2023.2
Principle of Programming Language
Programming Assignment #1

Internal Documentation

20184256 박성민
20226041 김규리

본 Parser 에서 처리한 오류와 경고

작성한 Parser 에서 처리한 경고는 아래와 같다.

```
typedef enum {
    INVALID_OP,          // operation repitition
    UNKNOWN_ID,          // Unknown token input
    NON_PAIR_LEFT_PAREN, // left parenthesis does not have pair
    EQUAL_MISSING,       // = is missing
    COLON_MISSING,       // : is missing
    SEMI_COLON_REPITITION,
    EOF_SEMI_COLON       // last statement has semicolon
} Warnings;
```

- 유효하지 않은 연산자 토큰 위치 : 가장 앞에 있는 연산자만 남기고 생략한다.

ex) abc := 3 ++**/- 5;

-> abc := 3 + 5;

- UNKNOWN 토큰의 입력 : 적절하지 않은 input 인 UNKNOWN 토큰을 생략한다.

ex) abc := $bcd + 5;

-> abc := bcd + 5;

- 좌 괄호보다 우 괄호가 부족함 : 부족한 좌괄호의 개수만큼 우괄호를 Statement 끝에 삽입한다.

ex) abc := (abc + (bcd + (cde + def;

-> abc := (abc + (bcd + (cde + edf)));

- 대입연산자에 등호 없음 : 대입연산자의 등호 자리에 등호를 채워준다.

ex) abc : 3 + 5;

-> abc := 3 + 5;

- 대입연산자에 콜론 없음 : 대입연산자의 콜론 자리에 콜론을 채워준다.

ex) abc = 3 + 5;

-> abc := 3 + 5;

- 세미콜론이 중복으로 입력됨 : 연속으로 입력된 세미콜론을 무시한다.

ex) abc := 3 + 5;;;;; abc := 5;

-> abc := 3 + 5; abc := 5;

- 마지막 statement 가 세미콜론으로 끝남 : 필요 없는 세미콜론으로 종료되었음을 알린다.

본 Parser 에서 처리한 에러는 아래와 같다.

```
typedef enum {
    UNKNOWN_ERROR,        // undefinable error
    BEGIN_IDENT_MISSING, // statement does not start with identifier
    WRONG_STATEMENT,      // statement structure error
    NOT_DECLARED,         // call undeclared identifier
    TOKEN_LEFT,           // Token is still left in stack
    ZERO_DIVISER,         // divide by 0
    PAREN_PAIR_MISSING,   // right parenthesis does not have pair
    ARGUMENT_MISSING      // operator does not have argument
} Errors;
```

- 대입연산자 이전에 IDENT 가 입력되지 않음

ex) := 3 + 5;

- 잘못된 STATEMENT 구조가 입력됨

ex) abc := 3 5 7;

- 아직 정의되지 않은 IDENTIFIER 에 대한 값 접근

ex) a := 3; a := b + 5;

- 0 으로 값 나누기를 시도함

ex) a := 3 / 0;

- 우 괄호의 개수가 좌 괄호의 개수보다 많음

ex) a := a + b);

- 연산자에 argument 가 부족함

ex) a := a + ;

