Programming Assignment 4

Type Checking & Scope Checking

소프트웨어학부
박영훈 교수

개요

- PA4에서는 AST가 주어져있을 때, 각 변수 및 함수가 적절하게 사용되었는지 확인하기 위하여 Type checking과 Scope checking을 수행하는 프로그램을 작성한다.
- Type checking 및 scope checking 과정에서 오류가 발생하면 적절한 오류 메시지를 출력하고, 아무 오류가 없으면 아무 메시지도 출력하지 않는다!

- 파일 설명
  - 여러분의 계정에 `PA4` 디렉토리가 있을 것이고, 그 안에 다음 파일들이 있을 것이다. 이들 중 `tac.c`의 내용을 채우면 된다:

```
ast.h    symtab.h    tac.h    tac.c    util.a    Makefile    test.c
```

  - 현재 `PA4` 디렉토리에 있을 때, `PA3` 디렉토리로부터 `foo.l`과 `bar.y`를 다음 명령어를 이용하여 복사해온다:

```
cp ../PA3/foo.l .          cp ../PA3/bar.y .
```

  - `make` 명령어로 컴파일 하면 `check` 라는 파일이 만들어질 것이다. `test.c` 파일을 type checking 및 scope checking 하기 위해서는 다음과 같이 입력하면 된다:

```
./check test.c
```

작성 방법

- `bar.y`에서
  - 맨 위에 `#include "ast.h"`를 삭제하고, 대신에 `#include "tac.h"`를 추가한다.
  - `main` 함수에서 `ASTNode *root = 0;`을 추가하고, `printAST(pop(stack));`을 없앤 뒤, 그 자리에 `buildTAC(root = pop(stack));` 을 추가한다.
  - 다음 생성 규칙들을 없앤다

```
FuncDeclaration : Type FuncID '(' Params ')' ';'

FuncID : TID
```

  - 그리고, `FuncDeclararion` 을 다음과 같이 변환한다 (즉, `PA2`처럼 다시 돌아가는 것임)

```
FuncDeclaration : Type TID '(' Params ')' CompoundStmt
```

- `tac.c`에서
  - `tac.c` 에서 `travelNodes(ASTNode *node, SYMTAB *table)` 의 내용을 수정하면 된다.

ast.h 의 함수 설명

- ast.h에 다음 함수들이 추가되었음:

- TYPE type
  - TYPE_INT_FUNC, TYPE_FLOAT_FUNC, TYPE_VOID_FUNC, TYPE_ERROR가 추가 되었음.
- void setType(ASTNode *n, TYPE t)
  - Node n에 type을 t로 지정함.
- int getIVal(ASTNode *n)
  - Node n에 정숫값이 저장되어 있을 경우, 그 정숫값을 반환함.
- float getRVal(ASTNode *n)
  - Node n에 실숫값이 저장되어 있을 경우, 그 실숫값을 반환함.
- char* getSVal(ASTNode *n)
  - Node n에 문자열이 저장되어 있을 경우, 그 문자열의 주소를 반환함.
- void printType(TYPE* t)
  - 타입 이름을 화면에 출력하는 함수

Symbol Table

- 본 Programming Assignment에서 사용되는 Symbol table의 형태는 다음과 같다:
- 소스코드가 다음과 같다고 했을 때, `int c;` 가 끝나면 다음과 같은 symbol table이 만들어진다.

```
int a[10], b;
void foo(int a, float b[5]){
   int c;
   ...
```

