گزارشکار پروژه نهایی کامپایلر

معصومه طهماسبی

در این پروژه قصد داریم بخش Frontend یک کامپایلر را پیاده سازی کنیم که گرامر زیر را بپذیرد:

```
<S> => Program <VARS> <BLOCKS> end
<VARS> => Var Identifier; <VARS> | Epsilon
<BLOCKS> => Start <STATES> End
<STATES> => <STATE> <M_STATES>
STATE> => <BLOCKS> | <IF> | <IN> | <OUT> | <ASSIGN> | <LOOP>
<M_STATES> => <STATES> | Epsilon
<OUT>=> Print (<EXPR>);
<IN> => Read ( Identifier ) ;
<IF> => If ( <EXPR> <O> <EXPR> ) { <STATE> }
<LOOP> => Iteration ( <EXPR> <O> <EXPR> ) { <STATE> }
<ASSIGN> => Put Identifier = <EXPR> ;
<0> => < | > | ==
<EXPR> => <EXPR> + <R> | <EXPR> - <R> | <R>
<R> => Identifier | Integer
```

باید ابتدا تحلیلگر لغوی را پیاده‌سازی نماییم تا کد را از ورودی گرفته و لیستی از توکن‌ها را تحویل دهد. سپس تحلیلگر نحوی را پیاده‌سازی می‌کنیم که براساس گرامر فوق، توکن‌ها را بررسی کرده و در صورت صحیح نبودن سینتکس کد، ارور می‌دهد.

توضیحات فایل Token.hpp

در این فایل، یک کلاس به نام Keywords تعریف کرده‌ایم که کلمات کلیدی را در قالب یک لیست ثابت، نگهداری می‌کند. هدف اصلی این فایل این است که دسترسی راحت‌تری را برای Scanner و Parser فراهم کند. با استفاده از این header، می‌توانیم برای ورودی‌های مختلف یک label تعریف کنیم.

```
enum class TokenType {
    IDENTIFIER,
    KEYWORD,
    NUMBER,
    OPERATOR,
    DELIMITER,
    INVALID
};
```

ابتدا یک کلاس enum برای دسته‌بندی انواع توکن‌ها ایجاد می‌کنیم. هر مقدار این کلاس، نشان‌دهنده نوع خاصی از توکن است که توسط اسکنر (Lexer) تولید میشود. جزئیات مقدارها به شرح زیر است:

- IDENTIFIER: شناسه‌های تعریف شده توسط کاربر (مانند نام متغیرها، توابع و کلاس‌ها).
- KEYWORD: کلمات رزرو شده زبان – دستورات یا ساختارهای اصلی.
- NUMBER: اعداد صحیح یا اعشاری.
- OPERATOR: عملگرهای محاسباتی یا مقایسه‌ای.
- DELIMITER: نویسه‌های جداکننده یا گروهبندی‌کننده بلوک‌ها (مانند ; و براکت)
- INVALID: توکن‌های نامعتبر یا ناشناخته (خطاهای لغوی)

```
class Token {
public:
    TokenType type;
    std::string value;
    int line;
    Token(TokenType type, std::string  value, int line)
        : type(type), value(std::move(value)), line(line) {}
    [[nodiscard]] std::string getTypeAsString() const {
        switch (type) {
            case TokenType::IDENTIFIER: return "IDENTIFIER";
            case TokenType::KEYWORD: return "KEYWORD";
            case TokenType::NUMBER: return "NUMBER";
            case TokenType::OPERATOR: return "OPERATOR";
            case TokenType::DELIMITER: return "DELIMITER";
            default: return "UNKNOWN";
        }
    }
};
```

سپس کلاس توکن را تعریف می‌کنیم. هر توکن سه مقدار type، value و line دارد.

- type یک شی از کلاس TokenType است و نوع توکن را مشخص می‌کند که می‌تواند یکی از شش مورد گفته شده باشد.
- value از نوع string است و صرفا Lexeme یا متن واقعی کلمه را نگهداری می‌کند.
- Line از نوع integer است و محل دقیق توکن در کد منبع را نشان می‌دهد. بعدا برای خطایابی می‌توان از آن استفاده کرد.

از یک constructor استفاده کرده‌ایم تا توکن را مقداردهی اولیه کنیم.

تابع ()getTypeAsString برای توکن تعریف شده است تا نوع توکن را به صورت رشته برگرداند. در هنگام تحلیل لغوی، با استفاده از این تابع می‌توانیم به سادگی به نوع توکن دسترسی داشته باشیم.

