Project3

Project 개요

- 간단한 C 컴파일러 구현
  - Project1 – Lexical Analysis
    - Input C code를 읽어 각 token 인식
  - Project2 – Syntax Analysis
    - Token input을 토대로 parse tree 구성 (bottom-up parsing)
    - Token들이 C 문법에 맞게 구성되어 있는지 체크
  - **Project3 – Semantic Analysis**
    - Input C code가 “의미”에 맞는지 체크
    - E.g. type check, use of undeclared variables

Project3 개요

- Semantic Analysis 구현

```
int main()  {
  int int a;     // Syntactic : Error
  int b;         // Syntactic : OK, Semantic : OK
  b = “abc”;  // Syntactic : OK, Semantic : Error
}
```

- 각 variable에 대한 정보(name, type, scope)를 저장하는 symbol table 구현
- subc.y 문법의 각 terminal 과 nonterminal 사이 적절한 위치에 action(C코드) 삽입해서 symbol table 활용해 semantic error 체크
- 에러가 발견될 경우 메시지 출력

String 연산

- String Copy
  - malloc : 동적 메모리 할당
  - strlen : string 의 길이
  - strcpy : malloc 통해 할당된 공간에 string 길이만큼 copy

```
char* name = “abc”;
char* new_str = malloc(strlen(name) + 1);
strcpy(new_str, name);
```

- String Compare
  - strcmp : 두 문자열을 끝까지(null문자) 사전 순으로 비교
  - **두 문자열이 동일하면 -> 0 반환**

```
printf("%d\n", strcmp("apple", "banana"));  // 음수 반환
printf("%d\n", strcmp("hello", "hello"));       // 0 반환
printf("%d\n", strcmp("zoo", "apple"));       // 양수 반환
```

Lex (subc.l)

- Comment (주석) 지원
  - /* */ 형태의 주석 지원 (skeleton code 에 이미 반영됨)
  - **중첩된 주석은 고려하지 않음**
  - Lex 의 mode 활용 (INITIAL/COMMENT mode)
  - INITIAL mode: 별도 모드 지정없는 경우 기본 상태로 자동 지정
  - %x COMMENT
    - Exclusive 모드로 COMMENT 선언
    - COMMENT 모드에서 <COMMENT> 로 명시된 규칙만 작동하고 그 외 규칙은 무시

```
%x COMMENT   // exclusive 상태 COMMENT 선언

%%

"/*"                    { BEGIN(COMMENT); }  // 주석 시작 → COMMENT 모드 진입

<COMMENT>"*/“     { BEGIN(INITIAL); }      // 주석 끝 → 초기 상태 복귀
```

Lex (subc.l)

- get_lineno()
  - 현재 컴파일 진행 중인 소스 코드의 line number 가져오는 함수
  - Lexer 에서 newline (\n) 스캔 할 때 마다 line number 1 증가
  - Skeleton code 에 이미 반영되어 있음

- File name
  - Yacc (subc.y) 에서 semantic error 메시지 출력 시 현재 line number (get_lineno 함수 이용) 및 input file 이름 같이 출력 필요
  - subc.l 내부 main 함수에서 argv[1] 통해 file name 정보 알 수 있음
    - E.g. argv[1] 의 file name 을 global variable 에 복사 (string copy)
  - **구현 필요! -> 방식은 자유**

Lex (subc.l)

- String 연산 시 주의사항
  - 항상 문자열 끝에 null 문자('\0') 고려
  - strcpy, strcmp 의 경우 null 문자 포함
  - strlen 의 경우 null 문자 제외한 문자열 길이 반환

- Lex 에서 제공하는 Lexeme 정보
  - 현재 인식한 토큰의 문자열(lexeme) 및 문자열 길이 정보
  - **yytext**: 현재 매치된 lexeme 문자열 가리키는 char* 포인터 (null 문자 포함)
  - **yyleng**: 현재 매치된 lexeme 문자열 길이 (null 문자 제외)
  - yytext 및 yyleng 은 각 토큰 인식할 때마다 값이 덮어쓰기 되기 때문에 lexeme 문자열을 복사하여 저장 필요

Yacc (subc.y)

- %empty
  - ε (empty string) 을 명시적으로 표현하기 위한 기호

```
ext_def_list
  : ext_def_list ext_def
  | %empty
  ;
```

```
ext_def_list -> ext_def_list ext_def
ext_def_list -> ε
```