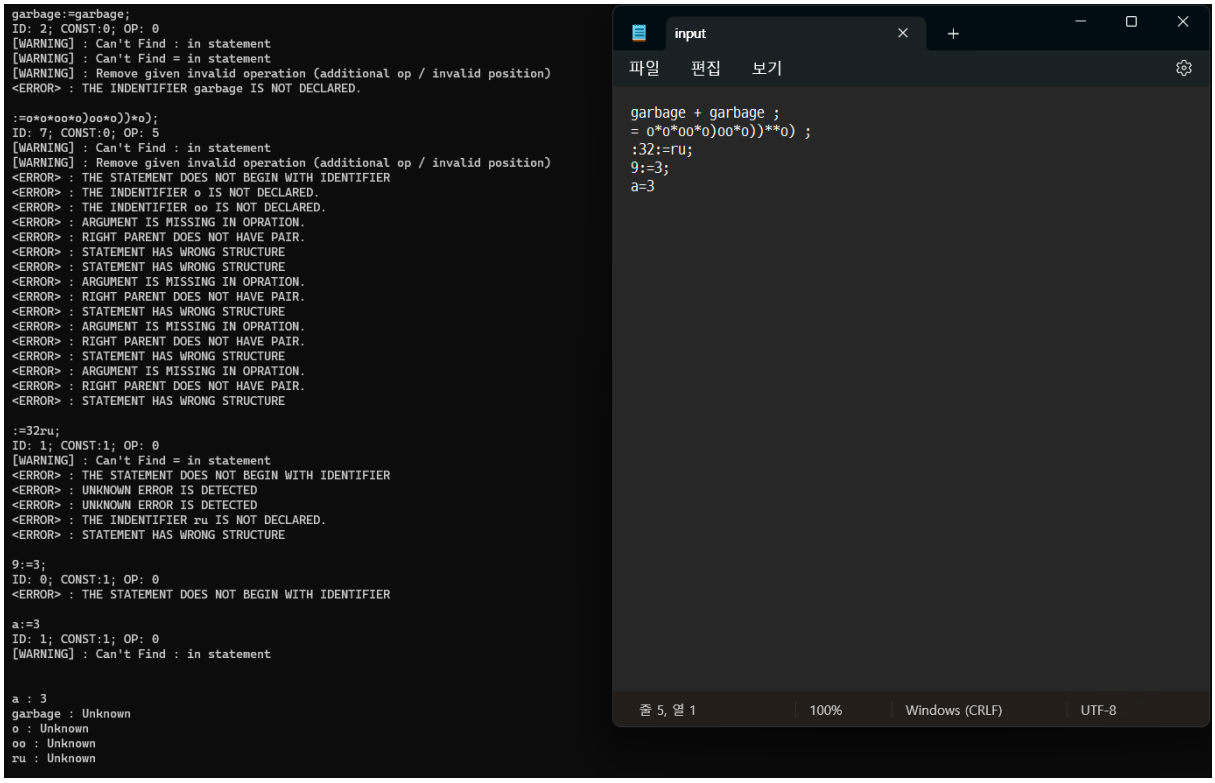
- 토큰 처리 종료 이후에도 스트림에 토큰이 남아있음

* 본 에러에 해당하는 케이스는 모두 상기한 에러로 대체되었음

- 그 외 알 수 없는 에러

* 이외에 발생한 기타 에러는 UNKNOWN_ERROR 로 분류하였음.

실제 출력 예시


garbage:=garbage;
ID: 2; CONST:0; OP: 0
[WARNING] : Can't Find : in statement
[WARNING] : Can't Find = in statement
[WARNING] : Remove given invalid operation (additional op / invalid position)
<ERROR> : THE INDENTIFIER garbage IS NOT DECLARED.
:=o*o*oo*o)oo*o))*o);
ID: 7; CONST:0; OP: 5
[WARNING] : Can't Find : in statement
[WARNING] : Remove given invalid operation (additional op / invalid position)
<ERROR> : THE STATEMENT DOES NOT BEGIN WITH IDENTIFIER
<ERROR> : THE INDENTIFIER o IS NOT DECLARED.
<ERROR> : THE INDENTIFIER oo IS NOT DECLARED.
<ERROR> : ARGUMENT IS MISSING IN OPRATION.
<ERROR> : RIGHT PARENT DOES NOT HAVE PAIR.
<ERROR> : STATEMENT HAS WRONG STRUCTURE
<ERROR> : STATEMENT HAS WRONG STRUCTURE
<ERROR> : ARGUMENT IS MISSING IN OPRATION.
<ERROR> : RIGHT PARENT DOES NOT HAVE PAIR.
<ERROR> : STATEMENT HAS WRONG STRUCTURE
<ERROR> : ARGUMENT IS MISSING IN OPRATION.
<ERROR> : RIGHT PARENT DOES NOT HAVE PAIR.
<ERROR> : STATEMENT HAS WRONG STRUCTURE
<ERROR> : ARGUMENT IS MISSING IN OPRATION.
<ERROR> : RIGHT PARENT DOES NOT HAVE PAIR.
<ERROR> : STATEMENT HAS WRONG STRUCTURE
:=32ru;
ID: 1; CONST:1; OP: 0
[WARNING] : Can't Find = in statement
<ERROR> : THE STATEMENT DOES NOT BEGIN WITH IDENTIFIER
<ERROR> : UNKNOWN ERROR IS DETECTED
<ERROR> : UNKNOWN ERROR IS DETECTED
<ERROR> : THE INDENTIFIER ru IS NOT DECLARED.
<ERROR> : STATEMENT HAS WRONG STRUCTURE
9:=3;
ID: 0; CONST:1; OP: 0
<ERROR> : THE STATEMENT DOES NOT BEGIN WITH IDENTIFIER
a:=3
ID: 1; CONST:1; OP: 0
[WARNING] : Can't Find : in statement
a : 3
garbage : Unknown
o : Unknown
oo : Unknown
ru : Unknown
input
파일 편집 보기
garbage + garbage ;
= o*o*oo*o)oo*o))**o) ;
:32:=ru;
9:=3;
a=3
줄 5, 열 1
100%
Windows (CRLF)
UTF-8


operand1:=3;
ID: 1; CONST:1; OP: 0
(OK)
operand2:=operand1+2;
ID: 2; CONST:1; OP: 1
[WARNING] : Remove given invalid operation (additional op / invalid position)
target:=operand1+operand2*3
ID: 3; CONST:1; OP: 2
(OK)
operand1 : 3
operand2 : 5
target : 18
input
파일 편집 보기
operand1 := 3 ;
operand2 := operand1 + + 2 ;
target := operand1 + operand2 * 3

본 Parser 에서의 오류, 경고 처리를 위한 설계

Parser 에서 에러나 경고가 발생한 경우에도 적절히 조치를 취하고 그 다음 statement 를 처리해주기 위해서 다양한 설계를 적용하였고, 그 중에서 Symbol Table 에서 유효하지 않은 변수 값을 가져오려고 하는 경우, 에러가 발생한 Statement 에서 값을 대입하는 경우에 대해 처리해주기 위해 OptionalData 이라는 Type 을 선언하고, 이를 상속받는 타입인 OptionalInt, OptionalDouble Type 을 정의하였다. 그리고 각 Type 에 대한 +, -, *, / 연산자를 오버로딩하여 Parser 에 활용해주었다.