توضیحات فایل Keywords.hpp

این فایل کلمات کلیدی را در کلاس Keywords نگهداری می‌کند تا در بخش‌های دیگر، دسترسی سریع و یکپارچه‌ای داشته باشیم. در Scanner می‌توانیم کلمات ورودی را با استفاده از این header چک کنیم و بین identifier و keyword ها تفاوت قائل شویم. همچنین این طراحی باعث کاهش خطا، آسان شدن نگهداری و بهبود خوانایی کد می‌شود.

```
class Keywords {
public:
    static const std::vector<std::string>& getKeywords() {
        static std::vector<std::string> keywords = {
            "Program", "End", "Start", "end", "Print",
            "Read", "If", "Iteration", "Put", "Var"
        };
        return keywords;
    }
};
```

در این کلاس تابع getKeywords را ایجاد کرده‌ایم که برای مدیریت متمرکز و کارآمد کلمات کلیدی استفاده می‌شود. هرگاه بخواهیم بررسی کنیم که کلمه مورد نظر، Keyword است یا خیر، می‌توانیم از این تابع استفاده کنیم.

توضیحات فایل DFA.hpp

یکی از راه‌های بهینه برای تحلیل لغوی، استفاده از ماشین حالت متناهی (DFA یا NFA) و کتابخانه regex است. این دو به ما کمک می‌کنند که کدهای بیشتر را با سربار کمتر بررسی کنیم. استفاده از DFA بهتر است زیرا در NFA قابلیت Backtrack داریم و سربار بیشتری دارد.

```
enum class State {
    START,
    IN_KEYWORD,
    IN_NUMBER,
    IN_OPERATOR,
    IN_DELIMITER,
    DONE
};
```

ابتدا یک کلاس تعریف کردیم تا حالت‌های مختلف DFA را در خود نگه دارد.

```
std::regex operatorRegex;
std::regex delimiterRegex;

void initializeRegexPatterns() {
operatorRegex = std::regex("^[+\\-*/=<>!&|^%]+$");
delimiterRegex = std::regex(R"([(){}[\];,])");
}
```

در کلاس DFA، یک تابع برای مقداردهی اولیه به الگوهای regex تعریف کرده‌ایم. بعدا از این الگوها برای بررسی ورودی‌ها و تغییر حالت استفاده می‌کنیم. الگوهای تعریف شده در خط فوق، برای مشخص کردن عملگرها و delimiterها هستند.

```
static bool isRegexMatch(const std::string& value, const std::regex& pattern) {
        return std::regex_match(value, pattern);
}
```

در تابع بالا، می‌توانیم بررسی کنیم که آیا رشته مد نظر، در الگوی regex صدق می‌کند یا نه. توسط این تابع، می‌توانیم الگوهای مختلف را برای یک رشته بررسی کنیم. چون برای operator و delimiter دو الگوی متفاوتی داریم، با استفاده از تابع بالا از تکرار کدها جلوگیری می‌کنیم.

```
static bool isIdentifierStart(char ch) {
        return std::isalpha(ch) || ch == '_';
}
static bool isIdentifierBody(char ch) {
        return std::isalnum(ch) || ch == '_';
}
static bool isOperatorChar(char ch) {
        static const std::string operators = "+-*/=<>!&|^%";
        return operators.find(ch) != std::string::npos;
}
bool isDelimiterChar(char ch) {
        return std::regex_match(std::string(1, ch), delimiterRegex);
}
```

از توابع بالا استفاده می‌کنیم تا نوع رشته ورودی را بفهمیم و بتوانیم تغییر حالت مناسب را انجام دهیم. در dfa براساس ورودی باید به حالت‌های دیگر برویم و این توابع مشخص می‌کنند که چه ورودی‌ای دیده‌ایم.

