- Simple to implement
- Guaranteed to terminate (no infinite loops)
- Effective for languages with clear statement boundaries

Limitations:

- May skip valid code after the error
- Can lead to cascading error reports

2. Phrase-Level Recovery

The parser performs local corrections by:

- Inserting missing tokens
- Deleting extraneous tokens
- Replacing incorrect tokens

Example Correction:

c

```
if (x > 0 {        // Missing closing parenthesis
    printf("Positive");
}
```

Recovery: Parser inserts) before { and continues

3. Error Productions

Grammar is augmented with intentional error productions that generate specific diagnostic messages:

text

```
statement → if ( expression ) statement
         | if ( expression error "Missing ')' in if statement"
```

4. Global Correction

Theoretically optimal but computationally expensive approach that finds the minimal sequence of changes to transform invalid input into valid syntax.

Implementation Considerations:

- Error recovery quality affects user experience
- Balance between skipping too much or too little input
- Maintain symbol table consistency during recovery
- Avoid misleading subsequent error messages

2. RECURSIVE DESCENT PARSER IMPLEMENTATION

Grammar Specification

The language $L = \{a^n b^n \mid n \geq 0\}$ (equal number of 'a's followed by equal number of 'b's)

Formal Grammar:

text

```
S → a S b | ε
```

Complete C++ Implementation

cpp

```cpp
#include <iostream>
#include <string>
#include <cctype>
```

```
class RecursiveDescentParser {
private:
    std::string input;
    size_t position;
    char currentToken;

    // Get next token from input
    void getNextToken() {
        if (position < input.length()) {
            currentToken = input[position++];
        } else {
            currentToken = '\0'; // End of input
        }
    }

    // Match expected token
    bool match(char expected) {
        if (currentToken == expected) {
            getNextToken();
            return true;
        }
        return false;
    }

    // Parse S → a S b | ε
    bool parseS() {
        // Try production S → a S b
        if (match('a')) {
            if (parseS()) {        // Recursive call for S
                if (match('b')) {
                    return true;
                } else {
                    std::cerr << "Error: Expected 'b' at position " << position << st
d::endl;
                    return false;
                }
```

```
      }
      return false;
    }
    // Try production S → ε (empty string)
    return true;
  }

public:
  RecursiveDescentParser(const std::string& str) : input(str), position(0
) {}

  // Main parsing function
  bool parse() {
    getNextToken(); // Initialize with first token
    if (!parseS()) {
      return false;
    }
    // Check if all input is consumed
    if (currentToken != '\0') {
      std::cerr << "Error: Extra characters after valid string" << std::endl
;
      return false;
    }
    return true;
  }

  // Test various strings
  static void runTests() {
    std::cout << "Testing Recursive Descent Parser for $a^nb^n$\n";
    std::cout << "==========================================\n
";

    std::string testCases[] = {"", "ab", "aabb", "aaabbb", "aab", "abb", "aba
", "aabbb"};

    for (const auto& test : testCases) {
      RecursiveDescentParser parser(test);
      bool result = parser.parse();
      std::cout << "String: \"" << test << "\" -> ";
      if (result) {
        std::cout << "ACCEPTED (Valid $a^nb^n$ pattern)\n";
      } else {
        std::cout << "REJECTED (Invalid pattern)\n";
      }
    }
  }
};

int main() {
  // Interactive mode
  std::cout << "Recursive Descent Parser for $a^nb^n$ Language\n";
  std::cout << "Enter string to parse (or 'test' for test suite): ";

  std::string input;
  std::cin >> input;

  if (input == "test") {
    RecursiveDescentParser::runTests();
  } else {
    RecursiveDescentParser parser(input);
    if (parser.parse()) {
      std::cout << "\nSUCCESS: String \"" << input << "\" follows $a^nb^n$ pa
ttern\n";
    } else {
      std::cout << "\nFAILURE: String \"" << input << "\" does not follow
$a^nb^n$ pattern\n";
    }
  }

  return 0;
}
```

Sample Output:

text

```
Testing Recursive Descent Parser for $a^nb^n$
========================================
String: "" -> ACCEPTED (Valid $a^nb^n$ pattern)
String: "ab" -> ACCEPTED (Valid $a^nb^n$ pattern)
String: "aabb" -> ACCEPTED (Valid $a^nb^n$ pattern)
String: "aaabbb" -> ACCEPTED (Valid $a^nb^n$ pattern)
String: "aab" -> REJECTED (Invalid pattern)
String: "abb" -> REJECTED (Invalid pattern)
String: "aba" -> REJECTED (Invalid pattern)
String: "aabbb" -> REJECTED (Invalid pattern)
```

Key Features:

1. **Recursive Structure**: Mirrors grammar productions directly
2. **Error Reporting**: Provides meaningful error messages with positions
3. **Complete Input Validation**: Ensures entire string is consumed
4. **Test Suite**: Comprehensive testing of valid and invalid cases

3. PARSE TREE FOR "AAB" USING GRAMMAR S → ASBS | E

Grammar Analysis:

text

```
S → a S b S | ε
```

This grammar generates strings with:

- 'a's and 'b's in interleaved pattern
- Potentially different numbers of 'a's and 'b's
- Empty string is valid

String Derivation for "aab":

Attempted Derivation 1:

text

```
S → a S b S
  → a (a S b S) b S     [First S → a S b S]
  → a (a ε b S) b S     [Inner S → ε]
  → a (a b S) b S
  → a (a b ε) b S       [Inner S → ε]
  → a a b b S
  → a a b b ε           [S → ε]
  → "aabb" ⍰ (Not "aab")
```

Attempted Derivation 2:

text

```
S → a S b S
  → a ε b S             [First S → ε]
  → a b S
  → a b (a S b S)       [S → a S b S]
  → a b (a ε b S)       [Inner S → ε]
  → a b a b S
  → a b a b ε           [S → ε]
  → "abab" ⍰ (Not "aab")
```

Conclusion:

The string "aab" **cannot be generated** by the grammar S → aSbS **|** ε because:

1. Every 'a' must be followed by a corresponding 'b' in the derivation
2. The grammar maintains a balanced structure where 'a's and 'b's are paired
3. "aab" has 2 'a's and 1 'b', breaking the pairing pattern

Valid Parse Tree Example for "abab":

Since "aab" is not in the language, here's a parse tree for "abab":

text

```
       S
      /|\
     / | \
    a  S  b S
       |    \
      /|\    ε
     / | \
    a  S  b S
       |    \
       ε    /|\
           / | \
          a  S  b S
             |    \
             ε     ε
```

Derivation: S → aSbS → a(aSbS)bS → a(aεbε)b(aSbS) → a(aεbε)b(aεbε) = "abab"

Alternative Grammar for "aab":

If we wanted to generate strings like "aab", we could use:

text

```
S → A B
A → a A | a
B → b
```

This would generate strings with one or more 'a's followed by exactly one 'b'.

Key Insights:

1. Grammar design determines language characteristics
2. Not all strings belong to a given grammar's language

3. Parse trees only exist for valid strings in the language
4. Understanding grammar limitations is crucial for compiler design