Yacc (subc.y)

- Action
  - C 코드로 grammar production 의 RHS 에 삽입
  - 해당 production parsing 중에 수행되는 코드
  - 주로 production 의 끝에 삽입되어서 해당 production reduce 될 때 수행

- Action variable
  - 각 production 에 속한 grammar symbol 들은 value 를 가질 수 있음
  - **$i** 형태로 action code 에서 할당 및 접근 가능
  - Lexer 에서 인식한 토큰의 값은 **yylval** 통해 Parser로 전달 가능

```
E :  E '+' E  {$$ = $1 + $3;}
   | E '*' E  {$$ = $1 * $3;}
   | num    {$$ = $1;}  // lexer 에서 yylval로 num 토큰값 (num의 정수값) 전달 가능
   ;
```

Yacc (subc.y)

- **yylval** variable
  - Lexer 에서 Parser로 토큰값을 전달할 때 사용하는 variable
  - yylval 타입은 별도 지정 없는 경우 int 타입
  - yacc 에서 지정한 %union 과 연동하여 yylval 에서 다양한 타입의 값 전달 가능
  - yacc file (.y) 의 declarations 에 %union 지정, 그리고 해당 토큰값의 타입도 지정

Yacc file (.y)

```
%union {
   int ival;
}

%token <ival> NUM  // NUM 토큰값의 타입을 ival(int)로 지정
```

Lex file (.l)

```
[0-9]+   {
           yylval.ival = atoi(yytext);  // 정수 값을 yylval에 저장, 전달
           return NUM;                  // NUM 토큰 반환
         }
```

Yacc (subc.y)

- 정수 계산기

Yacc file (.y)

```
%union {
   int ival;
}

%token <ival> NUM  // NUM의 토큰값 타입 지정
%type <ival> expr     // action에서 사용하는 symbol 값의 타입 지정

%%

program:
   expr { printf("Result = %d\n", $1); }
;

expr:
   expr '+' expr { $$ = $1 + $3; }
  | expr '-' expr { $$ = $1 - $3; }
  | NUM          { $$ = $1; }   // 받은 NUM 토큰값(정수)을 $1로 접근
;
```

Lex file (.l)

```
%%

[0-9]+  {
            yylval.ival = atoi(yytext);  // 정수 값을 yylval에 저장
            return NUM;
          }

[+\-]     return yytext[0];  // '+' 또는 '-' 자체를 반환

[ \t\n]+   ;              // 공백 무시

%%
```

Yacc (subc.y)

- Mid-Rule Action
  - Production rule의 사이에 action 삽입 및 수행 가능

```
A
  : B          {$$ = $1 + 1;}
    C          {$$ = $2 + $3;}
  |D
  ;
```

  - Mid-rule action의 경우 바로 이전 component ($i)만 접근 가능
  - Mid-rule 자신도 하나의 component ($$) 가짐
  - Mid-rule 이후 action 은 mid-rule component를 동일한 방식($i)으로 접근 가능
  - Production rule 의 LHS component 값은 rule 끝에서만 접근 및 할당 가능

Yacc (subc.y)

- Mid-Rule Action
  - 실제 구현에선 Mid-Rule 위해 Yacc 에서 임의의 terminal 과 production 을 생성

```
A
  : B          {$$ = $1 + 1;}
    C          {$$ = $2 + $3;}
  |D
  ;
```

```
A
  : B M1 C      {$$ = $2 + $3}
  |D
  ;
M1 : %empty {$$ = $0 + 1;}
```

C ($3)
M1 ($2)
B ($1)
. . .

Parsing Stack

Yacc (subc.y)

- Mid-Rule Action Conflict
  - Mid-Rule 위해 terminal 과 production 이 추가되므로, 새로운 Conflict 발생 가능
    - E.g. shift/reduce conflict

```
A
  : {do_something();} BEGIN decls stmts END
  | BEGIN stmts END
  ;
```

```
A
  : M BEGIN decls stmts END
  | BEGIN stmts END
  ;
M : %empty {do_something();}
```

    - M 으로 reduce Or shift empty string?

