Compiler Design: Tokenization, Symbol Table, and Intermediate Code Generation

1. Introduction

This report documents the design and development of a simplified compiler system that includes three key components: **Tokenization**, **Symbol Table Management**, and **Intermediate Code Generation**. The system parses a given source code string, tokenizes it into various components, verifies variable declarations, performs semantic analysis, and generates intermediate code for an abstract representation of the program. The code handles constructs like variable declarations, assignments, conditional statements, and loops.

2. Lexer: Tokenization

The **Lexer** component is responsible for breaking down the source code into a sequence of tokens. Each token represents a distinct language construct (e.g., keywords, operators, identifiers, or numbers).

Features:

- The lexer scans the source code character by character.
- It handles different types of tokens: keywords (e.g., int, float, if), operators (e.g., =, +, -, *, /), and punctuation marks (e.g., ;, (,)).
- It skips comments (single-line comments starting with //).
- It maintains a line counter to track the location of tokens for error reporting.

Improvements:

- Enhanced token recognition by detecting keywords and identifiers separately.
- Added support for floating-point numbers.

Code:

cpp

Copy code

```
class Lexer
{
private:
   string src;
```

```
    size_t pos;
    int line;

public:
    Lexer(const string &src) : src(src), pos(0), line(0) {}

    vector<Token> tokenize()
    {
        vector<Token> tokens;
        while (pos < src.size())
        {
            char current = src[pos];

            if (current == '/' && pos + 1 < src.size() && src[pos + 1] == '/')
            {
                pos += 2;
                while (pos < src.size() && src[pos] != '\n')
                    pos++;
                continue;
            }

            if (current == '\n')
            {
                line++;
                pos++;
                continue;
```

```
}

if (isspace(current))
{
   pos++;
   continue;
}

if (isdigit(current))
{
   tokens.push_back(Token{T_NUM, consumeNumber(), line});
   continue;
}

if (isalpha(current))
{
   string word = consumeWord();
   if (word == "int")
      tokens.push_back(Token{T_INT, word, line});
   else if (word == "float")
      tokens.push_back(Token{T_FLOAT, word, line});
   else if (word == "if")
      tokens.push_back(Token{T_IF, word, line});
   else if (word == "else")
      tokens.push_back(Token{T_ELSE, word, line});
   else if (word == "for")
```

```
            tokens.push_back(Token{T_FOR, word, line});
        else
            tokens.push_back(Token{T_ID, word, line});
        continue;
    }

    switch (current)
    {
    case '=': tokens.push_back(Token{T_ASSIGN, "=", line}); break;
    case '+': tokens.push_back(Token{T_PLUS, "+", line}); break;
    case '-': tokens.push_back(Token{T_MINUS, "-", line}); break;
    case '*': tokens.push_back(Token{T_MUL, "*", line}); break;
    case '/': tokens.push_back(Token{T_DIV, "/", line}); break;
    case '(': tokens.push_back(Token{T_LPAREN, "(", line}); break;
    case ')': tokens.push_back(Token{T_RPAREN, ")", line}); break;
    case '{': tokens.push_back(Token{T_LBRACE, "{", line}); break;
    case '}': tokens.push_back(Token{T_RBRACE, "}", line}); break;
    case ';': tokens.push_back(Token{T_SEMICOLON, ";", line}); break;
    case '>': tokens.push_back(Token{T_GT, ">", line}); break;
    case '<': tokens.push_back(Token{T_LT, "<", line}); break;
    default:
        cout << "Unexpected character: " << current << endl;
        exit(1);
    }
    pos++;
}
```

```
    tokens.push_back(Token{T_EOF, "", line});
    return tokens;
  }
};
```

3. Symbol Table Management

The **Symbol Table** is responsible for keeping track of all variables (and their types) used in the program. It ensures that variable declarations are valid, that variables are used only after they are declared, and detects redeclarations.

Features:

- Stores variable information such as the variable's name, datatype, and initial value.
- Verifies that each identifier is declared before use and checks for redefinition.
- Includes functionality to add, check, and retrieve entries for variables.

Improvements:

- The symbol table now handles both primitive types (int, float) and ensures variables are properly declared and initialized before use.