Type	ID	Nick ID	Array size / # Param	Param 1	Param 2	...
int	c	c_2				
float[]	b	b_2	5			
int	a	a_2				
void	foo	foo_1	2	int	float[]	
int	b	b_1				
int[]	a	a_1	10			

symtab.h 의 함수 설명

- `typedef struct symtab_t SYMTAB`
  - Symbol table을 위한 stack 자료구조를 정의한 type.
- `SYMTAB *initSymTab(void)`
  - Symbol table을 초기화하는 함수. 이미 `buildTAC` 함수에서 call이 되어 있으므로 신경쓰지 않아도 됨.
- `void delSymTab(SYMTAB* table)`
  - Symbol table을 메모리에서 해제하는 함수. 역시, 이미 `buildTAC` 함수에서 call이 되어 있으므로 신경쓰지 않아도 됨.
- `void pushSymTab(SYMTAB* table)`
  - Stack 자료구조에서 symbol table을 생성하여 push할 때 사용하는 함수. Compound statement가 시작될 때 call하면 된다.
- `void popSymTab(SYMTAB* table)`
  - Stack 자료구조에서 symbol table을 pop하고, 메모리에서 제거하는 함수. Compound statement가 끝날 때 call하면 된다.

symtab.h 의 함수 설명

- `void addSym(SYMTAB* table, char* id, TYPE t)`
  - Symbol table에 symbol, 즉 변수나 함수를 추가해주는 함수. 변수를 추가하려면 `_VARDEC` node에서, 함수를 추가려면 `_FUNCDEC` node에서 하는 것이 좋다.
  - 변수 추가 예: `int a;`라고 선언되었다고 했을 때,

```
addSym(table, "a", TYPE_INT);
```

  - 함수 추가 예: `int foo(int a, float b){..}`라고 선언되었다고 했을 때,

```
addSym(table, "foo", TYPE_INT_FUNC);
```

- `void addSymArray(SYMTAB* table, char* id, TYPE t, int size)`
  - Symbol table에 배열 타입의 변수를 추가해주는 함수. 이 함수 역시 `_VARDEC`에서 call 하는 것이 좋다.
  - 변수 추가 예: `int a[10];` 이라고 선언되었다고 했을 때,

```
addSym(table, "a", TYPE_INT_ARRAY, 10);
```

- 위의 두 경우, 같은 scope에 이미 같은 이름의 ID가 존재하면 오류 메시지 발생시키고 종료하므로, 여러분이 구현할 필요는 없다.

symtab.h 의 함수 설명

- `void addParam(SYMTAB* table, char* id, TYPE t)`
  - Symbol table에서 마지막으로 추가한 함수에 매개변수를 추가하는 함수.
  - 반드시 마지막으로 추가한 symbol의 type은 `TYPE_INT_FUNC`, `TYPE_FLOAT_FUNC`, `TYPE_VOID_FUNC` 중 하나이어야 한다.
  - 앞 슬라이드의 `int foo(int a, float b){..}`라는 소스코드가 있으면 `_FUNCDEC` node에서 다음 순서대로 실행이 될 것이다:

```
addSym(table, "foo", TYPE_INT_FUNC);
addParam(table, "a", TYPE_INT);
addParam(table, "b", TYPE_FLOAT);
```

  - 만일, 매개변수의 type이 `void`이면 `addSym(table, 0, TYPE_VOID);` 라고 실행한다. 예를 들어, `int bar(void){...}` 라는 소스코드가 있으면 다음 순서대로 실행될 것이다:

```
addSym(table, "bar", TYPE_INT_FUNC);
addParam(table, 0, TYPE_VOID);
```

symtab.h 의 함수 설명

- `void addParamArray(SYMTAB* table, char* id, TYPE t, int size)`
  - Symbol table에서 마지막으로 추가한 함수에 배열 타입의 매개변수를 추가하는 함수.
  - `void bar(int a[10]){ ... }` 라는 소스코드가 있으면 `_FUNCDEC` node에서 다음 순서대로 실행될 것이다:

```
addSym(table, "bar", TYPE_VOID_FUNC);
addParamArray(table, "a", TYPE_INT_ARRAY, 10);
```

- `TYPE getThisFuncType(SYMTAB* table)`
  - 현재 함수의 반환 타입을 반환하는 함수. 즉, 현재 node에서 조상 node 중에 `_FUNCDEC` 가 있으면, 그 함수의 반환 타입을 반환한다. 만일 `_FUNCDEC` 가 없으면 `TYPE_ERROR`를 반환한다. 참고로, 현재 함수의 type이 `TYPE_INT_FUNC`이면 `TYPE_INT`가 아니라 `TYPE_INT_FUNC`를 반환한다.