Yacc (subc.y)

- Solution for Mid-Rule Action Conflict
  - Mid-Rule 을 첫 nonterminal/terminal 이후에 삽입

```
A
  : BEGIN {do_something();} decls stmts END
  | BEGIN stmts END
  ;
```

```
A
  : BEGIN M decls stmts END
  | BEGIN stmts END
  ;
M : %empty {do_something();}
```

  - 다른 Solution

```
A
  : M BEGIN decls stmts END
  | M BEGIN stmts END
  ;
M : %empty {do_something();}
```

Project3
Semantic Analysis

Semantic Check List

- Undeclared Variables & Functions
- Re-declaration
- Type Checking
- Structure & Structure pointer declaration
- Function Declaration

Symbol (Name)

- Semantic Analysis 는 Parsing 과정(syntactic analysis) 중에 같이 진행
- Semantic Analysis의 핵심은 Symbol (name)
  - Variable name
  - Function name
  - Struct name 등
  - **Name** 은 **meaning** 을 담고 있음
  - Meaning이 맞는지 체크하는 것이 Semantic analysis
    - 이전에 선언된 variable 이 사용되고 있는지 (Undeclared variable)
    - 이전에 선언된 variable이 재선언 되고 있는지 (Redeclaration)
    - 연산에 사용되는 variable의 타입이 올바른지 (type check)
    - 호출하는 Function 의 parameter, return 타입이 맞는지

=> name 관련 정보를 담는 data structure 필요! (symbol table)

Scoped Symbol Table

- Scope (Block)
  - C 언어는 Scope 기반 Variable 선언 및 할당
  - Scope 별로 동일한 이름의 Variable 각각 선언 및 사용 가능
  - 각 variable 은 가장 가까운 scope 의 declaration을 따름

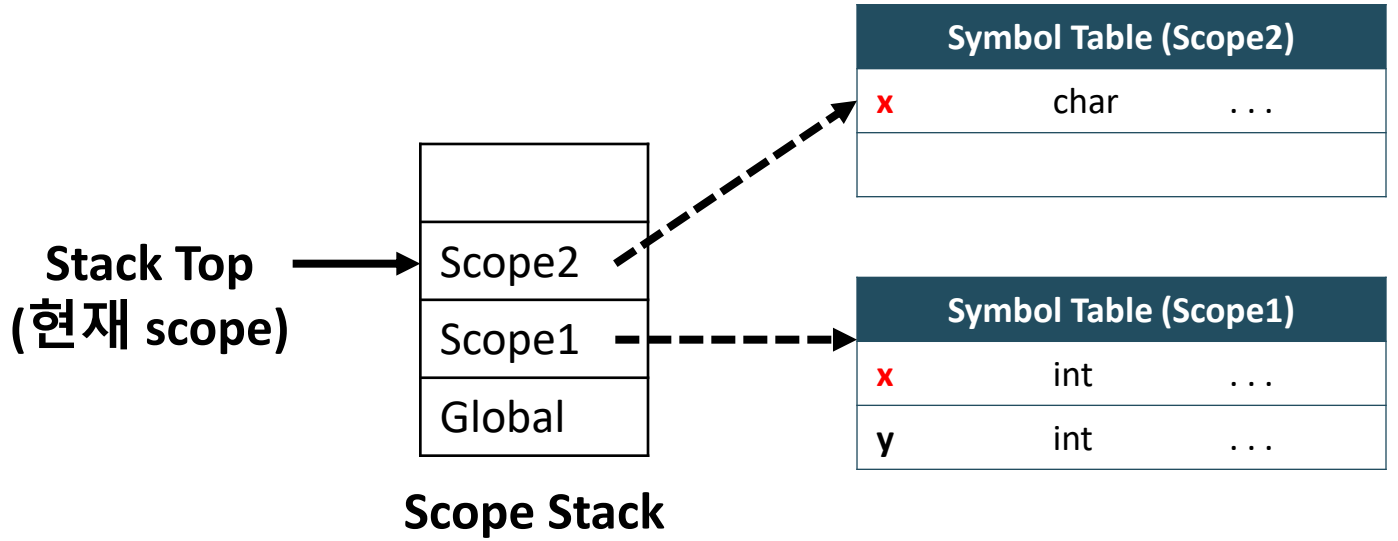
```
int x;
int y;
{
   char x;
   x = y;  /* char = int */
}
x = ..     /* int */
```