تابع transition مشخص می‌کند که در هر حالت، براساس هر ورودی، باید به چه حالت دیگری برویم. این تابع با بررسی کاراکتر ورودی ch و حالت فعلی currentState، تصمیم می‌گیرد که به کدام حالت جدید منتقل شود.

```
case State::START:
    if (isOperatorChar(ch)) {
        currentState = State::IN_OPERATOR;
    } else if (std::isdigit(ch)) {
        currentState = State::IN_NUMBER;
    } else if (isIdentifierStart(ch)) {
        currentState = State::IN_KEYWORD;
    } else if (isDelimiterChar(ch)) {
        currentState = State::IN_DELIMITER;
    } else {
        currentState = State::START;
    }
    break;
```

حالت START

در این حالت، DFA در ابتدای پردازش یک توکن جدید قرار دارد و هنوز هیچ کاراکتری برای توکن فعلی پردازش نشده است. براساس نوع توکن ورودی، به حالت‌های مختلف منتقل می‌شود.

```
case State::IN_KEYWORD:
    if (isIdentifierBody(ch)) {
        currentState = State::IN_KEYWORD;
    } else {
        currentState = State::DONE;
    }
    break;
```

اگر در حالت IN_KEYWORD باشیم، یعنی یک کاراکتر alphabet یا "_" مشاده کرده‌ایم. تا زمانی که کاراکتر ورودی از نوع حروف الفبای انگلیسی یا "_" است، در این حالت می‌مانیم و در غیر این صورت، به حالت پایانی می‌رویم.

```
case State::IN_NUMBER:
    if (std::isdigit(ch)) {
        currentState = State::IN_NUMBER;
    } else {
        currentState = State::DONE;
    }
    Break;
```

اگر در حالت IN_NUMBER یعنی یک رقم (عدد) شناسایی کرده‌ایم. تا زمانی که کاراکترهای ورودی از نوع ارقام (۰-۹) باشند، در این حالت باقی می‌مانیم تا اعداد پشت‌سرهم را به عنوان یک توکن عددی کامل پردازش کنیم. به محض مواجهه با هر کاراکتر غیررقمی (مثل حروف، عملگرها یا جداکننده‌ها)، به حالت DONE تغییر وضعیت می‌دهیم. این انتقال نشان می‌دهد که عدد کامل شده و باید توکن مربوط به آن تولید شود.

```
case State::IN_OPERATOR:
    tempOperator += ch;
    if (isRegexMatch(tempOperator, operatorRegex)) {
        currentState = State::IN_OPERATOR;
    } else {
        currentState = State::DONE;
    }
    break;
```

در حالت IN_OPERATOR، ما یک عملگر (مانند +, -, == و ...) را پردازش می‌کنیم. هر بار که یک کاراکتر جدید به tempOperator اضافه می‌شود، بررسی می‌شود آیا رشته‌ی جمع‌آوری‌شده (tempOperator) با الگوی regex تعریف‌شده برای عملگرها (operatorRegex) مطابقت دارد یا نه.

اگر تطابق وجود داشته باشد، در حالت IN_OPERATOR باقی می‌مانیم تا امکان تشکیل عملگرهای چندکاراکتری (مانند ++, +=, && و ...) فراهم شود.

اگر تطابق وجود نداشته باشد، به حالت DONE تغییر وضعیت می‌دهیم. این نشان می‌دهد عملگر کامل شده و باید توکن مربوط به آن تولید شود.

این منطق اجازه می‌دهد عملگرهای چندکاراکتری به صورت پویا شناسایی شوند. اگر operatorRegex شامل عملگرهای ترکیبی باشد (مثلاً == یا !=)، تا زمانی که رشته‌ی ساخته‌شده بخشی از یک عملگر معتبر است، پردازش ادامه می‌یابد.

```
case State::IN_DELIMITER:
    currentState = State::DONE;
    break;
```

در حالت IN_DELIMITER، یک جداکننده شناسایی شده است. برخلاف عملگرها یا اعداد که ممکن است چندکاراکتری باشند، جداکننده‌ها همیشه تک‌کاراکتری هستند. بنابراین، به محض ورود به این حالت، بلافاصله به حالت `DONE` تغییر وضعیت می‌دهیم. این نشان می‌دهد توکن مربوط به جداکننده کامل شده و نیازی به بررسی کاراکترهای بعدی نیست.

```
default:
    currentState = State::START;
    break;
```

این بخش به عنوان مکانیزم ایمنی عمل می‌کند و هرگاه وضعیتی غیرمنتظره یا نامعتبر (خارج از حالت‌های تعریف‌شده) شناسایی شود، بلافاصله وضعیت پردازش را به حالت شروع (START) برمی‌گرداند. این کار از گیر کردن تحلیلگر در حالت‌های ناشناخته یا ناممکن جلوگیری می‌کند و پردازش را با یک ریست ایمن ادامه می‌دهد.

```
void reset() {
        currentState = State::START;
        tempOperator.clear();
    }
```

این تابع دو عملیات اصلی انجام می‌دهد:

۱. بازنشانی حالت فعلی (currentState)

- وضعیت پردازش را به State::START تغییر می‌دهد. این یعنی تحلیلگر از ابتدای چرخه پردازش (حالت شروع) کار خود را برای توکن بعدی آغاز می‌کند.

