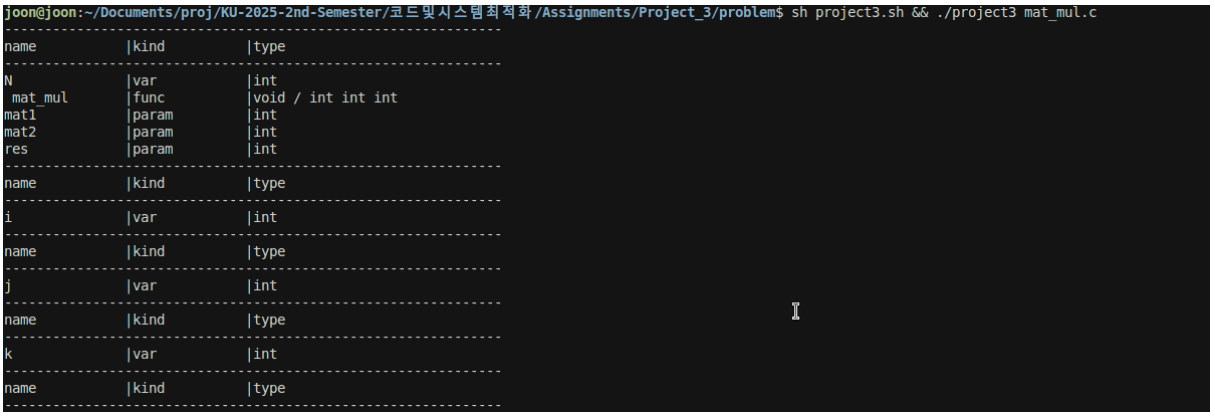
Project_3

2020170976 배연준

Problem1(Symbol table construction)

코드 실행 결과



컴파일러 작동 플로우

```
yacc -d project3.y
lex project3.l
cc lex.yy.c y.tab.c -o project3
```

1. `project3.y`을 통해 `parser` 코드를 생성한다. 이 때 `yyparse()`가 포함된 `y.tab.c`, 토큰 상수값이 정의된 `y.tab.h` 출력 파일이 생성된다.
2. `project3.l`을 통해 `scanner`코드를 생성한다. 이 때`yylex()`가 포함된 `lex.yy.c` 파일이 생성된다. 이는 입력된 c코드 텍스트 파일을 문자 단위로 분석해 `Yacc`가 이해할 수 있는 `token`으로 변환한다.
3. C 컴파일러 `cc`가 `lex.yy.c`, `y.tab.c`를 함께 링크하여 간단한 컴파일러 프로그램인 `project3`을 생성한다. 내부적으로 `yyparse()`함수가 실행되며, 필요 시 자동으로 `yylex()`를 호출해 입력을 처리한다.

구현된 함수 목록(symtab.h)

함수명	기능
`NewSymTab()`	새로운 symbol table 구조체 생성 및 초기화. DFS 알고리즘 통해 재귀 순회.
`AddSymTab()`	부모-자식 관계 설정
`NewSymbol()`	새로운 symbol 생성
`AddSymbol()`	symbol을 table에 추가
`ConstructSymTab()`	Parse tree를 순회하며 symbol table 구축. pre-order 방식의 전위 순회
`ProcessBody()`	함수 body 내 지역 변수 처리
`GetType()`	Type node에서 타입 정보 추출
`GetVarName()`	Variable node에서 변수명 추출
`ProcessDeclList()`	함수 매개변수 개수 세기
`AddParamsToScope()`	매개변수를 scope에 추가
`ProcessDeclListForVariables()`	지역 변수 선언 처리
`PrintSymTab()`	Symbol table을 재귀적으로 출력

Parse Tree 생성(project3.y)

NODE* create_token_node(const char* token_type, const char* token_value)

- 역할: 토큰 노드 생성(예: `ID: mat_mul`)

NODE* create_internal_node(const char* node_type, int child_count, ...)