  - variable에 대한 정보(name, type, scope)를 저장하는 scoped symbol table 필요

Scoped Symbol Table

- Stack of symbol tables
  - 각 scope 별로 하나의 symbol table 을 가진 형태
  - 새로운 scope 만나면 symbol table 생성하고 stack 에 추가
  - Scope 이 끝나면 stack top에 있는 symbol table 제거(pop)
  - Variable 정보를 찾기 위해 stack top 부터 차례대로 탐색

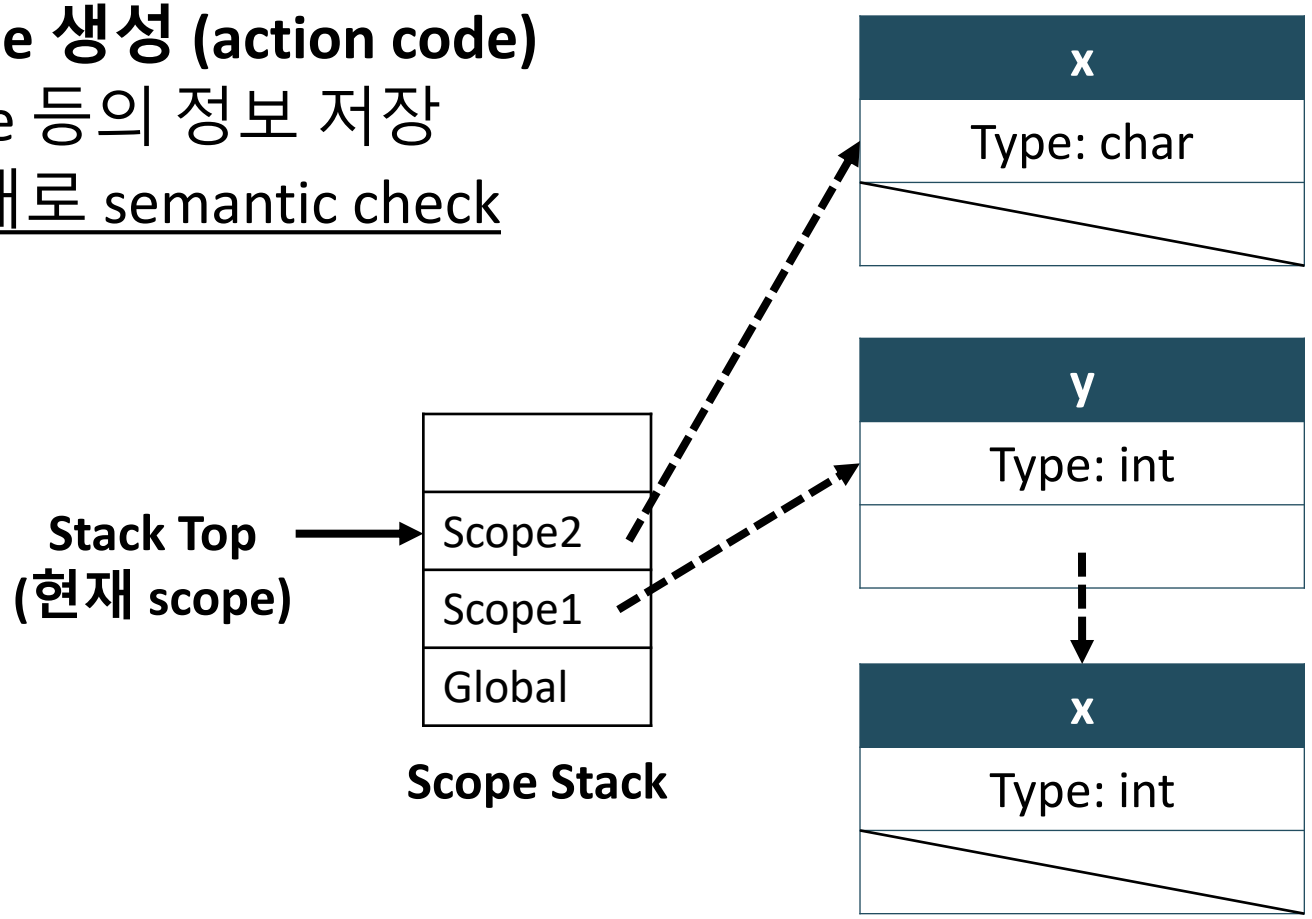
```
int x;
int y;
{
   char x;
   x = y;  /* char = int */
}
x = ..     /* int */
```