OptionalData 는 Nullable, Unknownable Type 으로, 아직 선언되지 않은 데이터는 isNull 이 참이고, Parsing 과정에서 오류가 발생해 유효하지 않은 값이 대입된 경우에는 isUnknown 이 참이 된다. 이후 연산에서 이 flag 를 활용해 symbol table 의 값을 적절하게 채워넣을 수 있다.

```
template<typename T>
class OptionalData {
protected:
    OptionalData() :isNull(true), isUnknown(false), data(0) {} // null data CTOR
    OptionalData(bool isUnknown) :isNull(false), isUnknown(isUnknown), data(0) {} // Unknown data CTOR
    OptionalData(T data) :isNull(false), isUnknown(false), data(data) {}
public:
    using data_type = T;
    bool isNull; // if not delared, Null
    bool isUnknown; // if statement cannot assure data, unknown data
    T data; // data

    bool isValid() { return !(isNull | isUnknown); }

    std::string GetData() { // output data
        if (isNull) return "NullData";
        if (isUnknown) return "Unknown";
        else return to_string(data);
    }
};
```

Optional Dat 를 상속받는 Optional Int Type 은 아래와 같이 정의되며, 아래에 여러 연산자들을 오버로드 하여 flag 를 활용한 연산들을 구현하였다.

```
class OptionalInt : public OptionalData<int> {
private:
    OptionalInt(bool isUnknown) : OptionalData(isUnknown) {} //OptionalInt(true) - UnknownData

public:
    static OptionalInt GetUnknown() { return OptionalInt(true); }

    OptionalInt() : OptionalData() {} //Null Data
    OptionalInt(int data) : OptionalData(data) {
    }

    //operator overloading
    OptionalInt& operator=(const OptionalInt& o) {
        isUnknown = o.isUnknown;
        data = o.data;
        return *this;
    }

    OptionalInt& operator=(const OptionalDouble& o) {
        isUnknown = o.isUnknown;
        data = o.data;
        return *this;
    }
```

본 프로그램에서는 연산 결과로 정수형(Integer)만 취급하고 있지만, a / b 연산을 a * (1 / b) 의 형식으로 처리하도록 설계하였기 때문에, Optional Double 타입을 구현하였고, 이를 OptionalInt 로 변환할 수 있게 내부 함수도 만들어주었다. Optional Double 타입에 맞게 연산자들도 오버로드하여 해당 타입에 대해 적절히 연산을 수행할 수 있게 구현하였다.

```
class OptionalDouble : public OptionalData<double> {
private:
    OptionalDouble(bool isUnknown) : OptionalData(isUnknown) {}

public:
    static OptionalDouble GetUnknown(){ return OptionalDouble(true); } // Unknown data
    OptionalDouble() :OptionalData() {} // Null data
    OptionalDouble(double data) : OptionalData(data) {}

    //operator overloading

    OptionalDouble& operator=(const OptionalDouble& o) {
        isUnknown = o.isUnknown;
        data = o.data;
        return *this;
    }

    OptionalDouble& operator+ (const OptionalDouble& o) {
        OptionalDouble value = OptionalDouble(0.0);
        value.isUnknown = isUnknown | o.isUnknown;
        value.data = data + o.data;
        return value;
    }

    OptionalDouble& operator* (const OptionalDouble& o) {
        OptionalDouble value = OptionalDouble(0.0);
        value.isUnknown = isUnknown | o.isUnknown;
        value.data = data * o.data;
        return value;
    }
};
```

그리고, 두 타입간의 형변환을 통해 쉽게 값을 연산할 수 있도록 Convert 함수도 작성해 적용하였다.

```
template <class T1, class T2>
T1 ConvertType(T2& t) { // Convert Function between OptionalInt - OptionalDouble
    static_assert(std::is_base_of<OptionalData<typename T1::data_type>, T1>::value, "OptionalData");
    static_assert(std::is_base_of<OptionalData<typename T2::data_type>, T2>::value, "OptionalData");
    T1 value = T1();
    value.isNull = t.isNull;
    value.isUnknown = t.isUnknown;
    value.data = t.data;

    return value;
}
```

전반적인 프로그램의 동작 절차

작성한 분석 프로그램은 main 함수 호출 인자에서 filename 을 전달받아 입력 파일 스트림을 생성하면서 시작된다. 만약 파일을 열 수 없다면 에러메시지를 출력하고 프로그램을 종료한다.

```
int main(int argc, char *argv[]) {

    // 파일명은 argv[1]을 통해 전달됨
    const char *filename = argv[1];

    // 입력 파일 스트림 생성
    ifstream inputFile(filename);
    if (!inputFile) {
        cout << "[" << filename << "] 파일을 열 수 없습니다" << endl;
        return 0;
    }
```

그 후, LexicalAnalyzer 클래스의 객체를 생성하고 멤버 변수 analyzeInputFile 함수에 파일스트림을 인자로 전달해 토큰에 대한 분석을 진행한다. 토큰 분석 결과와 그 과정에서 생성한 symbolTable 을 인자로 받아 Parser 객체를 생성해준 뒤, 입력에 대한 구문 분석을 진행한다.

```
    // Lexical Analyzer 객체를 생성
    LexicalAnalyzer lexicalAnalyzer = LexicalAnalyzer();

    // Lexical Analyzer를 통해 토큰 분석 진행
    lexicalAnalyzer.analyzeInputFile(inputFile);

    // Lexical Analyzer의 구문 분석 결과 -토큰 리스트, 심볼 테이블- 를 Parser에게 전달하며
    // Parser 객체를 생성
    Parser parser(lexicalAnalyzer.getAnalyzedResult(), lexicalAnalyzer.getSymbolTable());

    // Parser를 통해 구문 분석을 진행
    parser.Parse();
```

모든 처리가 완료되면 파일스트림을 닫고 프로그램을 종료한다.

```
    // 모든 처리가 완료된 후 파일 닫기
    inputFile.close();

    return 0;
}
```

Lexical Analyzer 의 동작

설계한 Lexical Analyzer 클래스의 인터페이스는 아래와 같다. 주어진 파일 스트림에 대한 토큰 분류 결과를 담는 _lexResult, 그 과정에서 생성된 심볼 테이블을 담는 _symbolTable, 입력 문자열에 대해 토큰을 분류하는 analyzeString 함수와, 로직 처리를 시작하고 결과를 반환하는 함수들로 구성되어 있다.

```
class LexicalAnalyzer {
private:
  vector<tuple<Tokens, string>> _lexResult;
  vector<string> _symbolTable;
  Tokens analyzeString (string);

public:
  void analyzeInputFile(ifstream&);
  vector<tuple<Tokens, string>> getAnalyzedResult();
  vector<string> getSymbolTable();
};
```

주어진 파일의 코드를 토큰으로 분류하기 위해, 아래처럼 토큰을 enum 타입으로 정의하였다.

```
#ifndef token_h
#define token_h

typedef enum {
    PROGRAM,
    STATEMENTS,
    STATEMENT,
    SEMI_COLON,
    IDENT,
    EQUAL,
    COLON,
    EXPRESSION,
    TERM,
    TERM_TAIL,
    ADD_OP,
    FACTOR,
    FACTOR_TAIL,
    MULT_OP,
    LEFT_PAREN,
    RIGHT_PAREN,
    CONST,
    END_OF_FILE,
    UNKNOWN,
} Tokens;

#endif /* token_h */
```

아래는 입력된 문자열에 대해 특수 기호가 아닌 숫자, 알파벳, _이 연속으로 입력된 경우, 이를 IDENT, CONST, UNKNOWN 으로 분류해주는 함수이다. 숫자로 시작하지 않으면 IDENT 로, 숫자로 시작할 때 모든 문자가 숫자이면 CONST 로, 그 외에는 UNKNOWN 으로 분류한다.

```
// 주어진 문자열에 대해 토큰을 분석하는 함수
Tokens LexicalAnalyzer::analyzeString (string str) {

    // 숫자로 시작하지 않는 경우
    if (!(str[0] >= '0' && str[0] <= '9')){

        // 새로운 Identifier가 들어오면 symbol table에 등록
        // Identifier에 대한 init 값은 null 이다.
        if(find(_symbolTable.begin() , _symbolTable.end(), str) == _symbolTable.end()) {
            LexicalAnalyzer::_symbolTable.push_back(str);
        }

        // IDENT로 토큰을 분류
        return IDENT;
    }

    // 숫자로 시작하는 경우, 모든 문자가 숫자인지 확인
    // 만약에 문자가 하나라도 포함되어 있다면 UNKNOWN 토큰으로 처리
    for (string::iterator iter = str.begin(); iter != str.end(); iter++) {
        if (!('0' <= *iter && *iter <= '9')) {
            return UNKNOWN;
        }
    }
    // 모든 문자가 숫자로 들어온 경우에는, CONST로 토큰을 분류
    return CONST;
}
```

아래는 주어진 파일스트림을 읽으면서 토큰을 분류하는 함수이다. 파일에 남은 문자가 없을 때까지 한 글자씩 반복하며 진행한다. line 변수에는 현재 읽은 문자가 혼자서 토큰이 될 수 없는 경우(IDENT, CONST)를 저장하는데에 사용된다.

받아온 글자가 공백문자인 경우에는 line 에 있는 문자열의 토큰을 분류하고 결과에 추가한다.

받아온 글자가 알파벳, 숫자, _ 인 경우에는 line 에 추가해 다음 문자와 함께 처리한다.

```
void LexicalAnalyzer::analyzeInputFile(ifstream& inputFile) {
    vector<tuple<Tokens, string>> lexicalResult;
    string line;
    char c;

    // 모든 문자를 처리할 때 까지 한 글자씩 아래 로직을 처리함
    while (inputFile.get(c)) {
        // 공백 문자가 들어온 경우에 대한 처리
        if (c <= 32) {
            // 공백 문자를 읽었을 때, 앞서 모으고 있던 line이 있다면
            // CONST/IDENT/UNKNOWN으로 분류 작업을 한 뒤 line을 비우고 넘어감
            if (!line.empty()) {
                lexicalResult.push_back(make_tuple(analyzeString(line), line));
                line.clear();
            }
        }

        // alphabet, number, underscore에 대한 처리
        else if (('0' <= c && c <= '9') || ('A' <= c && c <= 'Z') || ('a' <= c && c <= 'z') || c == '_') {
            line += c;
        }
```