Code:

```
cpp
Copy code
class SymbolTable
{
private:
  vector<Token> tokens;
  map<string, SymbolEntry> table;

public:
  SymbolTable(const vector<Token> &tokens) : tokens(tokens) {}
```

```
void addEntry(const string &name, TokenType datatype, const string &value = "")
{
   table[name] = SymbolEntry{name, datatype, value};
}

bool exists(const string &name) const
{
   return table.find(name) != table.end();
}

void makeTable()
{
   for (size_t i = 0; i < tokens.size(); i++)
   {
      const auto &token = tokens[i];
      if (token.type == T_INT || token.type == T_FLOAT)
      {
         if (i + 1 < tokens.size() && tokens[i + 1].type == T_ID)
         {
            const auto &varName = tokens[i + 1].value;
            if (!exists(varName))
               addEntry(varName, token.type);
            else
            {
               cout << "Redefinition of variable: " << varName << endl;
               exit(1);
```

```
            }
        }
    }
    else if (token.type == T_ID)
    {
        if (!exists(token.value))
        {
            cout << "Undeclared variable: " << token.value << endl;
            exit(1);
        }
    }
  }
};
```

4. Intermediate Code Generation

The **Intermediate Code Generator** generates an intermediate representation of the program using temporary variables and labels. This allows for a more manageable and platform-independent representation of the program's logic.

Features:

- The intermediate code includes assignments, arithmetic operations, and control flow operations (e.g., conditionals, loops).
- Temporary variables (T0, T1, etc.) are used for intermediate values.
- Labels (L0, L1, etc.) are used to manage control flow during execution.

Improvements:

- The code generation supports operations for arithmetic expressions, conditional statements, and for loop constructs.

- Uses temporary variables for intermediate results and generates labels for if and for loops.

Code:

cpp

Copy code

```
class IntermediateCodeGenerator
{
private:
  vector<string> instructions;
  int tempCounter = 0;
  int labelCounter = 0;

public:
  string newTemp() { return "T" + to_string(tempCounter++); }
  string newLabel() { return "L" + to_string(labelCounter++); }

  void addInstruction(const string &instr)
  {
    instructions.push_back(instr);
  }

  void printInstructions() const
  {
    for (const auto &instr : instructions)
      cout << instr << endl;
  }
```

};

5. Parser

The **Parser** component interprets the sequence of tokens produced by the lexer and ensures the program's syntax is correct. It also generates intermediate code as it parses the source.

Features:

- Supports constructs such as variable declarations, assignments, if statements, and for loops.
- Handles expressions and generates intermediate code for arithmetic operations and conditions.
- Uses helper methods for parsing different parts of the syntax (e.g., parseExpression(), parseAssignment()).

Improvements:

- The parser now supports nested blocks, handling both if statements with else and for loops.

Code:

cpp

Copy code

```
class Parser
{
private:
  vector<Token> tokens;
  size_t pos = 0;
  IntermediateCodeGenerator &icg;
  SymbolTable &symbolTable;

public:
```

```
  Parser(const vector<Token> &tokens, IntermediateCodeGenerator &icg, SymbolTable &symbolTable)
      : tokens(tokens), icg(icg), symbolTable(symbolTable) {}


  void parseProgram()
  {
    while (tokens[pos].type != T_EOF)
    {
      parseStatement();
    }
  }


private:
  void parseStatement()
  {
    if (tokens[pos].type == T_INT || tokens[pos].type == T_FLOAT)
    {
      parseDeclaration();
    }
    else if (tokens[pos].type == T_ID)
    {
      parseAssignment();
    }
    else if (tokens[pos].type == T_IF)
    {
      parseIfStatement();
```

```
    }
    else if (tokens[pos].type == T_LBRACE)
    {
      parseBlock();
    }
    else if (tokens[pos].type == T_FOR)
    {
      parseForStatement();
    }
    else
    {
      cout << "Syntax error: unexpected token " << tokens[pos].value << " at line: " << tokens[pos].line << endl;
      exit(1);
    }
  }

  // Additional methods for parsing specific constructs (e.g., blocks, assignments, expressions)
};
```

6. Conclusion

This project demonstrates the essential stages of compiling a simple program: tokenization, symbol table management, and intermediate code generation. It also showcases how to handle basic programming constructs like variable declarations, assignments, conditional statements, and loops. Through careful design and handling of tokens and symbols, this compiler system can be extended to support more complex features such as functions and more advanced expressions.

The improvements in the lexer, parser, and code generator ensure a smooth experience with efficient error handling and modular design for easy extension and modification.