Scoped Symbol Table

- Stack of symbol tables 구현
  - 구현의 편의를 위해 linked list 형태로 구현 추천(자유롭게 구현 가능!!)
  - **Parsing 중에 symbol table 생성 (action code)**
  - 각 variable 의 name, type 등의 정보 저장
  - 저장된 symbol table 토대로 semantic check

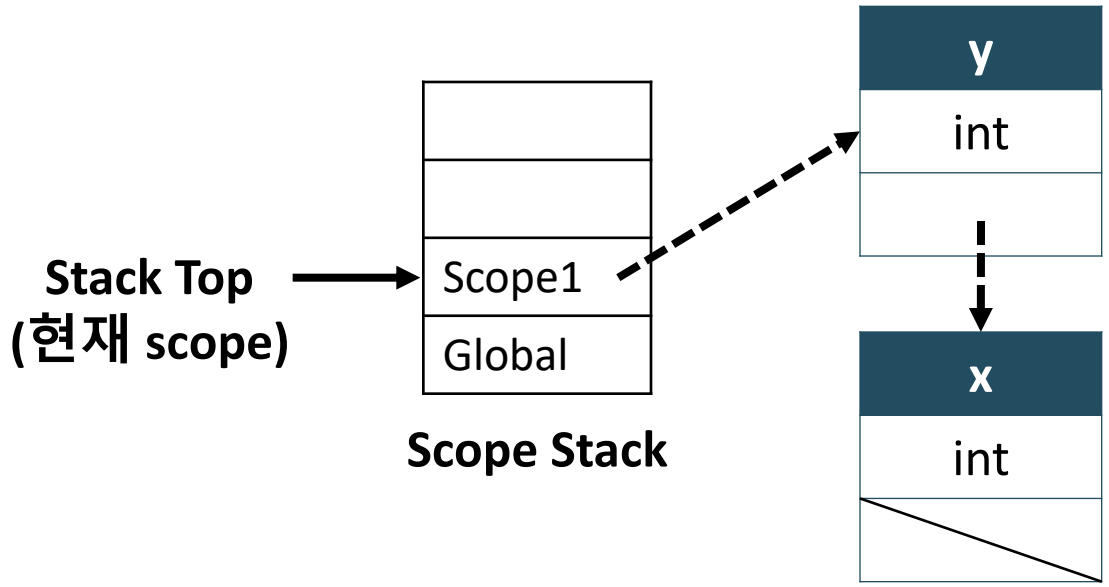
```
int x;
int y;
{
   char x;
   x = y;  /* char = int */
}
x = ..     /* int */
```



Scoped Symbol Table

- Stack of symbol tables 동작 과정
  - Variable declaration 마다 symbol table 에 해당 variable 정보 추가