나머지 기호들에 대해서는 앞서 입력받던 line 에 대해 처리를 해주고 나서 입력에 해당하는 토큰으로 분류해 결과에 추가해준다. 주어진 문법에 없는 기호의 경우 UNKNOWN 으로 처리하였다.

```
    // 기호에 대한 처리
    else {
        // 기호를 읽었을 때, 앞서 모으고 있던 line이 있다면
        // CONST/IDENT/UNKNOWN으로 분류 작업을 한 뒤 line을 비우고 기호에 대한 처리를 이어감
        if (!line.empty()) {
            lexicalResult.push_back(make_tuple(analyzeString(line), line));
            line.clear();
        }

        switch (c) {
            case '(':
                lexicalResult.push_back(make_tuple(LEFT_PAREN, "("));
                break;
            case ')':
                lexicalResult.push_back(make_tuple(RIGHT_PAREN, ")"));
                break;
            case '*':
                lexicalResult.push_back(make_tuple(MULT_OP, "*"));
                break;
            case '/':
                lexicalResult.push_back(make_tuple(MULT_OP, "/"));
                break;
            case '+':
                lexicalResult.push_back(make_tuple(ADD_OP, "+"));
                break;
            case '-':
                lexicalResult.push_back(make_tuple(ADD_OP, "-"));
                break;
            case ';':
                lexicalResult.push_back(make_tuple(SEMI_COLON, ";"));
                break;
            case ':':
                lexicalResult.push_back(make_tuple(COLON, ":"));
                break;
            case '=':
                lexicalResult.push_back(make_tuple(EQUAL, "="));
                break;
            default:
                lexicalResult.push_back(make_tuple(UNKNOWN, string(1, c)));
                break;
        }
    }
```

모든 입력을 다 처리한 뒤에는 마지막으로 line 에 남아있는 문자열을 처리한 뒤, END_OF_FILE 토큰을 결과에 넣고 분석 결과를 저장한다..

```
    // 모든 글자를 읽은 뒤 마지막으로 남은 line에 있는 문자열에 대한 분류를 진행
    if (!line.empty()) {
        lexicalResult.push_back(make_tuple(analyzeString(line), line));
        line.clear();
    }

    // END_OF_FILE 토큰을 가장 마지막에 삽입
    lexicalResult.push_back(make_tuple(END_OF_FILE, ""));

    // 분석 결과를 Lexical Analyzer 객체의 _lexResult 변수에 저장
    LexicalAnalyzer::_lexResult = lexicalResult;
}
```

Parser 의 동작

Parser 에서 Program 함수를 실행하면서 파싱을 시작한다. Program 함수에서는 값을 초기화하고 statements 함수를 호출한 뒤 파싱이 완료된 뒤 Symbol Table 의 데이터를 출력하면서 함수를 종료한다..

```
void Parser::Parse() { program(); }

// **** TOKEN FUNCTION ****
// Program
void Parser::program() {
    resetVariablesForNewStatement();
    statements();
    cout << "\n";
    SymbolOutput();
    return;
}
```

statements 함수에서는 statement 함수를 호출해 하나의 statement 를 처리한다. 이후 세미콜론이 나올 때 초기화를 진행하고 statement 함수를 다시 호출해 처리한다. 중간에 세미콜론이 반복해서 등장하는 경우에는, 이에 대해 로그를 띄우고 처리한다. 모든 statement 에 대한 처리가 끝나면 END_OF_FILE 토큰을 확인하고 처리를 종료한다.

```
// STATEMENTS
void Parser::statements() {

    // call STATEMENT
    statement();

    while (isToken(SEMI_COLON)) {
        // SEMICOLON에 대한 처리
        std::cout << getToken() << endl;

        // SEMICOLON
        moveNextAndCheckValid();

        while (isToken(SEMI_COLON)) {
            logWarning(SEMI_COLON_REPITITION);
            moveNextAndCheckValid();
        }

        // ERROR, WARNING, COUNT message
        printStatementLog();
        // STATEMENT
        statement();
    }

    // 만약 모든 처리가 완료되었음에도 토큰이 남아있다면
    // 에러로 기록
    if (!isToken(END_OF_FILE)) {
        cout << getToken() << endl;
        logError(TOKEN_LEFT);
    }

    // 남은 ERROR, WARNING, COUNT 메시지 출력하며 statement 파싱 종료
    cout << endl;
    printStatementLog();

    return;
}
```

statement 함수에서는 세미콜론으로 구분되는 하나의 문장 단위를 파싱하는 함수이다. statment 함수 내에서, 좌항 IDENT 와 :=, 이후 우항의 EXPRESSION 까지 처리를 한다. 이 과정에서 오류나 경고가 발생하는 경우 이를 처리해준다. IDENT 는 함수로 호출되어, subprogram 으로 처리된다.

```
// STATEMENT
void Parser::statement() {

    //identifier
    std::string id;

    // 만약 입력된 토큰이 모두 소진되었는데 세미콜론이 있는 경우에는
    // EOF 위치에 세미콜론이 있음을 경고로 출력
    if (isToken(END_OF_FILE)) {
        logWarning(EOF_SEMI_COLON);
        return;
    }

    // IDENT 처리
    if (isToken(IDENT)) {
        id = ident();
    }
    // IDENT 처리를 못한 경우 오류, 경고 처리
    else {
        if (isToken(COLON) || isToken(EQUAL)) {
            logError(BEGIN_IDENT_MISSING);
        }
        else {
            printToken();
            logError(BEGIN_IDENT_MISSING);
            moveNextAndCheckValid();
        }
    }
```