- 내부 노드 생성 및 자식 노드 연결
- 가변 인자를 통해서 여러 자식 노드를 한 번에 추가

전체 구조는 아래와 같이 요약이 가능하다.

```
int main(int argc, char **argv) {
    // 1. 파일 열기 및 파싱
    yyin = fopen(argv[1], "r");
    yyparse();  // Parse tree 생성

    // 2. Symbol table 구축
```

```
    SYMTAB* root_symtab = NewSymTab();
    ConstructSymTab(root_symtab, head);

    // 3. Symbol table 출력
    PrintSymTab(root_symtab);

    // 4. 스코프 분석 및 타입 분석
    ScopeAnalysis(root_symtab, head);
    TypeAnalysis(root_symtab, head);

    fclose(yyin);
    return 0;
}
```

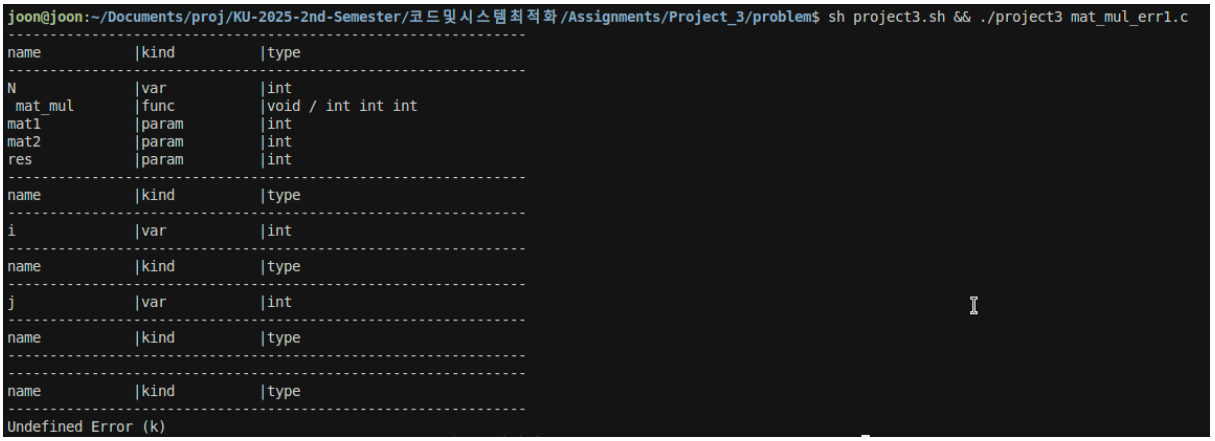
이렇게 생성된 계층 구조를 보면 아래와 같다.

```
Global Scope (root)
├─ N (var)
├─ mat_mul (func)
├─ mat1, mat2, res (param)
└─ Child Scope 1 (func_scope)
   ├─ i (var)
   └─ Child Scope 2
      ├─ j (var)
      └─ Child Scope 3
         └─ k (var)
```

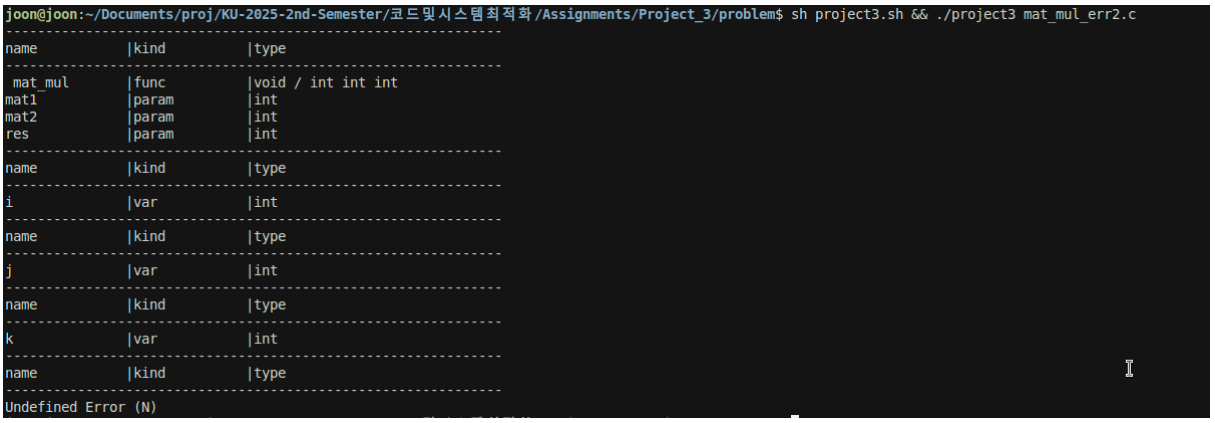
Problem2(Scope Analysis)

