Lab 6

Session: 2021 – 2024

Submitted by:

Uswa Arif

2021-CS-77

Submitted To:

Laeeq Khan Niazi

Department of Computer Science

University of Engineering and Technology

Lahore Pakistan

Code:

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

using namespace std;

enum TokenType
{
    T_INT, T_ID, T_NUM, T_IF, T_ELSE, T_RETURN,
    T_ASSIGN, T_PLUS, T_MINUS, T_MUL, T_DIV,
    T_LPAREN, T_RPAREN, T_LBRACE, T_RBRACE,
    T_SEMICOLON, T_GT, T_EOF,
    T_FOR, T_WHILE, T_EQ, T_LE, T_AND, T_FLOAT, T_STRING
};

struct Token
{
    TokenType type;
    string value;
    int lineNumber;
    int columnNumber;
    string dataType;

    Token(TokenType t, const string& val, int ln, int col, const string& dt = "")
        : type(t), value(val), lineNumber(ln), columnNumber(col), dataType(dt) {}
};

class Lexer{
    private:
        string src;   //code
        size_t pos;   //position of pointer
        int line;
        int column;

    public:
```

```
30  class Lexer{
38          Lexer(const string &src)
39          {
40              this->src = src;
41              this->pos = 0;
42              this->line = 1;
43              this->column = 1;
44          }
45
46          string consumeNumber()
47          {
48              size_t start = pos;
49              while(pos < src.size() && isdigit(src[pos]))
50              {
51                  pos++;
52              }
53              return src.substr(start, pos - start);
54          }
55
56          string consumeWord()
57          {
58              size_t start = pos;
59              while(pos < src.size() && isalnum(src[pos]))
60              {
61                  pos++;
62              }
63              return src.substr(start, pos - start);
64          }
65
66          vector<Token> tokenize()
67          {
68              vector<Token> tokens;
69              while (pos < src.size())
70              {
71                  char current = src[pos];
72                  if (isspace(current))
73                  {
```

```
            if (current == '\n') {
                line++;
                column = 1;
            } else {
                column++;
            }
            pos++;
            continue;
        }

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

        if (isalpha(current))
        {
            string word = consumeWord();
            if (word == "int") tokens.push_back(Token{T_INT, word, line, column});
            else if (word == "if") tokens.push_back(Token{T_IF, word, line, column});
            else if (word == "else") tokens.push_back(Token{T_ELSE, word, line, column});
            else if (word == "return") tokens.push_back(Token{T_RETURN, word, line, column});
            else if (word == "for") tokens.push_back(Token{T_FOR, word, line, column});
            else if (word == "while") tokens.push_back(Token{T_WHILE, word, line, column});
            else tokens.push_back(Token{T_ID, word, line, column});
            continue;
        }

        if (current == '=' && pos + 1 < src.size() && src[pos + 1] == '=')
        {
            tokens.push_back(Token{T_EQ, "==", line, column});
            pos += 2;
            column += 2;
            continue;
        }
```

```
class Lexer{
    vector<Token> tokenize()
        if (current == '&' && pos + 1 < src.size() && src[pos + 1] == '&')
        {
            tokens.push_back(Token{T_AND, "&&", line, column});
            pos += 2;
            column += 2;
            continue;
        }

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

class Parser
{
public:
    Parser(const vector<Token> &tokens)
```

```
141  class Parser
144          Parser(const vector<Token> &tokens)
145          {
146              this->tokens = tokens;
147              this->pos = 0;
148          }
149
150          void parseProgram() {
151          while (tokens[pos].type != T_EOF)
152              {
153                  parseStatement();
154              }
155              cout << "Parsing completed successfully! No Syntax Error" << endl;
156          }
157
158      private:
159          vector<Token> tokens;
160          size_t pos;
161
162          void parseStatement()
163          {
164              if (tokens[pos].type == T_INT) {
165                  parseDeclaration();
166              } else if (tokens[pos].type == T_ID) {
167                  parseAssignment();
168              } else if (tokens[pos].type == T_IF) {
169                  parseIfStatement();
170              } else if (tokens[pos].type == T_RETURN) {
171                  parseReturnStatement();
172              } else if (tokens[pos].type == T_LBRACE) {
173                  parseBlock();
174              } else {
175                  cout << "Syntax error: unexpected token " << tokens[pos].value << endl;
176                  exit(1);
177              }
178          }
179
```

```
180         void parseBlock()
181         {
182             expect(T_LBRACE);
183             while (tokens[pos].type != T_RBRACE && tokens[pos].type != T_EOF) {
184                 parseStatement();
185             }
186             expect(T_RBRACE);
187         }
188         void parseDeclaration() {
189             expect(T_INT);
190             expect(T_ID);
191             expect(T_SEMICOLON);
192         }
193 
194         void parseAssignment() {
195             expect(T_ID);
196             expect(T_ASSIGN);
197             parseExpression();
198             expect(T_SEMICOLON);
199         }
200 
201         void parseIfStatement() {
202             expect(T_IF);
203             expect(T_LPAREN);
204             parseExpression();
205             expect(T_RPAREN);
206             parseStatement();
207             if (tokens[pos].type == T_ELSE) {
208                 expect(T_ELSE);
209                 parseStatement();
210             }
211         }
212 
213         void parseReturnStatement() {
214             expect(T_RETURN);
215             parseExpression();
```

```
213             void parseReturnStatement() {
216                 expect(T_SEMICOLON);
217             }
218
219             void parseExpression() {
220                 parseTerm();
221                 while (tokens[pos].type == T_PLUS || tokens[pos].type == T_MINUS) {
222                     pos++;
223                     parseTerm();
224                 }
225                 if (tokens[pos].type == T_GT) {
226                     pos++;
227                     parseExpression();  // After relational operator, parse the next expression
228                 }
229             }
230
231             void parseTerm() {
232                 parseFactor();
233                 while (tokens[pos].type == T_MUL || tokens[pos].type == T_DIV) {
234                     pos++;
235                     parseFactor();
236                 }
237             }
238
239             void parseFactor() {
240                 if (tokens[pos].type == T_NUM || tokens[pos].type == T_ID) {
241                     pos++;
242                 } else if (tokens[pos].type == T_LPAREN) {
243                     expect(T_LPAREN);
244                     parseExpression();
245                     expect(T_RPAREN);
246                 } else {
247                     cout << "Syntax error: unexpected token " << tokens[pos].value << endl;
248                     exit(1);
249                 }
250             }
252         void expect(TokenType type) {
253             if (tokens[pos].type == type) {
254                 pos++;
255             } else {
256                 cout << "Syntax error: expected " << type << " but found " << tokens[pos].value << endl;
257                 exit(1);
258             }
259         }
260
261 };
262
263 int main() {
264     string input = R"(
265         int a;
266         a = 5;
267         int b;
268         b = a + 10;
269         if (b > 10) {
270             return b;
271         } else {
272             return 0;
273         }
274     )";
275
276     Lexer lexer(input);
277     vector<Token> tokens = lexer.tokenize();
278
279     Parser parser(tokens);
280     parser.parseProgram();
281
282     return 0;
283 }
```

Output:

```
D:\7 semester\CC Lab\Lab6>compiler.exe
Parsing completed successfully! No Syntax Error
```