- `TYPE getSymType(SYMTAB* table, char* id)`
  - Symbol table에서 해당 `id`의 타입을 반환하는 함수. 만일 symbol table에서 같은 이름을 가진 `id`가 둘 이상 있다면, 최근에 추가된 `id`의 타입을 반환한다.

symtab.h 의 함수 설명

- `TYPE getParamType(SYMTAB* table, char* func, int index)`
  - Symbol table에서 이름이 `func`인 함수의 `index`번째 parameter의 type을 반환하는 함수. 이 때, 맨 왼쪽의 parameter를 0번째라고 한다.
  - 만일, 해당 함수가 없거나, `index`의 값이 해당 함수의 parameter 개수 이상일 때는 `TYPE_ERROR`를 반환한다.
  - 사용 예:
    - `int foo(int a, float b[10]){ ... }` 라고 정의되어 있을 때,
      - `getParamType(table, "foo", 1)` 을 실행하면 `TYPE_FLOAT_ARRAY`를 반환.
      - `getParamType(table, "foo", 2)` 를 실행하면 `TYPE_ERROR`를 반환.
    - `int bar(void){ ... }` 라고 정의되어 있을 때
      - `getParamType(table, "bar", 0)` 을 실행하면 `TYPE_VOID` 를 반환.
      - `getParamType(table, "bar", 1)` 을 실행하면 `TYPE_ERROR`를 반환
- `int getNumOfParam(SYMTAB* table, char* func)`
  - 이름이 `func`인 함수의 매개변수 개수를 반환하는 함수. 만일 매개변수가 없으면 0을, 함수 자체가 없으면 -1을 반환한다.
- `void printSymTab(SYMTAB* table)`
  - Symbol table을 화면에 출력하는 함수. 출력 포맷은 5쪽과 같다.