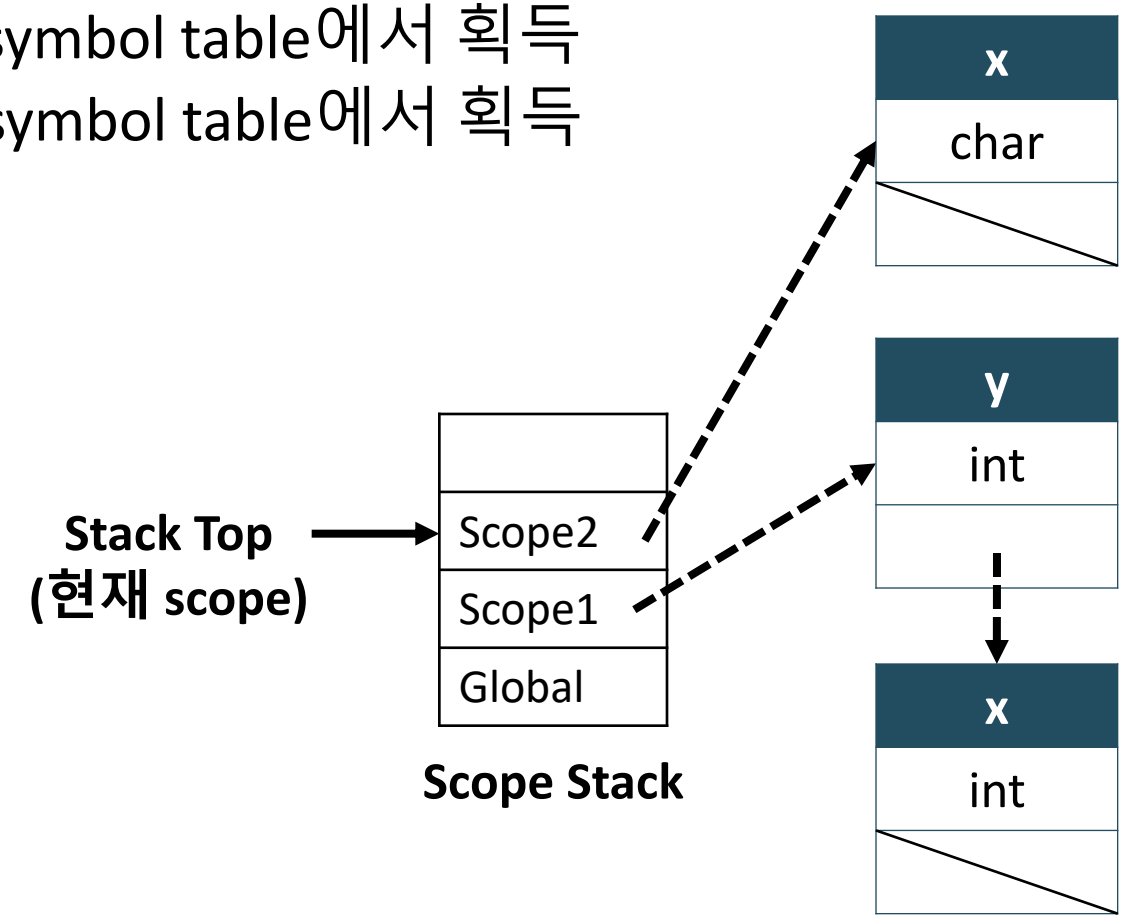
```
int x;
int y;
{
   char x;
   x = y;  /* char = int */
}
x = ..     /* int */
```



Scoped Symbol Table

- Stack of symbol tables 동작 과정
  - 새로운 scope 만나면 scope 및 symbol table 추가
  - x 정보는 현재 scope의 symbol table에서 획득
  - y 정보는 이전 scope의 symbol table에서 획득

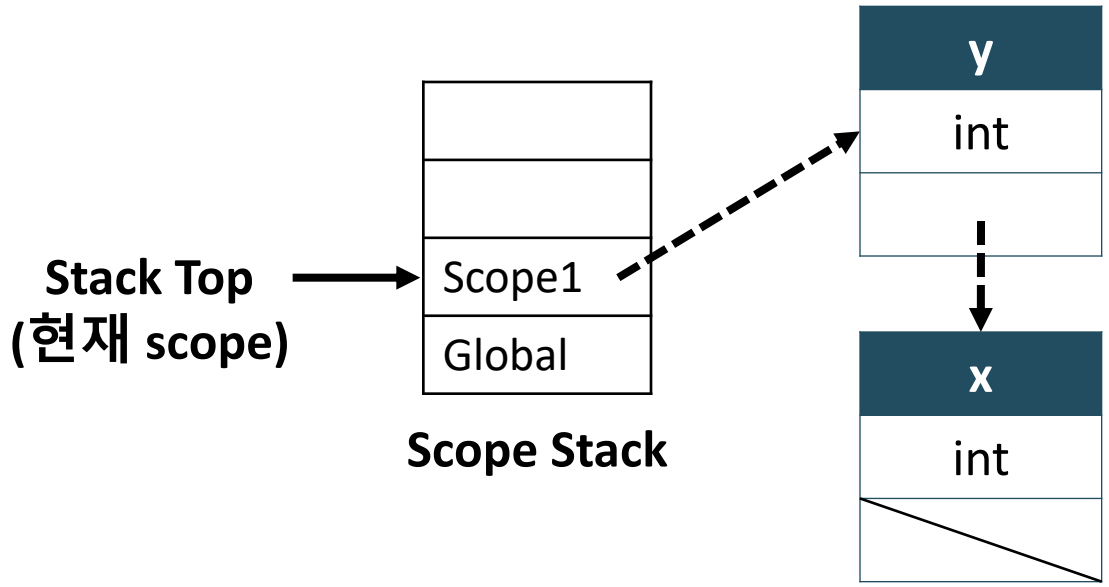
```
int x;
int y;
{
   char x;
   x = y;  /* char = int */
}
x = ..     /* int */
```



