# Internal Document

**20216018_이동훈 20210189_한동희**

```
int main() {
    string fname, str;
    cin >> fname;
    ifstream ifs( s: fname);
    Parser parser;
    while (getline( &: ifs,  &: str)) {
        cout << str << endl;
        vector<string> lineinf = splitter(str);
        lexical( v: lineinf);
        parser.program( v: lineinf);
        parser.print_result();
        cout << endl << endl;
    }
    parser.print_ID();
    cout << endl;
    return 0;
}
```

`getline(ifs, str)` 한줄식 입력받고 입력 받은 한 줄을 먼저 출력한 뒤

입력 받은 한 줄 단위로 while루프를 돌게됩니다.

**splitter 함수에 대해 설명하겠습니다**

```
vector<string> splitter(const string &str) {
    vector<string> result;
    istringstream iss(str);
    string word;
    while (iss >> word) {
        size_t pos = 0;
        while ((pos = word.find( c: '(', pos)) != string::npos) {
            if (pos != 0) {
                result.push_back(word.substr( pos: 0,  n: pos));
                word = word.substr(pos);
                pos = 0;
            } else {
                if (word.size() > 1) {
                    result.emplace_back("(");
                    word = word.substr( pos: 1);
                } else {
                    result.push_back(word);
                    word = "";
                    break;
                }
            }
        }
        pos = 0;
        while ((pos = word.find( c: ')', pos)) != string::npos) {
            if (pos != 0) {
                result.push_back(word.substr( pos: 0,  n: pos));
                word = word.substr(pos);
                pos = 0;
            } else {
                if (word.size() > 1) {
                    result.emplace_back(")");
                    word = word.substr( pos: 1);
                } else {
                    result.push_back(word);
                    word = "";
                    break;
                }
            }
        }
        if (!word.empty()) result.push_back(word);
    }
    return result;
}
```

1. 빈 문자열 `result`를 생성하여 결과를 저장할 `vector<string>`를 초기화합니다.
2. 입력 문자열 `str`을 `istringstream` 객체 `iss`를 사용하여 스트림으로 변환합니다. 이 스트림을 사용하여 문자열을 공백으로 구분된 여러 단어로 분할합니다.
3. `while` 루프를 사용하여 스트림에서 단어를 읽어옵니다. 각 단어는 `iss >> word`를 통해 읽어옵니다.
4. 각 단어인 `word`에 대해 두 개의 루프가 있습니다. 하나는 여는 괄호 "("를 처리하고, 다른 하나는 닫는 괄호 ")"를 처리합니다.
5. 여는 괄호 "(" 루프는 `while ((pos = word.find('(', pos)) != string::npos)`로 시작합니다. 이 루프는 단어에서 "(" 문자를 찾을 때까지 반복하며, `pos` 변수를 사용하여 검색 위치를 업데이트합니다.
6. "("를 찾으면, 먼저 `pos`가 0이 아닌 경우에는 해당 위치까지의 부분 문자열을 `result` 벡터에 추가하고, `word`에서 해당 부분 문자열을 삭제합니다.
7. "("를 찾았지만 `pos`가 0인 경우, 이것은 "(" 문자가 단어의 처음에 있는 경우입니다. 이 경우, `word`가 길이가 1보다 크면 "("를 따로 `result` 벡터에 추가하고, 그렇지 않으면 "("를 그대로 `result`에 추가합니다. 그리고 `word`를 빈 문자열로 만들어 루프를 종료합니다.
8. 닫는 괄호 ")" 루프도 위와 유사한 방식으로 작동합니다.
9. 각 루프가 끝나면 남은 `word`가 있다면, 이것은 괄호 "("나 ")"가 없는 일반 단어입니다. 따라서 이 단어를 `result`에 추가합니다.
10. 모든 단어를 처리한 후, `result` 벡터가 문자열 내의 괄호를 올바르게 분리한 결과를 포함하고 있습니다. 이 벡터를 반환합니다.

▼ 이 함수를 호출하면 입력 문자열에서 괄호를 분리한 결과가 담긴 문자열 벡터를 얻을 수 있습니다. 동일한 반환형인 `vector<string> lineinf` 에 반환 받습니다.