۲. پاک‌سازی tempOperator

محتوای ذخیره‌شده در tempOperator (رشته‌ای که برای جمع‌آوری عملگرهای چندکاراکتری استفاده می‌شود) را حذف می‌کند تا آماده دریافت عملگرهای جدید باشد.

```
DFA() : currentState(State::START) {
    initializeRegexPatterns();
}
```

این تابع وضعیت اولیه تحلیلگر را روی `START` تنظیم میکند، الگوهای regex را برای شناسایی عملگرها و جداکننده‌ها آماده میکند، و شی DFA را برای شروع پردازش توکنها راه اندازی می‌کند.

جدول تغییر حالت DFA:

حالت فعلی	کاراکتر ورودی	حالت بعدی
Start	isOperatorChar(ch)	IN_OPERATOR
Start	std::isdigit(ch)	IN_NUMBER
Start	isIdentifierStart(ch)	IN_KEYWORD
Start	isDelimiterChar(ch)	IN_DELIMITER
Start	Others	START
IN_KEYWORD	isIdentifierBody(ch)	IN_KEYWORD
IN_KEYWORD	Others	DONE
IN_NUMBER	isRegexMatch(tempOperator, operatorRegex)	IN_OPERATOR
IN_NUMBER	Others	DONE
IN_DELIMITER	All	DONE
DONE	-	Start

توضیحات فایل Scanner.hpp

این کد یک اسکنر (Scanner) برای پردازش متن ورودی و استخراج توکنها در یک کامپایلر یا مفسر است. اسکنر نقش مهمی در مرحله تحلیل لغوی (Lexical Analysis) دارد و متن خام را به دنباله‌ای از توکن‌ها تبدیل میکند. کلاس Scanner از یک ماشین حالت متناهی قطعی (DFA) برای تشخیص نوع توکن‌ها استفاده می‌کند.

```
std::vector<Token> scan(const std::string& line, int& lineNumber) {
    std::vector<Token> tokens;
    std::string tokenValue;
    dfa.reset();
    for (char ch : line) {
        if (std::isspace(ch)) {
            if (!tokenValue.empty()) {
                processAndAddToken(tokenValue, lineNumber, tokens);
                tokenValue.clear();
            }
            dfa.reset();
            continue;
        }
        if (dfa.isDelimiterChar(ch)) {
            if (!tokenValue.empty()) {
                processAndAddToken(tokenValue, lineNumber, tokens);
                tokenValue.clear();
            }
            tokens.emplace_back(TokenType::DELIMITER,std::string(1, ch), lineNumber);
            dfa.reset();
            continue;
        }
        dfa.transition(ch);
        tokenValue += ch;
        if (dfa.getCurrentState() == State::DONE) {
            processAndAddToken(tokenValue, lineNumber, tokens);
            tokenValue.clear();
            dfa.reset();
        }
    }
```

متد scan قلب اسکنر است که یک خط ورودی (رشته) را دریافت کرده و توکن‌های آن خط را برمیگرداند. این متد با حلقه روی هر کاراکتر خط ورودی، چهار حالت اصلی را بررسی میکند: فاصله خالی (space)، نویسه‌های جداکننده (delimiter)، انتقال در DFA و حالت تکمیل توکن (DONE).

برای هر کاراکتر، ابتدا بررسی میشود اگر فضای خالی باشد و توکنی در حال پردازش وجود داشته باشد، آن را نهایی میکند. سپس اگر کاراکتر یک delimiter باشد (مانند ; یا ,)، توکن فعلی (در صورت وجود) و خود delimiter را به عنوان توکن جدید اضافه میکند. در غیر این صورت، DFA را بهروزرسانی کرده و کاراکتر را به مقدار توکن فعلی اضافه میکند. وقتی DFA به حالت DONE میرسد، نشان‌دهنده تکمیل یک توکن معتبر است.

```
void processAndAddToken(std::string& tokenValue, int line, std::vector<Token>&
tokens) {
    TokenType type = identifyTokenType(tokenValue);


    // Check identifier length
    if (type == TokenType::IDENTIFIER) {
        if (tokenValue.length() > 5) {
            std::cerr << "Lexical Error: Identifier '" << tokenValue << "' exceeds 5
characters at line " << line << std::endl;
            return;
        }
    }


    tokens.emplace_back(type, tokenValue, line);
}
```

