project2.y 코드 구현

1. 구현을 도와줄, 헬퍼 함수들을 Prologue 영역에 추가하였습니다.

```
NODE* create_token_node(const char* token_type, const char* token_value);
NODE* create_internal_node(const char* node_type, int child_count, ...);
```

- `create_token_node`: **단말 노드(Terminal Node) 또는 리프 노드(Leaf Node)**를 생성합니다. 렉서(Lexer)로부터 전달받은 토큰(예: ID, NUM, IF)과 그 값(예: `my_var`, `123`)을 받아 "TOKEN: value" 형태의 문자열을 이름으로 갖는 노드를 만듭니다. 이 노드들은 항상 트리의 가장 끝에 위치합니다.

`create_internal_node`: **비단말 노드(Non-terminal Node)**를 생성합니다. 문법 규칙의 좌변에 해당하는 노드(예: `func_def`, `statement`)를 만들고, 가변 인자(...)를 통해 전달받은 자식 노드들을 InsertChild 함수를 이용해 연결합니다. 이 함수는 트리 구조를 만들어나가는 역할을 합니다.

1. 핵심 아이디어: bottom-up

- Yacc는 Bottom-up 방식으로 파싱을 진행합니다.
- 환원이 일어나는 시점에 C로 정의된 액션(중괄호로 감싸진 코드)을 실행합니다.

1. 예시

```
assign_stmt:
    variable OP_ASSIGN al_expr
                    {
                    NODE* assign_tok = create_token_node("OP_ASSIGN", $2);
                $$ = create_internal_node("assign_stmt", 3, $1,
assign_tok, $3);
                    }
    ;
```

- 입력 코드: `a = 10;`가정

1. lexer가 코드를 토큰 스트림으로 분해: `ID("a")`, `OP_ASSIGN("=")`, `NUM("10")`, `SEMICOLON(";")`
2. `NODE* assign_tok = create_token_node("OP_ASSIGN", $2);` -> `OP_ASSIGN: =` 노드 생성
3. `$$ = create_internal_node("assign_stmt", 3, $1, assign_tok, $3);` -> `assign_stmt: ID = NUM` 노드 생성
4. 이제 `assign_stmt` 노드와 세미콜론 노드가 statement 규칙에 의해 환원되어 더 상위의 statement 노드와 연결됩니다.
5. 이렇게 트리가 완성되면, `WalkTree` 함수를 통해 트리를 순회하며 노드들을 출력할 수 있습니다.
6. 코드 구조

```
c_code:
    code
```

- `c_code`: 코드 전체를 나타내는 노드
- `code`: 정의 헤더 또는 함수 정의를 나타내는 노드

```
{
    head = create_internal_node("c_code", 1, $1);
    $$ = head;
}
```

- `head`: 코드 전체를 나타내는 노드
- ...

1. 최종 출력 procedure
    1. main함수에서는 `yyparse()`를 호출하여 파싱을 시작.
    2. 파싱 완료 후, 최상위 노드를 가리키는 head를 WalkTree함수에 전달.
    3. DFS 순회하여 모든 노드 정해진 형식에 맞게 출력.