**이제 () 기준으로 분할한 string을 담은 벡터를 lexical 분석을 진행 합니다.**

```
void lexical(const vector<string> &v) {
    set<string> ID;
    vector<string> CONST;
    vector<string> OP;
    for (auto i : string : v) {
        if (i == "*" || i == "-" || i == "+" || i == "/") {
            OP.push_back(i);
        } else {
            if (i[0] == '+') {
                i = i.substr( pos: 1);
            } else if (i[0] == '-') {
                i = i.substr( pos: 1);
            }
            if (i[0] <= '9' && i[0] >= '0') {
                CONST.push_back(i);
            } else if ((i[0] >= 'A' && i[0] <= 'Z') || (i[0] >= 'a' && i[0] <= 'z')) {
                ID.insert( x: i);
            }
        }
    }
    cout << "ID: " << ID.size() << "; CONST: " << CONST.size() << "; OP: " << OP.size() << ";" << endl;
}
```

1. `set<string> ID`는 중복되지 않는 식별자들을 저장하기 위한 집합(set) 자료구조를 생성합니다.
2. `vector<string> CONST`는 상수를 저장하기 위한 문자열 벡터를 생성합니다.
3. `vector<string> OP`는 연산자를 저장하기 위한 문자열 벡터를 생성합니다.
4. 주요 작업은 입력 문자열 벡터 `v`를 반복하면서 각 요소 `i`를 처리하는 것입니다.
5. `if (i == "*" || i == "-" || i == "+" || i == "/")`를 사용하여 `i`가 연산자인 경우, 이를 `OP` 벡터에 추가합니다.
6. 그렇지 않은 경우, `i[0]`를 통해 문자열의 첫 번째 문자를 확인합니다. 이때 `+` 또는 `-` 기호가 문자열의 첫 번째에 나타날 수 있으므로 이를 처리하는 로직이 있습니다.
    - `i[0] == '+'`의 경우, 첫 번째 문자가 `+` 기호이므로 `i`의 첫 번째 문자를 제거하고 `i`는 양수 상수가 됩니다.
    - `i[0] == '-'`의 경우, 첫 번째 문자가 `-` 기호이므로 `i`의 첫 번째 문자를 제거하고 `i`는 음수 상수가 됩니다.
7. 그 다음, 문자열의 첫 번째 문자를 확인하여 이것이 숫자인 경우 (`i[0] <= '9' && i[0] >= '0'`), `i`를 `CONST` 벡터에 추가합니다. 이는 상수를 나타냅니다.
8. 그렇지 않으면, 문자열의 첫 번째 문자가 알파벳인 경우 (`(i[0] >= 'A' && i[0] <= 'Z') || (i[0] >= 'a' && i[0] <= 'z')`), `i`를 `ID` 집합에 추가합니다. 이는 식별자를 나타냅니다.

9. 모든 문자열 요소를 처리한 후, `ID`, `CONST`, 및 `OP`의 크기를 출력하여 각 범주에 속하는 항목의 수를 보여줍니다.

▼ 이 함수를 호출하면 입력 문자열 벡터 `v`에서 식별자, 상수, 및 연산자를 분류하고, 분류된 결과의 크기를 출력합니다

**이제 본격적으로 Parser class의 program에 `vector<string> lineinf` 넘겨줍니다.**

1. `private` 섹션:

- `map<string, double> ID_name`: 식별자(ID)와 해당 값(상수 또는 변수 값)을 저장하는 맵입니다.
- `int error_code`: 오류를 나타내는 코드. `0`은 오류가 없음을 나타내며, 양수 값은 오류를 나타냅니다.
- `int putNAN`: `NAN` (숫자가 아님) 값을 처리하기 위한 플래그. `1`이면 `NAN`이 저장됩니다.
- `string error_str`: 오류 메시지를 저장하는 문자열.

```
class Parser {
private:
    map<string, double> ID_name;
    int error_code = 0, putNAN = 0;
    string error_str;
public:
    void program(const vector<string> &v) {
        error_code = 0; // error 발생을 대비한 값 0이 아니면 코드 에러 발생을 의미
        error_str = ""; // error 메시지 저장할 멤버 string
        putNAN = 0;
        statements(v);
    }
```

1. `public` 섹션:

- `void program(const vector<string> &v)`: 파싱 및 계산 작업을 시작하는 함수로, 입력된 문자열 벡터 `v`를 받아서 `statements` 함수를 호출하여 파싱을 수행합니다.
- `void statements(const vector<string> &v)`: 입력 문자열 벡터 `v`를 처리하여 세미콜론(`;`)으로 구분된 개별 문장들을 분리하고, 각 문장을 `statement` 함수로 전달하여 파싱합니다.

