|  |  |
| --- | --- |
| **https://lh7-us.googleusercontent.com/cqr1zywKLZ-KYtGHQsJs_4r0Pz65g7Hm9cYAB_QVFrqm6JK4FqjMzqYGew6RHuFzmfT56Wdn2C69ISfHmuDwuy_tnadpQXO2ujqRH_tBzkPaOHrK6awj4voQaDIwnQBQeNfh8u1bi15aREV24NpZmw** | **Compiler Construction**  **BSCS 5-A**  **Department of Computer Science**  **Bahria University, Lahore Campus** |

**Assignment: [3 & 4]**

Date: Week 2, March 2024

Name: \_AFFAN AHMAD\_\_\_\_

Roll No:  \_03-134221-003\_\_\_\_\_

|  |  |  |  |
| --- | --- | --- | --- |
| **Evaluation of CLO** | **Question Number** | **Marks** | **Obtained Marks** |
| **CLO3:  Design and implement a compiler using a software engineering approach** | 1 | 5 |  |
|  |  |  |
| **Total Marks** | | **5** |  |

**Q1: Implementing a Compiler for an Object-Oriented Programming Language with Fibonacci Sequence Generation**

**Create a simple object-oriented programming language, which we'll call "OOPL." OOPL should support the following features:**

1. Variables of type integer, float, and string
2. Basic arithmetic operations (+, -, \*, /) for integer and float types
3. String concatenation using the "+" operator
4. Conditional statements (if-else) with boolean expressions
5. Loops (while) with conditional expressions
6. Functions with parameters and return types
7. Classes with attributes and methods
8. Inheritance and polymorphism
9. A built-in function to generate the Fibonacci sequence up to a given number

**Task**

1. The OOPL language should have a built-in function fib(n) that generates the Fibonacci sequence up to the n-th number. For example, fib(5) should return the sequence 0, 1, 1, 2, 3.
2. Design a non-deterministic finite automaton (NFA) and a deterministic finite automaton (DFA) that can recognize the Fibonacci sequence generated by the fib(n) function. The NFA and DFA should be able to recognize the sequence of numbers in the Fibonacci sequence, regardless of the input n.
3. Implement a parsing technique suitable for compiling OOPL programs. You can choose from top-down parsing (e.g., recursive descent parsing) or bottom-up parsing (e.g., LR parsing). Justify your choice of parsing technique and explain how it handles the syntax of OOPL.

**Constraints**

1. The NFA and DFA should have a finite number of states (less than 20).
2. The parsing technique should be able to handle left recursion in the OOPL grammar.
3. The compiler should be able to generate assembly code for the target machine that is efficient in terms of execution time and memory usage.

**Deliverables**

1. A written report detailing the design and implementation of the compiler, including the lexical analyzer, parser, semantic analyzer, code generator, and optimizer.
2. The source code for the compiler, including any necessary libraries or tools.
3. A test suite of OOPL programs that demonstrate the functionality of the compiler, including programs that use the fib(n) function.
4. A detailed description of the NFA and DFA designs, including state transition diagrams and explanations of how they recognize the Fibonacci sequence.
5. A justification of the chosen parsing technique, including an explanation of how it handles the syntax of OOPL.

· Designing the OOPL Language:

· Define the syntax and semantics of the language, including grammar rules, data types, operators, control structures, functions, classes, and inheritance.

Specify the syntax for the fib(n) function and its behavior.

· **Lexical Analysis:**

· Implement a lexical analyzer (lexer) to tokenize the input source code into a sequence of tokens.

·  **Parsing Technique:**

· Choose a parsing technique suitable for OOPL. Since the language requirements include left recursion and complex syntax, a top-down parsing approach such as recursive descent parsing might be suitable. It allows for easy implementation of complex syntax rules and can handle left recursion effectively.

· **Syntax Analysis:**

· Implement the parser to recognize the syntactic structure of the input program according to the defined grammar rules.

Handle left recursion and construct a parse tree or abstract syntax tree (AST) representing the program's structure.

· **Semantic Analysis:**

· Perform semantic analysis to check for semantic correctness, including type checking, scoping rules, and resolving symbols.

Verify the correctness of function calls, expressions, and class declarations.

·  **Code Generation:**

· Translate the AST into intermediate representation (IR) or directly into assembly code targeting the desired machine architecture.

Implement efficient code generation techniques to produce optimized code for execution.

·  **Fibonacci Sequence Recognition:**

· Design a non-deterministic finite automaton (NFA) and a deterministic finite automaton (DFA) to recognize the Fibonacci sequence generated by the fib(n) function.

Create state transition diagrams for both NFA and DFA.

Implement algorithms to simulate the automata and recognize the sequence of numbers.

· **Testing and Optimization:**

· Develop a test suite to verify the functionality and correctness of the compiler.

Profile the generated code for performance and optimize it for execution time and memory usage.

Address any potential issues or bugs discovered during testing.

·  **Documentation:**

· Provide a written report detailing the design and implementation of the compiler, including explanations of each component and the rationale behind design decisions.

Include the source code of the compiler, test suite, and any necessary libraries.

Describe the NFA and DFA designs, including state transition diagrams and explanations of how they recognize the Fibonacci sequence.

Justify the choice of parsing technique and explain how it handles the syntax of OOPL effectively.

#include <iostream>

#include <string>

#include <vector>

using namespace std;

class Token {

public:

string type;

int value;

Token(string type, int value = 0) : type(type), value(value) {}

};

class Lexer {

public:

string text;

int pos;

Lexer(string text) : text(text), pos(0) {}

Token get\_next\_token() {

if (pos >= text.length()) {

return Token("EOF");

}

char current\_char = text[pos];

if (isdigit(current\_char)) {

pos++;

return Token("INTEGER", current\_char - '0');

} else if (current\_char == '+') {

pos++;

return Token("PLUS");

}

else {

throw runtime\_error("Invalid character");

}

}

};

class Parser {

public:

Lexer lexer;

Token current\_token;

Parser(Lexer lexer) : lexer(lexer), current\_token(lexer.get\_next\_token()) {}

int parse() {

return expr();

}

int expr() {

int result = term();

while (current\_token.type == "PLUS") {

current\_token = lexer.get\_next\_token();

result += term();

}

return result;

}

int term() {

Token token = current\_token;

if (token.type == "INTEGER") {

current\_token = lexer.get\_next\_token();

return token.value;

} else {

throw runtime\_error("Invalid syntax");

}

}

};

bool is\_fibonacci\_sequence(vector<int>& sequence) {

if (sequence.size() < 2) {

return false;

}

int a = sequence[0];

int b = sequence[1];

for (size\_t i = 2; i < sequence.size(); ++i) {

int next = a + b;

if (sequence[i] != next) {

return false;

}

a = b;

b = next;

}

return true;

}

int main() {

string text = "1+2+3";

Lexer lexer(text);

Parser parser(lexer);

int result = parser.parse();

cout << "Result: " << result << endl;

vector<int> fib\_sequence = {0, 1, 1, 2, 3};

cout << "Is Fibonacci sequence: " << boolalpha << is\_fibonacci\_sequence(fib\_sequence) << endl;

return 0;

}

Output:

