코드 및 시스템 최적화

Project 3

2020171049 허가은

symtab.h의 구현에 대한 보고서

1. Symbol Table Construction

Symbol Table Construction을 수행하는 함수의 이름을 TreeRetrieve로 작성하였다.

mat_mul.c의 경우, 크게 define_header와 func_def 두 곳에서 symbol의 def가 일어난다. define_header의 경우 parse tree를 dfs 순회하면서 define_header가 발생하면, node->child->next->name가 ID: symbol name이기 때문에, 그 name값과 kind = 2, type = 1로 심볼을 만들어 심볼테이블에 넣도록 하였다.

func_def에서 정의되는 심볼은 function 이름과 관련된 심볼과, function body에서 발생하는 심볼이 있다. grammar에 의해 func_def는 type, ID, func_arg_dec, body등으로 구성된다. function의 타입과 이름을 결정하기 위해 func_def의 child들을 순회하며 type[0]과 function name을 funcsym symbol에 저장하였다.

func_arg를 순회하면 function의 argument type정보 뿐만 아니라, 그 아래 scope에 추가할 parameter 정보 또한 알 수 있기 때문에, ArgDerivation 함수를 사용하여 symbol table을 먼저 추가하고, 이 symbol table을 순회하여 function의 type 정보를 채우기로 하였다. 그림 1을 보면 func_arg_def는 여러 decl_list를 통과한 후 decl_init 노드에서 변수의 type과 이름을 알 수 있다. 따라서 ArgDerivation 함수를 통과하여 kind에 1을 갖고 DecLDerivation 함수로 decl_list node를 derivation할 수 있도록 하였다.

DecLDerivation 함수는 tree를 재귀적으로 탐색하다가, decl_init 노드를 발견하면 빈심볼을 만들어 DecIDerivation에 넘겨주고, DecIDerivation에서 심볼이 채워져 돌아오면 그것을 symtab에 채우는 일을 한다. DecIDerivation 함수는 decl_init 아래 노드들에서 변수의 type과 이름노드를 발견하고, 적절한 type과 FindName에서 return된 이름값을 바탕으로 심볼을 채운다.

이렇게 ArgDerivation을 통해 function 정의 심볼테이블 밑의 심볼테이블이 채워지면, func_arg_dec에서는 채워진 심볼테이블을 num_entry만큼 순회, funcsym의 type을 채우고 num_type갯수를 맞춘다.

마지막으로 body 노드를 만나면 BodyDerivation함수를 호출하여 아래 symtab들을 채운다.

BodyDerivation은 clause와 statement를 검출한다. clause는 if와 for 노드로 나뉘

어지는데, if노드에선 심볼이 생성되지 않으므로 for 노드가 발생하면 ForDerivation 함수를 호출한다. statement는 하위의 decl_list를 통해 심볼이 발생하므로 func_arg_def에서 사용한 것과 같은 방식으로 DecLDerivation를 호출하고, 이때 kind는 2로 지정한다.
for 노드가 발생하면, symbol의 정의는 body 노드 하위에 있다. node->next로 계속 이동하여 body 노드가 검출되면 새 symbol 테이블을 만들어 BodyDerivation을 호출하고, derivation을 마치고 돌아오면 AddSymTab으로 상위 테이블에 심볼테이블을 연결한다.

2. Scope Analysis
Scope in이 발생하여 하위 심볼테이블로 내려가야 하는 경우는 func_arg_dec노드를 만났을 때나, FOR: for노드를 만났을 때이다. ScopeAnalysis에서는 해당 노드를 발견하면 ScopeIn함수를 실행한다. ScopeIn함수에서는 symtab을 scopetab->child[0]로 바꿔준 후 다시 ScopeAnalysis를 수행한다. 또한 ScopeAnalysis에서 variable노드를 발견하면 FindSymbol 함수에 variable의 name을 넣어 실행하고, 리턴값이 NULL이라면 Undefined Error를 출력하고 프로그램을 종료한다.

3. Type Analysis
원래대로라면 Type Analysis를 위해 symbol table을 탐색할 때 올바른 scope를 찾아 거기서 analyze를 하는 것이 옳으나, 현재 err_3과 err_4에서는 undefined 오류가 발생하지 않고, symbol table 또한 일렬로 연결되어있으므로 Type analyze를 진행하기 전 findsymbol을 위한 symtab을 SymtabInitialize를 통해 가장 하위 symtab으로 지정한다.
err_3의 경우 index의 type이 올바른지 확인해야 한다. 그림 3에서 보듯이, index값의 경우 항상 LBRACKET: [의 next node에 위치한다. 따라서 parse tree의 재귀적 탐색을 통해 LBRACKET: [를 발견하였다면 node->next의 이름을 찾고, symbol table에서 이 node->next의 type을 찾아 타입을 검사, 1이 아니라면 Type error: array index should be integer 메시지를 출력하고 프로그램을 종료한다.
err_4의 경우 할당 연산자를 기준으로 left가 right보다 넓은 범위의 변수인지 확인해야 한다. 재귀적 탐색을 통해 assign_stmt를 검출하고, 이것의 첫 번째 child를 left, 세 번째 child를 right라고 한다. left는 variable node이기 때문에 VariableType함수를 이용하여 이름을 찾고 symtab에서 type을 찾아 리턴하고, right는 좀 더 복잡한 모습으로 이루어진 al_expr이기 때문에 AL_exprTypeDerivation함수를 호출한다. al_expr은 NUM과 variable, al_expr OP_ADD al_expr, al_expr OP_MUL al_expr으로 이루어져있으나 코드에서 right의 al_expr이 variable로 derivation되는 경우는 없다. NUM인 경우 type에 1을 리턴하고, OP_ADD와 OP_MUL이 있는 경우 연산자 양쪽의 타입을 찾아낸다. 그리고 그 중 큰 타입을 리턴한다.

다시 VariableType과 AL_exprTypeDerivation를 통해 할당 연산자 양쪽의 타입을 알아냈다면, float을 int에 저장하려는 경우에 float number cannot be stored in integer variable!에러를 출력하고 프로그램을 종료하였다.

4. Appendix

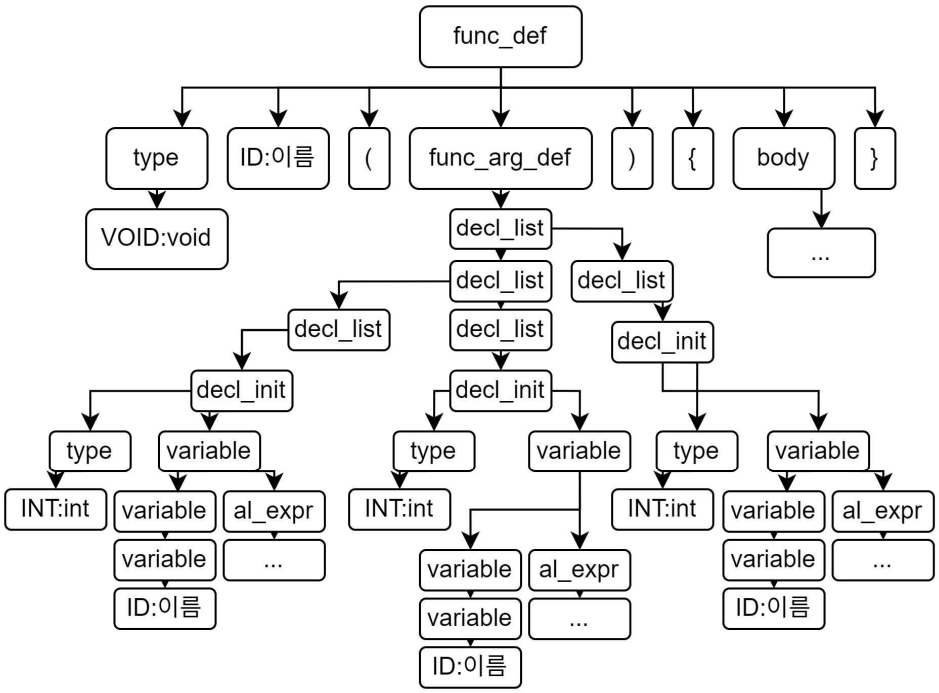

그림 1 function symbol을 찾기 위한 parse tree 구조도

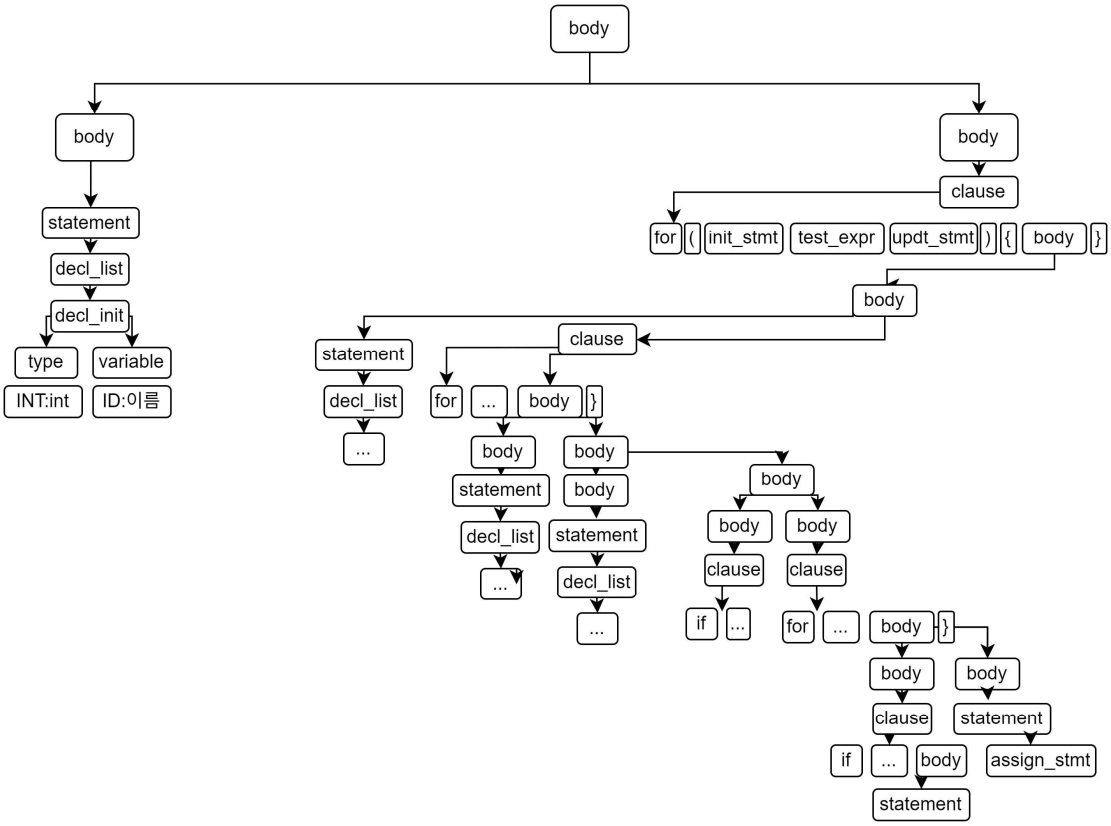

그림 2 function의 body symbol을 찾기 위한 parse tree 구조도

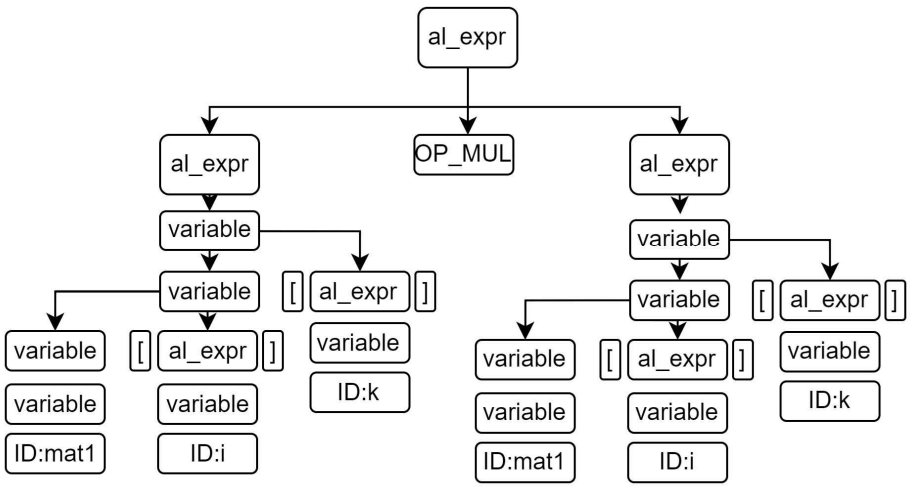

그림 3 Type Analysis를 위한 al_expr parse tree 구조도