- `void statement(const vector<string> &v)`: 한 문장을 처리하며, 할당 연산자(`:=`)를 사용하여 변수에 값을 할당합니다. `v`에서 L-value와 R-value를 구분하여 값을 할당하고, 결과를 `ID_name` 맵에 저장합니다.

```
void statements(const vector<string> &v) {
    vector<string> tmp1; // ; 앞의 한 문장을 임시로 저장
    vector<string> tmp2; // ; 뒤의 전체 문장을 임시로 저장 *사실상 비어있는 벡터(한줄씩 입력받기 때문)
    int state = 0;       // statements 특정 값이 식별 되었을 때 분기를 위한 상태 변수
    for (const string &i: v) {
        if (i == ";") {
            if (state == 1) tmp2.push_back(i); // 한 번 ";" 등장한 이후 ;는 tmp2에 임시 저장
            else state = 1; // 처음 한 번 등장한 ";"만 의미가 있어서 상태 변수 값을 1로 만들어 분기
        } else if (state == 0) tmp1.push_back(i); // ";" 나오기 전까지 앞의 한 문장을 push_back
        else tmp2.push_back(i); // state 1이고 ; 나오지 않은 상황
    }
    if (!v.empty()) {
        statement( v: tmp1);
        statements( v: tmp2);
    }
}

void statement(const vector<string> &v) {
    vector<string> tmp1; // := 앞 L-value
    vector<string> tmp2; // := 뒤 R-value
    int state = 0;
    for (const string &i: v) {
        if (i == ":=") state = 1; // 등장 했을 때 따로 임시 vector에 넣지는 않고 상태변수 값 1로 변경
        else if (state == 0) tmp1.push_back(i); // L-value tmp1에 차례로 push_back
        else tmp2.push_back(i); // R-value tmp2에 차례로 push_back
    }
    double exp = expresssion( v: tmp2); // R-value이므로 결국 int 값으로 리턴받음
    if (putNAN) {
        ID_name[tmp1.front()] = NAN;
    } else {
        ID_name[tmp1.front()] = exp; //L-value: 하나의 key 값을 묶어둠.(즉 tmp1은 ID이고 exp는 R-value)
    }
}
```

- `double expression(const vector<string> &v)`: 수식을 계산하는 함수로, 입력된 문자열 벡터 `v`를 처리하여 수식을 파싱하고 계산한 후 결과 값을 반환합니다.

```
double expresssion(const vector<string> &v) {
    vector<string> tmp1; // + or - 가 나오기 전의 term을 저장할 임시 벡터
    vector<string> tmp2; // + or - 가 나온 후의 term_tail 저장할 임시 벡터(+,- 포함)
    int state = 0;
    for (const string &i: v) {
        if (i == "+" || i == "-") { // + or - 식별되면 상태 변수 : 1 하고 이 연산자를 tmp2에 push_back
            state = 1;
            tmp2.push_back(i); // 사실 + -는 무조건 termtail 이 받음
        } else if (state == 0) tmp1.push_back(i); // + or - 나오기 전 어떠한 value(variable 또는 constant)
        else tmp2.push_back(i); // + or - 한 번이라도 나온 뒤 termtail
    }
    return term( v: tmp1) + term_tail( v: tmp2); // 결국 계산될 R-value를 int exp = expresssion(tmp2); 이유로 return
}
```

- `double term_tail(const vector<string> &v)`: 수식의 항(terms) 사이의 곱셈과 나눗셈 연산을 처리합니다. `v`에서 이러한 연산자를 찾아 계산하고 결과 값을 반환합니다.
- `double term(const vector<string> &v)`: 수식의 항(terms)을 처리합니다. `v`에서 항을 구분하여 곱셈 또는 나눗셈 연산을 처리하고 결과 값을 반환합니다.

```
double term_tail(const vector<string> &v) {
    vector<string> tmp1;
    vector<string> tmp2;
    int state = 0;
    for (int i = 0; i < v.size(); i++) {
        if (i) {
            if (i == 1 && isoperator( str: v[i])) {
                error_code = max(error_code, 1);
                error_str += "# 중복 연산자(" + v[i] + ") 제거\t";
            } else {
                if (state) tmp2.push_back(v[i]);
                else if (v[i] == "+" || v[i] == "-") {
                    state = 1;
                    tmp2.push_back(v[i]);
                } else tmp1.push_back(v[i]);
            }
        }
    }
    if (v.empty()) return 0;
    else if (v.front() == "+") return term( v: tmp1) + term_tail( v: tmp2);
    else return term( v: tmp1) - term_tail( v: tmp2);
}

double term(const vector<string> &v) {
    vector<string> tmp1; // term은 내부적으로 더 하위 노드에서 (* or /)가 계산되어야 해서 * / 나오기 전  factor로 넘김
    vector<string> tmp2; // * or / 한 번 등장 이후의 모든 factor들의 결합 넘김
    int state = 0;
    for (const string &i: v) {
        if (i == "*" || i == "/") {
            state = 1;
            tmp2.push_back(i);
        } else if (state == 0) tmp1.push_back(i);
        else tmp2.push_back(i);
    }
    return factor( v: tmp1) * factor_tail( v: tmp2);
}
```