تابع processAndAddToken مسئول تعیین نوع توکن و اعمال قوانین زبانی است. به طور خاص، اگر توکن یک شناسه (IDENTIFIER) باشد، طول آن بررسی میشود و در صورت بیش از ۵ کاراکتر بودن، خطای لغوی چاپ می‌شود.

```
TokenType identifyTokenType(const std::string& value) {
    switch (dfa.getCurrentState()) {
        case State::IN_KEYWORD:
            if (isKeyword(value))
                return TokenType::KEYWORD;
            return TokenType::IDENTIFIER;
        case State::IN_NUMBER: return TokenType::NUMBER;
        case State::IN_OPERATOR: return TokenType::OPERATOR;
        case State::IN_DELIMITER: return TokenType::DELIMITER;
```

```
        default: return TokenType::IDENTIFIER;
    }
}
```

تابع identifyTokenType با بررسی وضعیت فعلی DEA، نوع توکن (کلمه کلیدی، عدد، عملگر و...) را تشخیص میدهد. برای مثال، اگر DFA در حالت IN_KEYWORD باشد، ابتدا بررسی میکند آیا رشته در لیست کلمات کلیدی وجود دارد یا خیر. اگر وجود داشته باشد، آن را به عنوان Keyword در نظر می‌گیرد، درغیراین‌صورت، آن را به عنوان Identifier برمی‌گرداند.

```
static bool isKeyword(const std::string& value) {
    return std::find(Keywords::getKeywords().begin(), Keywords::getKeywords().end(),
value) != Keywords::getKeywords().end();
}
```

کلاس Scanner از یک شی DFA برای مدیریت حالتهای پردازش استفاده میکند. DFA با هر کاراکتر به روزرسانی می‌شود و وضعیت فعلی را دنبال می‌کند. تشخیص نهایی نوع توکن‌ها با ترکیب وضعیت DFA و لیست از پیش تعریف شده کلمات کلیدی انجام میشود. تابع isKeyword با جستجو در لیست Keywords::getKeywords، ماهیت کلمه کلیدی را تأیید میکند.

ین اسکنر با ترکیب الگوریتم‌های مبتنی بر DFA، مدیریت دستی کاراکترهای خاص و اعمال قوانین ، یک سیستم انعطافپذیر برای تحلیل لغوی ایجاد کرده است. رویکرد شی‌گرا با جداکردن مسئولیت‌ها بین کلاسهای Scanner، DFA و Keywords، قابلیت نگهداری و توسعه‌پذیری کد را افزایش میدهد. خروجی نهایی یک لیست از توکن‌ها است که برای مراحل بعدی کامپایلر (پارسر نحوی) آماده می‌شود.

توضیحات فایل Parser.hpp

این فایل یک پارسر نحوی برای بررسی صحت دستورات برنامه‌نویسی براساس گرامر داده شده است. پس از مرحله‌ی شناسایی توکن‌ها (توسط اسکنر)، چک میکند آیا این توکن‌ها طبق قواعد گرامر چیده شده‌اند یا نه. پارسر هر دستور پیچیده (مثل حلقه یا شرط) را به اجزای ساده‌تر میشکند و هر جزء را با توابع جداگانه بررسی میکند. اگر ساختار برنامه اشتباه باشد (مثلاً کلیدواژه‌ی اصلی جا افتاده باشد)، خطای واضحی با جزئیات موقعیت خطا نمایش میدهد. این مرحله برای اطمینان از درستی ساختار برنامه قبل از تبدیل به کد ماشین یا اجرا ضروری است.

```
enum class NonTerminal { S, VARS, BLOCKS, STATES, STATE, IF, IN, OUT, ASSIGN, LOOP,
EXPR, R, O };
```

این enum نمادهای غیرپایانی (Non-Terminal) گرامر را تعریف می‌کند. هر نماد مانند S (شروع برنامه)، VARS (متغیرها)، BLOCKS (بلوک‌های کد)، IF (دستور شرطی) و ... نشان‌دهنده‌ی یک قاعده در گرامر است.

```
enum class Terminal { PROGRAM, VAR, START, END, IF, ITERATION, PRINT, READ, PUT,
IDENTIFIER, NUMBER, OPERATOR, DELIMITER, END_OF_FILE };
```