대입연산자 :=는 콜론 : 과 등호 = 로 분리하여 입력받아 처리한다. 이 과정에서 예상치 못한 입력이 들어오는 경우에는 경고를 넣으며 적절하게 처리해준다.

```
    // COLON 처리
    if (isToken(COLON)) {
        printToken();
        moveNextAndCheckValid();
    }
    // COLON 처리를 못한 경우 오류, 경고 처리
    else {
        // COLON이 없는 경우, 콜론을 삽입하며 경고 추가 후 진행
        logWarning(COLON_MISSING);
        cout << ":";
    }

    // EQUAL 처리
    if (isToken(EQUAL)) {
        printToken();
        moveNextAndCheckValid();
    }
    // EQUAL 처리를 못한 경우 오류, 경고 처리
    else {
        // EQUAL이 없는 경우, 콜론을 삽입하며 경고 추가 후 진행
        logWarning(EQUAL_MISSING);
        cout << "=";
    }
```

대입연산자 이후에 등장하는 토큰들인 EXPRESSION 은 expression 함수를 호출해 subprogram 에서 처리를 한다. 이 과정에서 발생한 문제에 대해서 expression 함수가 종료된 뒤에 에러와 경고를 기록한다.

위 과정이 종료된 뒤에, 만약 좌항 IDENT 의 입력이 정상적으로 들어온 경우에는 정상적으로 계산된 expression 에서의 결과를 대입해준다.

```
    // EXPRESSION 처리
    OptionalInt value = expression();

    // EXPRESSION 내 잘못된 구조에 대한 처리
    while (!isToken(END_OF_FILE) && !isToken(SEMI_COLON)) {
        expression();
        logError(WRONG_STATEMENT);
        value = OptionalInt::GetUnknown();
    }

    // EXPRESSION 내 쌍이 맞지 않는 좌우 괄호에 대한 처리
    while (parenCountPerStatement > 0) {
        logWarning(NON_PAIR_LEFT_PAREN);
        cout << ")";
        parenCountPerStatement--;
    }

    // IDENT로 시작하지 않는 경우에 대한 처리
    if (id.empty()) {
        return;
    }

    // IDENT가 유효한 값임을 처리
    _symbolTable.find(id)->second.isNull = false;

    // 문제가 없는 경우, IDENT에 연산된 값 대입을 수정
    if (hasError()) {
        _symbolTable.find(id)->second = OptionalInt::GetUnknown();
    }
    else {
        _symbolTable.find(id)->second = value;
    }

    return;
}
```

expression 함수는 아래처럼 term 과 term_tail 함수를 호출한 뒤, 하위 계산 결과를 더해 반환한다. term_tail 에 아무것도 없는 경우에는 0 을 반환해 더해주는 방식으로 처리하였다.

```
// EXPRESSION
OptionalInt Parser::expression() {
    OptionalInt value1 = term();
    OptionalInt value2 = term_tail();
    return (value1 + value2);
}
```

term 함수에서는 factor 함수와 factor_tail 함수의 결과를 곱한 값을 반환한다. 나눗셈의 계산을 수행하기 위해서 factor 의 값과 factor_tail 의 값을 Double 타입으로 변환해 곱셈 연산을 처리해준 뒤, 이를 Int 타입으로 다시 캐스팅해 반환한다.

예를 들어, 16 / 8 의 연산을 수행해야 하는 경우, 16.0 * (0.125) 을 연산한 결과인 2.0 을 Int 로 형변환한 2 를 반환하게 된다.

```
// TERM
OptionalInt Parser::term() {
    OptionalInt factorResult = factor();
    // casting value
    OptionalDouble value1 = ConvertType<OptionalDouble, OptionalInt>(factorResult);
    OptionalDouble value2 = factor_tail();
    OptionalDouble multResult = value1 * value2;
    // casting value
    return ConvertType<OptionalInt, OptionalDouble>(multResult);
}
```

term_tail 함수에서는 ADD_OP 연산자가 +인지 -인지에 따라 하위의 term 과 term_tail subprogram 의 연산 결과를 부호에 맞게 변환하여 반환한다. 변환된 값은 해당 함수를 호출한 지점에 반환되어 덧셈을 통해 - 또는 + 연산을 정상적으로 수행하도록 설계하였다.

```
OptionalInt Parser::term_tail() {
    if (isToken(ADD_OP)) {
        int opType = add_op();
        OptionalInt value1 = term();
        OptionalInt value2 = term_tail();
        OptionalInt value = value1 + value2;
        // return negative is operation is minus
        if (opType) {
            value.data = 0 - value.data;
        }
        return value;
    }
    // return default value
    else {
        return OptionalInt(0);
    }
}
```

factor 함수에서는 처리될 순서의 토큰을 IDENT, CONST, (EXPRESSION) 3 가지로 분류한다.