- `double factor_tail(const vector<string> &v)`: 항(factor) 내부에서 더 깊은 계층의 연산을 처리합니다. `v`에서 더 복잡한 연산을 처리하고 결과 값을 반환합니다.
- `double factor(const vector<string> &v)`: 수식의 기본 단위인 factor를 처리합니다. 이 함수는 식별자, 상수, 그리고 괄호로 묶인 표현식을 다룹니다.

```
double factor_tail(const vector<string> &v) {
    vector<string> tmp1;
    vector<string> tmp2;
    int state = 0;
    for (int i = 0; i < v.size(); i++) {
        if (i) {
            if (i == 1 && isoperator( str: v[i])) {
                error_code = max(error_code, 1);
                error_str += "# 중복 연산자(" + v[i] + ") 제거\t";
            } else {
                if (state) tmp2.push_back(v[i]);
                else if (v[i] == "*" || v[i] == "/") {
                    state = 1;
                    tmp2.push_back(v[i]);
                } else tmp1.push_back(v[i]);
            }
        }
    }
    if (v.empty()) {
        return 1;
    } else {
        double a = factor( v: tmp1);
        double b = factor_tail( v: tmp2);
        if (b != 0) {
            if (v.front() == "*") return a * b;
            else return 1.0 / a * b;
        } else return a;

    }
}
```

```
double factor(const vector<string> &v) {
    vector<string> tmp1;
    int state = 0;
    if (v.size() == 1) {
        string str = v.front();
        if (str[0] == '+') {
            str = str.substr( pos: 1);
        } else if (str[0] == '-') {
            str = str.substr( pos: 1);
            state = 1;
        }

        if (str[0] <= '9' && str[0] >= '0') {
            if (state) return -stod(str);
            else return stod(str);
        } else {
            if (ID_name.find( x: str) != ID_name.end()) {
                if (ID_name[str] == NAN) {
                    putNAN = 1;
                    return ID_name[str];
                } else {
                    if (state) return -ID_name[str];
                    else return ID_name[str];
                }
            } else {
                error_code = max(error_code, 2);
                putNAN = 1;
                error_str += "# 정의되지 않은 변수(" + str + ")가 참조됨\t";
                ID_name[str] = NAN;
                return 0;
            }
        }
    } else {
        for (const auto &i : string const & : v) if (i != "(" && i != ")") tmp1.push_back(i);
        return expresssion( v: tmp1);
    }
};
```

- `void print_result() const`: 파싱 및 계산 결과를 출력하는 함수로, 오류 코드에 따라 결과 또는 오류 메시지를 출력합니다.
- `void print_ID()`: 변수와 그 값들을 출력하는 함수로, `ID_name` 맵에 저장된 변수와 그 값들을 출력합니다.
- `bool isoperator(const string &str)`: 문자열이 연산자인지 확인하는 함수로, 입력된 문자열이 `+`, `-`, `*`, 또는 `/` 중 하나인 경우 `true`를 반환하고, 그렇지 않으면 `false`를 반환합니다. 중복 연산자를 처리하기 위해 만든 메소드 입니다.

```
    void print_result() const {
        if (error_code <= 0) cout << "(OK)";
        else cout << "(Warning)  " << error_str;
    }

    void print_ID() {
        cout << "Result ==>";
        for (const auto &i : pair<...> const & : ID_name) {
            if (i.second != NAN) cout << " " << i.first << ": " << i.second << ";";
            else cout << " " << i.first << ": Unknown;";
        }
    }

    bool isoperator(const string &str) {
        char arr[4][2] = { [0]: "+",  [1]: "-",  [2]: "*",  [3]: "/"};
        for (auto &i : char &[2] : arr)
            if (str == i) return true;
        return false;
    }
};
```