این enum نمادهای پایانی (Terminal) را شامل می‌شود که متناظر با توکن‌های شناسایی شده توسط اسکنر هستند. مانند PROGRAM، VAR، IDENTIFIER، OPERATOR و ... .

```
inline Terminal getTerminal(const Token& token) {
    if (token.value == "Program") return Terminal::PROGRAM;
    if (token.value == "Var") return Terminal::VAR;
    if (token.value == "Start") return Terminal::START;
    if (token.value == "End") return Terminal::END;
    if (token.value == "If") return Terminal::IF;
    if (token.value == "Iteration") return Terminal::ITERATION;
    if (token.value == "Print") return Terminal::PRINT;
    if (token.value == "Read") return Terminal::READ;
    if (token.value == "Put") return Terminal::PUT;
    if (token.type == TokenType::IDENTIFIER) return Terminal::IDENTIFIER;
    if (token.type == TokenType::NUMBER) return Terminal::NUMBER;
    if (token.type == TokenType::OPERATOR) return Terminal::OPERATOR;
    if (token.type == TokenType::DELIMITER) return Terminal::DELIMITER;
    return Terminal::END_OF_FILE;
}
```

تابع getTerminal، یک Token را به مقدار متناظرش در Terminal تبدیل می‌کند. با بررسی value و type توکن ورودی، مقدار مناسب Terminal را برمی‌گرداند. مثلاً اگر توکن از نوع IDENTIFIER باشد، Terminal::IDENTIFIER برگردانده می‌شود.

کلاس Parser دارای تعدادی method است که در ادامه توضیح داده شده‌اند:

```
explicit Parser(const std::vector<Token>& tokens) : tokens(tokens),
currentTokenIndex(0) {}
```

سازنده explicit Parser لیستی از توکن‌ها را دریافت می‌کند و موقعیت فعلی توکن (currentTokenIndex) را مقداردهی اولیه می‌کند.

```
bool Parse() {
        parseStack.push(NonTerminal::S);
        return parseS();
    }
```

تابع Parse فرایند تجزیه را آغاز می‌کند. ابتدا نماد S (شروع گرامر) را به پشته اضافه می‌کند و متد parseS را فراخوانی می‌کند. نتیجه نهایی تحلیل (true/false) را برمی‌گرداند.

```
[[nodiscard]] Token currentToken() const {
        return tokens[currentTokenIndex];
}
```

تابع currentToken توکن فعلی را برمی‌گرداند.

```
void advance() {
        if (currentTokenIndex < tokens.size() - 1) currentTokenIndex++;
}
```

تابع advance نشانگر توکن فعلی را به توکن بعدی منتقل می‌کند.

```
bool match(Terminal expected) {
        if (getTerminal(currentToken()) == expected) {
            advance();
            return true;
        } else {
            std::cerr << "Syntax Error: Expected " << getTerminalName(expected) << "
but got " << currentToken().value << " at line " << currentToken().line << std::endl;
            return false;
```

تابع match بررسی می‌کند آیا توکن فعلی با Terminal مورد انتظار مطابقت دارد یا خیر. در صورت تطابق، به توکن بعدی می‌رود. در غیر این صورت، خطای نحوی با جزئیات موقعیت توکن چاپ می‌کند.

```
static std::string getTerminalName(Terminal terminal) {
    switch (terminal) {
        case Terminal::PROGRAM: return "Program";
        case Terminal::VAR: return "Var";
        case Terminal::START: return "Start";
        case Terminal::END: return "End";
        case Terminal::IF: return "If";
        case Terminal::ITERATION: return "Iteration";
        case Terminal::PRINT: return "Print";
        case Terminal::READ: return "Read";
        case Terminal::PUT: return "Put";
        case Terminal::IDENTIFIER: return "Identifier";
        case Terminal::NUMBER: return "Number";
        case Terminal::OPERATOR: return "Operator";
        case Terminal::DELIMITER: return "Delimiter";
        case Terminal::END_OF_FILE: return "End of File";
        default: return "Unknown";
    }
}
```

تابع getTerminalName، نام قابل خواندن Terminal را به صورت string برمی‌گرداند (برای نمایش خطا).