(EXPRESSION) 의 경우, 좌 괄호의 개수를 동일하게 우 괄호를 배치할 수 있도록 처리하기 위해 좌괄호의 개수를 카운트하고, EXPRESSION 을 처리한 뒤 정상적으로 우 괄호 토큰이 들어오는지 확인해준다. 만약 우 괄호의 개수가 부족한 경우에는 연산 결과에 영향을 미치지 않기 위해 Statement 의 가장 오른쪽에 우 괄호를 부족한 만큼 추가하는 방식으로 처리하였다. 우 괄호의 개수가 좌 괄호보다 많은 경우에는 에러로 처리하였다.

```
OptionalInt Parser::factor() {
    if (isToken(IDENT)) {
        return ident_val();
    }
    else if (isToken(CONST)) {
        return const_val();
    }
    else if (isToken(LEFT_PAREN)) {
        parenCountPerStatement++;
        printToken();
        moveNextAndCheckValid();
        OptionalInt value = expression();

        // paren count matching
        if (isToken(RIGHT_PAREN)) {
            if (parenCountPerStatement > 0) {
                parenCountPerStatement--;
                printToken();
            }
            // no left parethesis
            else {
                logError(PAREN_PAIR_MISSING);
                printToken();
            }
            moveNextAndCheckValid();
        }
        return value;
    }
```

IDENT, CONST, (EXPRESSION) 로 분류될 수 없는 토큰이 들어온 경우인 세미콜론, EOF, 우 괄호, 연산자 또는 그 외의 토큰이 들어온 경우에는 각각의 경우에 대해 적합하게 경고 또는 에러를 기록하고 처리한다. 이 경우, 신뢰할 수 없는 연산 값이라 판단하여, statement subprogram 에서 대입 연산 시 IDENT 에 UNKNOWN 으로 값이 대입되도록 설계하였다.

```
    else {
        // check error and warnings
        // statement end without argument
        if (isToken(SEMI_COLON)||isToken(END_OF_FILE)) {
            logError(ARGUMENT_MISSING);
            return OptionalInt::GetUnknown();
        }
        else if (isToken(RIGHT_PAREN)) {
            //operator needs argument
            logError(ARGUMENT_MISSING);

            // check for parenthesis
            if (parenCountPerStatement > 0)
                parenCountPerStatement--;
            // check for parenthesis
            else
                logError(PAREN_PAIR_MISSING);

            printToken();
            moveNextAndCheckValid();
            return OptionalInt::GetUnknown();
        }

        // operator repetition
        if (isToken(MULT_OP) || isToken(ADD_OP)) {
            logWarning(INVALID_OP);
            moveNextAndCheckValid();
            return factor();
        }
        // unknown
        else {
            logError(UNKNOWN_ERROR);
            moveNextAndCheckValid();
            return factor();
        }
    }
}
```

factor_tail 함수의 경우 MULT_OP 에 대해 처리를 한다. 연산자가 * 인지 / 인지에 따라 하위 factor 와 factor_tail subprogram 의 연산 결과에 대해 그대로 또는 역수로 반환한다. 변환된 값은 해당 함수를 호출한 지점에 반환되어 곱셈을 통해 * 또는 / 연산을 정상적으로 수행하도록 설계하였다.

```
OptionalDouble Parser::factor_tail() {
    if (isToken(MULT_OP)) {
        int opType = mult_op();
        OptionalInt value1 = factor();
        OptionalDouble value2 = factor_tail();
        OptionalDouble value = value1.data * value2.data;

        // if operation is division
        if (opType) {
            // cannot assure data
            if (value.isUnknown) {
                return OptionalDouble::GetUnknown();
            }
            // zero divisor error
            else if (value.data == 0) {
                logError(ZERO_DIVISER);
                return OptionalDouble::GetUnknown();
            }
            else {
                value = 1.0 / value.data;
            }
        }
        return value;
    }

    // return default value
    else {
        return OptionalDouble(1.0);
    }
}
```

ident 함수는 대입연산자의 왼쪽에 위치한 IDENT TOKEN 에 대해 처리하는 함수이다.

각 Statement 에 대한 ident 토큰의 개수를 카운트한다. IDENT 토큰에서 발생한 문제에 대해서는 Statement 에서 처리하도록 설계하였다.

```
// STATEMENT begin identifier - check declarations
std::string Parser::ident() {
    // CODE ERROR : >>CHECK CODE<<
    if (!isToken(IDENT)) { throw std::exception(); }

    std::string value = getToken();
    printToken();
    idCountPerStatement += 1;
    moveNextAndCheckValid();
    return value;
}
```

ident_val 함수는 대입연산자의 오른쪽에 위치한 IDENT TOKEN 에 대해 처리하는 함수이다.

IDENT 토큰을 읽은 경우에 Symbol Table 에서 저장된 값을 찾아 반환하는 역할을 수행한다. 만약 아직 선언되지 않은 변수의 경우 null 로 되어있어 에러를 기록하고, 대입에서 오류가 있었던 값은 unknown 으로 저장되어있고, unknown 값을 반환해 상위 연산의 결과가 연쇄적으로 unknown 으로 처리되도록 설계하였다.

```
// ident value
OptionalInt Parser::ident_val() {
    OptionalInt value;

    // CODE ERROR : >>CHECK CODE<<
    if (!isToken(IDENT)) { throw std::exception(); }
    printToken();
    idCountPerStatement += 1;
    auto iter = _symbolTable.find(getToken());

    // declaration check
    if (iter->second.isNull) {
        logError(NOT_DECLARED, getToken());

        // Identifier
        iter->second.isNull = false;
        iter->second.isUnknown = true;

        moveNextAndCheckValid();
        return OptionalInt::GetUnknown();
    }
    moveNextAndCheckValid();
    return iter->second;
}
```