symtab.h 의 함수 설명

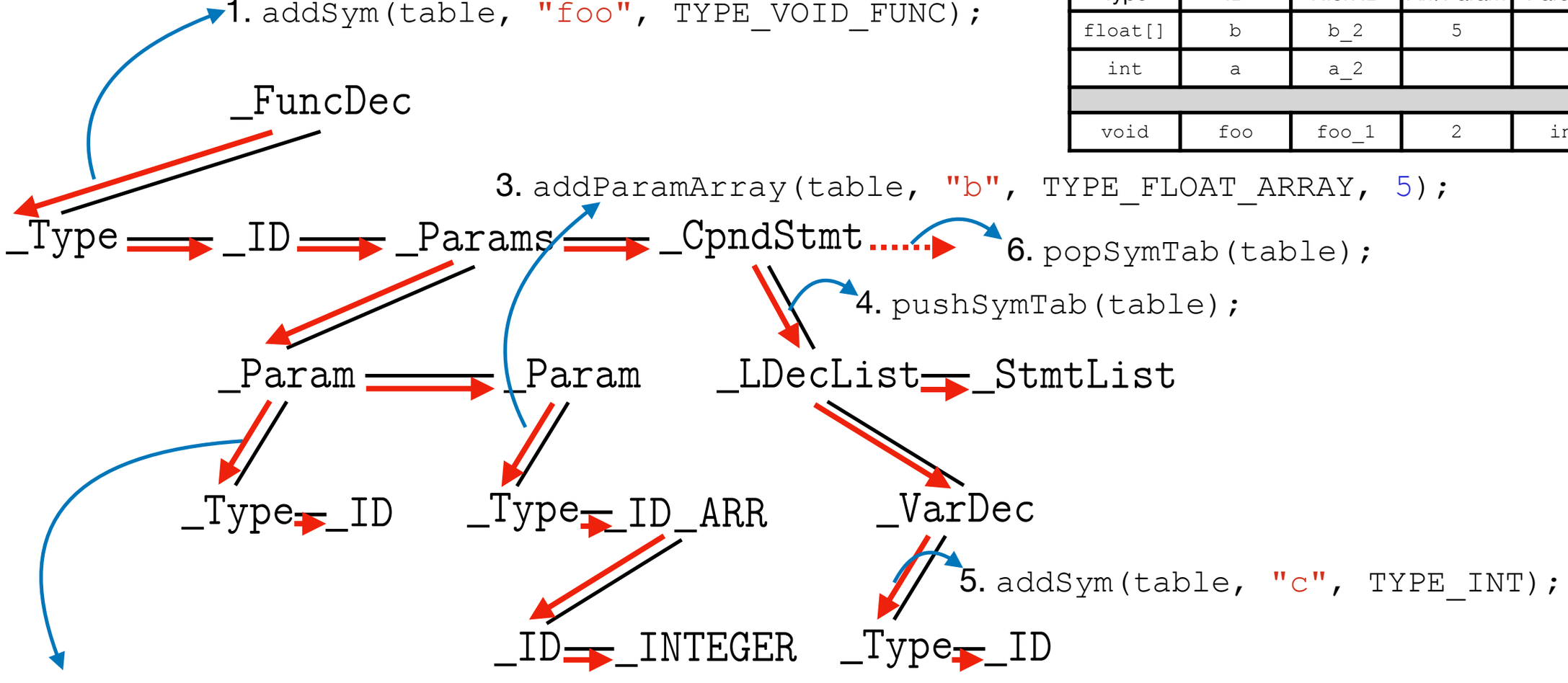
- `void pushSymTab(SYMTAB* table)` 함수의 활용
  - `pushSymTab` 함수를 call하면 새로운 symbol table이 만들어져서 push된다. `pushSymTab` 함수가 불리기 직전에 함수가 symbol table에 올라왔으면 그 함수의 parameter들을 새로운 symbol table에 자동으로 올려준다.
  - 사용 예:

```
void foo(int a, float b[5]){
    int c;
    ...
```

Type	ID	Nick ID	Arr/Param	Param 1	Param 2
void	foo	foo_1	2	int	float[]

`pushSymTab(table);` 수행 후

Type	ID	Nick ID	Arr/Param	Param 1	Param 2
float[]	b	b_2	5		
int	a	a_2			
void	foo	foo_1	2	int	float[]



tac.c의 함수 설명

- `void buildTAC(ASTNode *root)`
  - Symbol table을 초기화 하고, 아래의 travelNode를 실행하는 함수로, 여러분이 수정할 필요는 없음.

- `void travelNodes(ASTNode *node, SYMTAB *table)`
  - 여러분이 주로 작성해야 할 함수. Depth First Search 기반의 함수로, 크게 다음과 같이 구성되어 있다:
    1. `switch-case`문
    2. `if(getChild(node)) travelNodes(getChild(node), table);`
    3. `switch-case`문
    4. `if(getSibling(node)) travelNodes(getSibling(node), table);`
  - 위의 1번과 3번을 채우면 된다. 1번은 해당 노드인 node를 방문하자마자 실행될 코드들이고, 3번은 node를 나가기 직전에 실행될 코드들이다.
  - 만일 필요하다면, 새로운 변수를 선언해도, 함수를 변형해도 된다. (심지어 다 지우고 여러분이 스스로 짜도 된다)

- 만일 추가 함수가 필요하다면 `tac.h`에 선언하고 `tac.c`에 정의하여 써도 된다.