코드 실행 결과

- `matmul_err1.c`



- matmul_err2.c



작동 플로우

- problem 2는 미정의 변수(undefined variables) 오류를 검출하는데 초점을 맞추었다. Symbol Table 내부에 있는지 확인하고, 없으면 상위 symbol table에서 찾아보고, 없으면 그때 없는 변수의 이름을 출력하는 식이다.
- project3.y에서 symtab.h에 정의된 ScopeAnalysis 함수를 실행하면서 해당 오류를 찾아낸다.

구현된 함수 목록 및 사용(symtab.h)

함수명	기능	역할
ScopeAnalysis()	Scope 분석 메인 함수	전체 분석 프로세스 조율
FindFunctionDefinition()	함수 정의 노드 탐색	Parse tree에서 func_def 찾기
AnalyzeFunctionBody()	함수 body 분석	Statement별로 변수 사용 추적

함수명	기능	역할
`AnalyzeNodeForUndefinedVars()`	노드별 미정의 변수 검사	재귀적으로 모든 변수 검사
`FindVariableInAllScopes()`	전체 scope에서 변수 검색	Symbol table 전체 탐색
`ExtractVariableName()`	변수명 추출	Parse tree node에서 이름 추출

```
main()
  ↓
yyparse() - Parse tree 생성
  ↓
ConstructSymTab() - Symbol table 구축
  ↓
ScopeAnalysis() - Scope 분석 시작
  ↓
FindFunctionDefinition() - 함수 찾기
  ↓
AnalyzeFunctionBody() - 함수 body 분석
  ↓
AnalyzeNodeForUndefinedVars() - 각 노드 검사
  ↓
FindVariableInAllScopes() - 변수 존재 확인
  ↓
[변수 없으면]
fprintf(stderr, "Undefined Error (%s)\n", var_name)
```

`project3.y` 구현

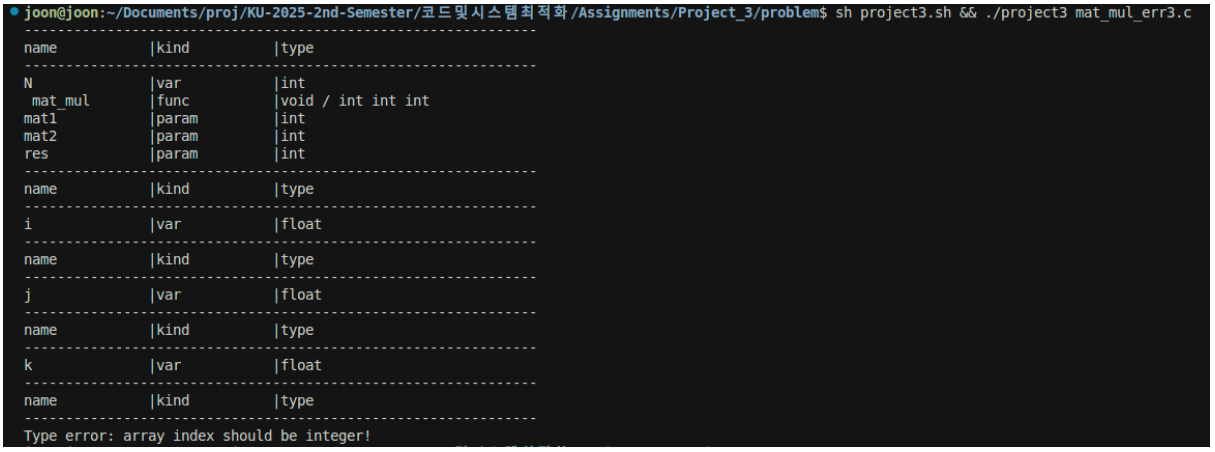
```
static inline void ScopeAnalysis(SYMTAB* symtab, NODE* head) {
    if (!symtab || !head) return; //head나 심볼 테이블 모두 없으면 수행하지 않는다.

    /* 이미 존재하는 에러 중복 방지 위한 배열 */
    char reported_errors[32][64];
    int num_reported = 0;

    /* 함수 내부 변수 사용 검사. parseTree를 DFS 방식으로 순회하며 T_FUNC_DEF
```

```
노드를 찾음 */
    NODE* func_def = FindFunctionDefinition(head);
    if (func_def) {
        NODE* body_node = func_def→child;
        while (body_node && strcmp(body_node→name, "body") != 0) {
            body_node = body_node→next;
        }

        if (body_node) {
                /* 내부 재귀적으로 분석하여 해당 body에 대하여 symbol table 전체 순
회. */
            AnalyzeFunctionBody(symtab, body_node, reported_errors, &num_r
eported);
        }
    }
    //...

    if (!already_reported) {
    fprintf(stderr, "Undefined Error (%s)\n", var_name); //여기서 정의되지 않은
변수 오류 출력.
    strcpy(reported_errors[*num_reported], var_name);
    (*num_reported)++;
}
}
```

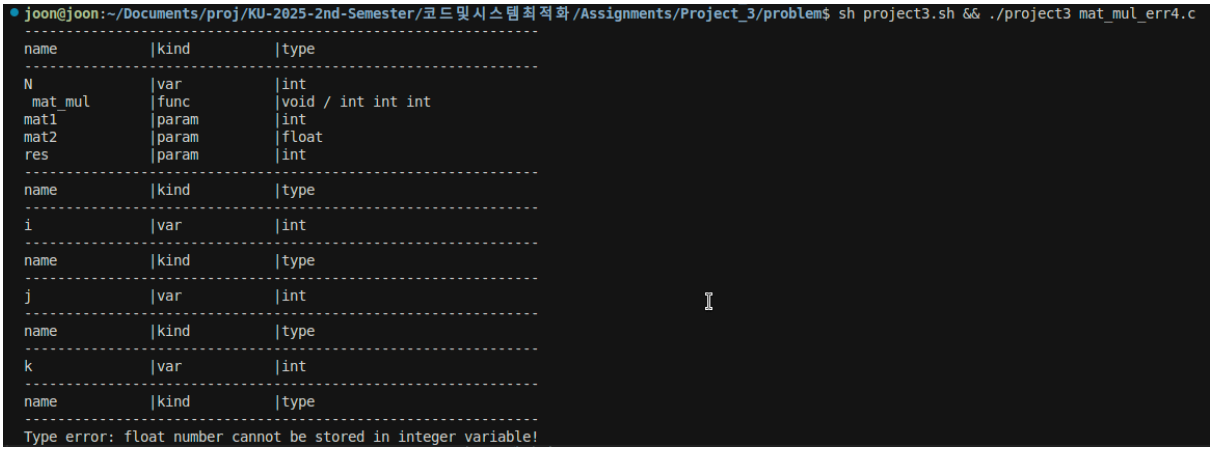
Problem3(Type Analysis)

코드 실행 결과

- `mat_mul_err3.c`



- mat_mul_err4.c



작동 플로우

- problem 3 는 타입 오류를 검출하는데 초점을 맞추었다. Symbol table의 타입 정보를 활용해, 우선은 간단한 2가지 경우의 에러만을 검출한다.