add_op 함수는 입력받은 토큰이 +인지 -인지를 검사한 뒤 반환한다.

```
int Parser::add_op() {
    // CODE ERROR : >>CHECK CODE<<
    if (!isToken(ADD_OP)) { throw std::exception(); }
    printToken();
    opCountPerStatement += 1;

    // return true only if op is -
    int value = getToken() == "-";
    moveNextAndCheckValid();

    // return null value - no logic for unknown value
    if (hasError()) {
        return 0;
    }
    return value;
}
```

mult_op 함수는 입력받은 토큰이 /인지 *인지를 검사한 뒤 반환한다.

```
int Parser::mult_op() {
    // CODE ERROR : >>CHECK CODE<<
    if (!isToken(MULT_OP)) { throw std::exception(); }
    printToken();
    opCountPerStatement += 1;
    int value = (getToken() == "/");
    moveNextAndCheckValid();

    // return null value - no logic for unknown value
    if (hasError()) return 0;
    return value;
}
```

const_val 함수는 입력받은 토큰의 정수 값을 반환한다.

```
OptionalInt Parser::const_val() {
    //CODE ERROR : >>CHECK CODE<<
    if (!isToken(CONST)) { throw std::exception(); }

    int data;
    std::stringstream ss(getToken());
    printToken();
    constCountPerStatement += 1;
    ss >> data;
    moveNextAndCheckValid();

    //ERROR -> return Unknown value
    if (hasError()) {
        return OptionalInt::GetUnknown();
    }
    return OptionalInt(data);
}
```

printWarningAndErrorList 함수에서는 각 Statement 를 처리하면서 기록해둔 에러와 경고를 출력한다.

경고로 처리한 케이스는 다음과 같다.

- 유효하지 않은 연산자 토큰 위치
- UNKNOWN 토큰의 입력
- 좌 괄호보다 우 괄호가 부족함
- 대입연산자에 등호 없음
- 대입연산자에 콜론 없음
- 세미콜론이 중복으로 입력됨
- 마지막 statement 가 세미콜론으로 끝남

```
// print Error and Warning
void Parser::printWarningAndErrorList() {
    if (warningList.size() == 0 && errorList.size() == 0) {
        std::cout << "(OK)" << std::endl;
    }
    else {
        for (auto it : warningList) {
            cout << "[WARNING] : ";
            switch (get<0>(it)) {
                case INVALID_OP:
                    std::cout << "Remove given invalid operation (additional op / invalid position)" << std::endl;
                    break;

                case UNKNOWN_ID:
                    std::cout << "Invalid Lexeme is detected" << std::endl;
                    break;

                case NON_PAIR_LEFT_PAREN:
                    std::cout << "() Pair does not matched" << std::endl;
                    break;

                case EQUAL_MISSING:
                    std::cout << "Can't Find = in statement" << std::endl;
                    break;

                case COLON_MISSING:
                    std::cout << "Can't Find : in statement" << std::endl;
                    break;

                case SEMI_COLON_REPITITION:
                    std::cout << "Semicolon is repeated." << std::endl;
                    break;

                case EOF_SEMI_COLON:
                    std::cout << "Last statement do not require semi colon" << std::endl;
                    break;
            }
        }
```

에러로 처리한 케이스는 다음과 같다.

- 대입연산자 이전에 IDENT 가 입력되지 않음
- 잘못된 STATEMENT 구조가 입력됨
- 아직 정의되지 않은 IDENTIFIER 에 대한 값 접근
- 토큰 처리 종료 이후에도 스트림에 토큰이 남아있음
- 0 으로 값 나누기를 시도함
- 우 괄호의 개수가 좌 괄호의 개수보다 많음
- 연산자에 argument 가 부족함
- 그 외 알 수 없는 에러

```
        for (auto it : errorList) {
            cout << "<ERROR> : ";
            switch (get<0>(it)) {

                case UNKNOWN_ERROR:
                    std::cout << "UNKNOWN ERROR IS DETECTED" << std::endl;
                    break;
                case BEGIN_IDENT_MISSING:
                    std::cout << "THE STATEMENT DOES NOT BEGIN WITH IDENTIFIER" << std::endl;
                    break;
                case WRONG_STATEMENT:
                    std::cout << "STATEMENT HAS WRONG STRUCTURE" << std::endl;
                    break;
                case NOT_DECLARED:
                    std::cout << "THE INDENTIFIER " << get<1>(it)<< " IS NOT DECLARED." << std::endl;
                    break;
                case TOKEN_LEFT:
                    std::cout << "TOKEN IS STILL LEFT IN STREAM" << std::endl;
                    break;
                case ZERO_DIVISER:
                    std::cout << "VALUE CAN'T BE DIVIDED BY ZERO" << std::endl;
                    break;
                case PAREN_PAIR_MISSING:
                    std::cout << "RIGHT PARENT DOES NOT HAVE PAIR." << std::endl;
                    break;

                case ARGUMENT_MISSING:
                    std::cout << "ARGUMENT IS MISSING IN OPRATION." << std::endl;
                    break;
            }
        }
    }
    cout << '\n';
}
```