Your Missions (Scope Checking)

- 새로운 변수가 선언되면 symbol table에 변수를 추가한다
- 새로운 함수가 정의되면 symbol table에 함수를 추가한다
- compound statement가 시작되면 새로운 symbol table을 push하고, 끝나면 symbol table 하나를 pop한다.
- 변수나 함수가 사용되면 symbol table에서 그 변수나 함수가 있는지 찾아본다. 만일 없으면 다음과 같은 에러를 발생시키고 종료한다 (대괄호는 출력할 필요 없음):

```
[ID name] is not declared!
```

Your Missions (Type Checking)

- `a` **op** `b`에서 (단, `a`와 `b`는 operand, **op**는 operator) `a`와 `b`의 타입이 둘 다 `int`이거나 둘 다 `float`인지 확인한다. 또한, `-a` 에서 (단, `a`는 operand), a가 `int`이거나 `float`인지 확인한다.
  - 위의 조건을 만족시키지 않은 경우 아래 메시지를 발생시키고 종료한다. 또한, 연산자가 %일 때는 `a`와 `b` 둘다 `int`이어야 하는데, 그렇지 않으면 마찬가지로 에러를 발생시키고 종료한다:

```
Type error in expression!
```

- `if (Expr)` 에서 `Expr`가 `int` 타입인지
  - 만일 `int`가 아니면 다음과 같은 에러 메시지를 발생시키고 종료한다:

```
Type error in if statement!
```

- `while (Expr)` 에서 `Expr`가 `int` 타입인지
  - 만일 `int`가 아니면 다음과 같은 에러 메시지를 발생시키고 종료한다:

```
Type error in while statement!
```

- `for (Expr1; Expr2; Expr3)` 에서 `Expr2`가 `int` 타입인지
  - 만일 `int`가 아니면 다음과 같은 에러 메시지를 발생시키고 종료한다:

```
Type error in for statement!
```

Your Missions (Type Checking)

- `switch(Expr)`에서 `Expr`가 `int` 타입인지
  - 만일 `int`가 아니면 다음과 같은 에러 메시지를 발생시키고 종료한다:

```
Type error in switch statement!
```

- `return` 다음에 오는 값의 타입이 정의되어 있는 함수의 반환값의 타입과 일치하는지
  - 현재 함수가 `void` 타입인데 반환 값이 있을 때, 현재 함수가 `int`나 `float` 타입인데 반환값이 없거나 반환값의 타입이 다르면 다음과 같은 에러 메시지를 발생시키고 종료한다:

```
Return type error!
```

- 함수를 call할 때
  - 만일 인자의 개수가 적거나 많으면 다음과 같은 메시지를 발생시키고 종료한다:

```
Too few arguments!
Too many arguments!
```

  - 만일 인자의 타입이 안 맞으면 다음과 같은 메시지를 발생시키고 종료한다:

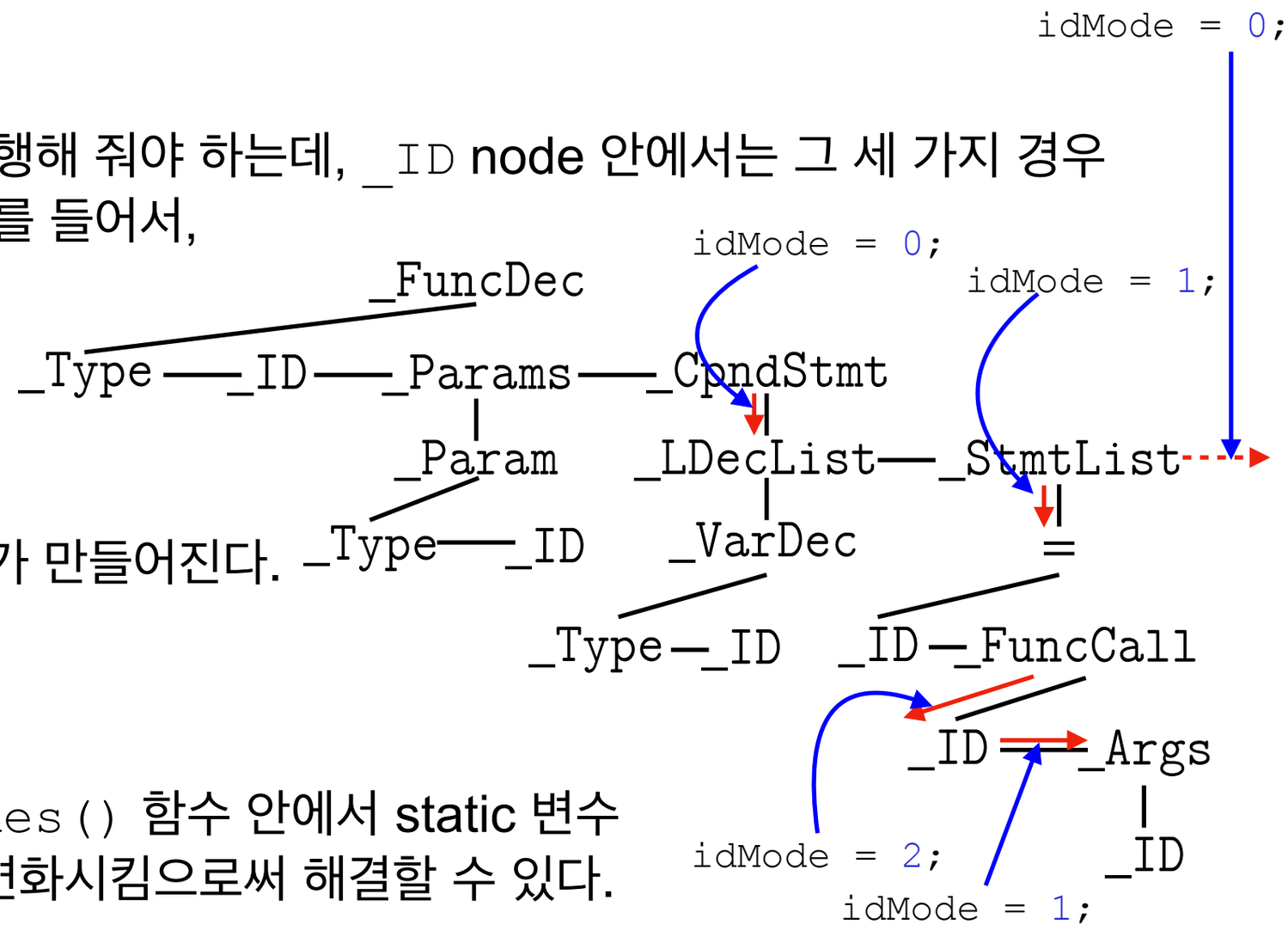
```
Type mismatch in function call!
```

  - 이 때, 오류 종류가 둘 이상이면 (ex. 타입도 안 맞고 개수도 안 맞을 때) 둘 중 아무 메시지나 하나 출력하면 된다.

TIP

- `_ID` node를 다음과 같은 세 가지 이유 중 하나로 거쳐갈 것이다.
  - 변수나 함수를 선언할 때
  - 수식에 변수를 이용할 때
  - 함수 call할 때
- 위의 세 가지 경우마다 다르게 수행해 줘야 하는데, `_ID` node 안에서는 그 세 가지 경우를 구분해주기 어려울 것이다. 예를 들어서,

```
void foo(int a){
   int b;
   b = bar(a);
}
```



- 라는 코드는 오른쪽과 같이 AST가 만들어진다.
- 그러면 전역변수나 `travelNodes()` 함수 안에서 static 변수를 만들어, 그 값을 상황에 맞게 변화시킴으로써 해결할 수 있다.
  - 오른쪽 예에서는 그 변수 이름을 `idMode`라고 할 때, 선언할 때는 0, 수식에 이용할 때는 1, 함수 call할 때는 2로 둔다.