검출 대상 에러

1. **배열 인덱스가 정수가 아닌 경우**
   - mat_mul_err3.c : 변수 i , j , k 를 float 로 선언하고 배열 인덱스로 사용
2. **Float 값을 정수 변수에 저장하는 경우**
   - mat_mul_err4.c : float 배열 mat2 의 값을 int 배열 res 에 저장

구현된 함수 목록 및 사용(symtab.h)

함수명	기능	역할
TypeAnalysis()	타입 분석 메인 함수	전체 프로세스 조율

`GetVariableType()`	변수의 타입 조회	symbol table에서 변수의 타입 조회. (우선은 int, float, void만 찾고, 그 외는 -1 반환)
`CheckArrayIndexUsage()`	배열 인덱스 타입 검사	C언어의 배열 인덱스는 반드시 정수여야하고, float는 허용되지 않음. 여기서 사용된 방식은 하드코딩된 방식으로, 배열 인덱스로 사용된 i, j, k에 대해서 float인지 아닌지 여부를 확인하고, float라면 type error 출력.
`CheckArithmeticTypeMismatch()`	산술 연산 타입 불일치 검사	산술 연산에서 int와 float가 혼합된 경우 검사.
`AnalyzeFunctionBodyForTypes()`	함수 body 타입 분석	함수의 body 타입 검사
`FindVariableInAllScopes()`	전체 scope에서 변수 검색	Symbol table 전체 탐색

1. 배열 인덱스 검사

문제: 배열 인덱스로는 `float type`을 사용할 수 없음.

```
int i_type = GetVariableType(symtab, "i");
// i가 "float i"로 선언되었다면 → 2 반환
// i가 "int i"로 선언되었다면 → 1 반환
```

```
static inline void TypeAnalysis(SYMTAB* symtab, NODE* head) {
    if (!symtab || !head) return;

    /* i, j, k가 float인가? */
    int i_type = GetVariableType(symtab, "i"); // 2반환
    int j_type = GetVariableType(symtab, "j"); // 2반환
    int k_type = GetVariableType(symtab, "k"); // 2반환

    if (i_type == 2 || j_type == 2 || k_type == 2) { /* float type */
        fprintf(stderr, "Type error: array index should be integer!\n");
    }
    ...
}
```

2. 잘못된 값 저장

문제: `mat2[k][j]`는 float 타입인데, `res[i][j]` (int 타입)에 저장됨.

```
static inline void TypeAnalysis(SYMTAB* symtab, NODE* head) {
    ...

    /* mat2가 float인지 검사 */
    SYMBOL* mat2_var = FindVariableInAllScopes(symtab, "mat2"); // 2 반환.
    if (mat2_var && mat2_var→type[0] == 2) { /* float type */
        fprintf(stderr, "Type error: float number cannot be stored in integer var
iable!\n");
    }
}
```

즉, 이때의 플로우는 아래와 같다.

```
main()
  ↓
yyparse() - Parse tree 생성
  ↓
ConstructSymTab() - Symbol table 구축
  ├─ func_def 처리
  │   └─ AddParamsToScope()
  │       └─ "float mat2[][N]" 파싱
  │           └─ NewSymbol("mat2", 1, 2) 생성
  │               └─ {name:"mat2", kind:1, type:[2]}
  ↓
TypeAnalysis() - 타입 분석 시작
  ↓
FindVariableInAllScopes(symtab, "mat2")
  ├─ Global scope 검색
  │   └─ "mat2" 발견 → SYMBOL* 반환
  ↓
if (mat2_var→type[0] == 2)  // 애러 조건 만족
  ↓
fprintf(stderr, "Type error: float number cannot be stored in integer variabl
e!\n")
```