توابعی که نام آنها ترکیبی از Parse و اسم متغیرهاست، قوانین گرامر را به صورت کد اجرا می‌کنند. برای ترمینال‌ها از تابع match استفاده می‌شود و برای غیرترمینال‌ها، خود تابع parse آنها فراخوانی می‌شود.

توضیحات فایل main.cpp

این فایل به عنوان نقطه شروع برنامه عمل می‌کند و دو وظیفه اصلی دارد:

۱. اسکن فایل ورودی (توسط کلاس Scanner) و استخراج توکن‌ها.

۲. تجزیه و تحلیل نحوی (توسط کلاس Parser) برای بررسی صحت ساختار کد بر اساس قواعد گرامر.

```
std::vector<Token> scanFile(const std::string& filename) {
    std::vector<Token> parsedTokens;
    Scanner scanner;
    std::ifstream file(filename);
    if (!file.is_open()) {
        std::cerr << "Error: Could not open the file " << filename << std::endl;
        return parsedTokens;
    }
    std::string line;
    int lineNumber = 1;
    while (std::getline(file, line)) {
        std::vector<Token> tokens = scanner.scan(line, lineNumber);
        for (const Token& token : tokens) {
            parsedTokens.push_back(token);
        }
        lineNumber++;
    }
    file.close();

    // Example: Displaying all parsed tokens after file scanning
    std::cout << "\nAll parsed tokens:\n";
    for (const Token& token : parsedTokens) {
        std::cout << "Token: " << token.value << ", Type: " <<
token.getTypeAsString() << ", Line: " << token.line << std::endl;
    }
    return parsedTokens;
}
```

این تابع مسئول خواندن فایل ورودی و تبدیل محتوای آن به لیستی از توکن‌ها است. مراحل کار آن به شرح زیر است:

- باز کردن فایل

با استفاده از std::ifstream، فایل ورودی باز می‌شود. اگر فایل وجود نداشته باشد یا قابل دسترسی نباشد، خطا چاپ شده و یک لیست خالی از توکن‌ها بازگردانده می‌شود.

- خواندن خط به خط فایل

فایل به صورت خط به خط (std::getline) خوانده می‌شود. برای هر خط یک شیء Scanner ایجاد می‌شود، متد scan اسکنر روی خط فعلی فراخوانی می‌شود و توکن‌های آن خط استخراج می‌شوند، توکن‌های استخراج شده به لیست کلی توکن‌ها (parsedTokens) اضافه می‌شوند و شماره خط (lineNumber) برای گزارش‌دهی خطاها به روز می‌شود.

- نمایش توکن‌ها (برای دیباگ)

پس از اتمام اسکن تمام خطوط، تمام توکن‌های استخراج شده در کنسول نمایش داده می‌شوند. این بخش برای اطمینان از صحت عملکرد اسکنر و مشاهده خروجی آن مفید است.

```
int main(int argc, char* argv[]) {
    if (argc != 2) {
        std::cerr << "Usage: " << argv[0] << " <input_file>" << std::endl;
        return 1;
    }

    std::string filePath = argv[1];
    std::vector<Token> Tokens = scanFile(filePath);
    Parser parser(Tokens);

    if(parser.Parse()) {
        std::cout << "Parsing completed successfully" << std::endl;
    }else {
        std::cout << "Parsing failed" << std::endl;
    }
    return 0;
}
```

تابع `main` نقطه شروع اجرای برنامه است و مراحل زیر را دنبال می‌کند:

- بررسی آرگومان‌های ورودی

اگر تعداد آرگومان‌های ورودی دقیقاً ۲ نباشد (نام برنامه + مسیر فایل)، خطای چاپ می‌شود و برنامه با کد خروجی ۱ پایان می‌یابد.

- اسکن فایل ورودی

مسیر فایل از آرگومان دوم (`argv[1]`) خوانده می‌شود.

تابع `scanFile` فراخوانی شده و توکن‌های فایل در متغیر `Tokens` ذخیره می‌شوند.

- تجزیه و تحلیل نحوی (پارسر)

شیء `Parser` با توکن‌های استخراج شده مقداردهی می‌شود.

متد `Parse` فراخوانی می‌شود تا صحت ساختار توکن‌ها بررسی شود.

- اگر پارسر موفق باشد `Parsing completed successfully` چاپ می‌شود.
- اگر پارسر شکست بخورد `Parsing failed` چاپ می‌شود.