Scoped Symbol Table

- Stack of symbol tables 동작 과정
  - Scope 이 끝나면 현재 scope 및 해당 symbol table 제거

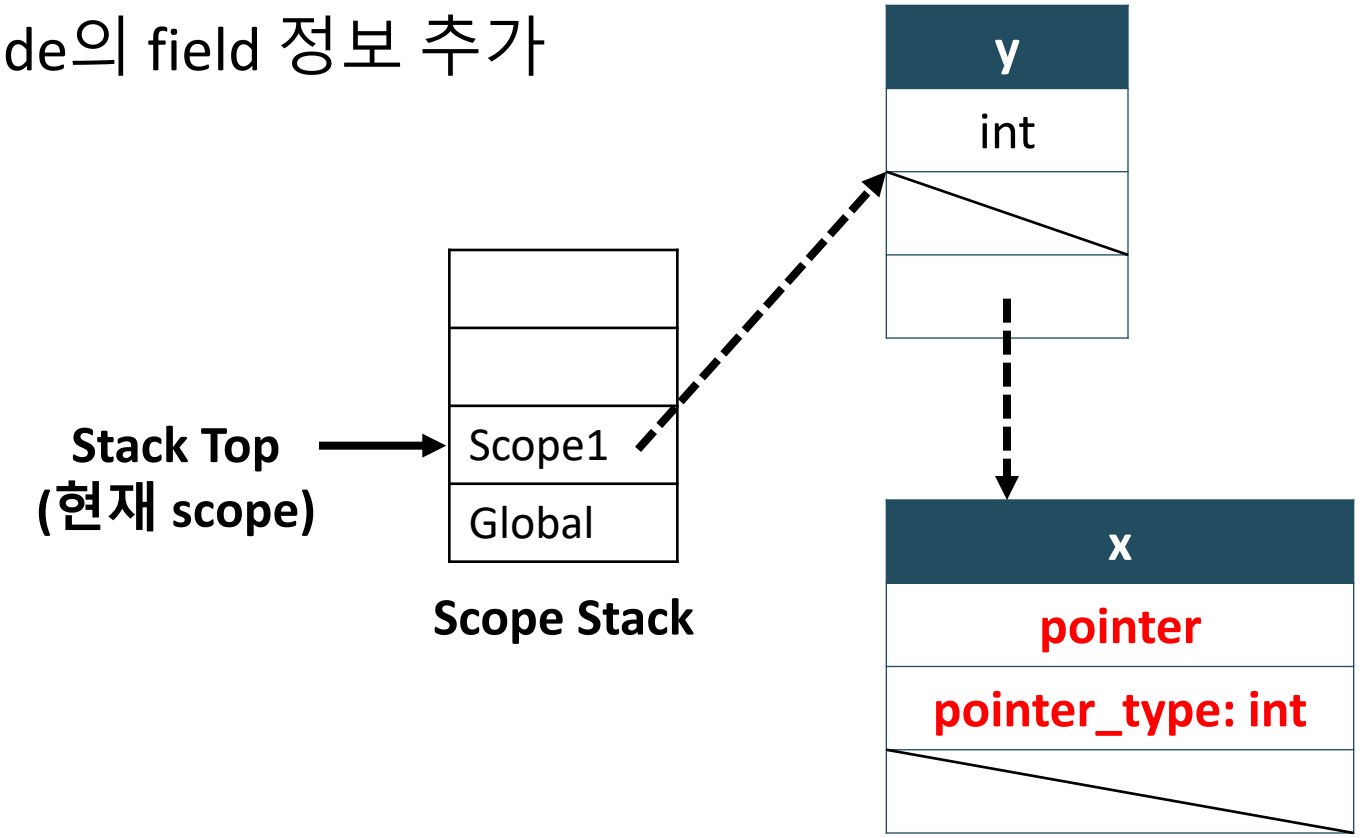
```
int x;
int y;
{
   char x;
   x = y;  /* char = int */
}
x = ..     /* int */
```



Scoped Symbol Table

- Pointer 처리
  - Pointer의 경우 pointer type에 대한 정보 추가 저장 필요
    - Symbol table(linked list) node의 field 정보 추가
    - 구현 방식은 자유!

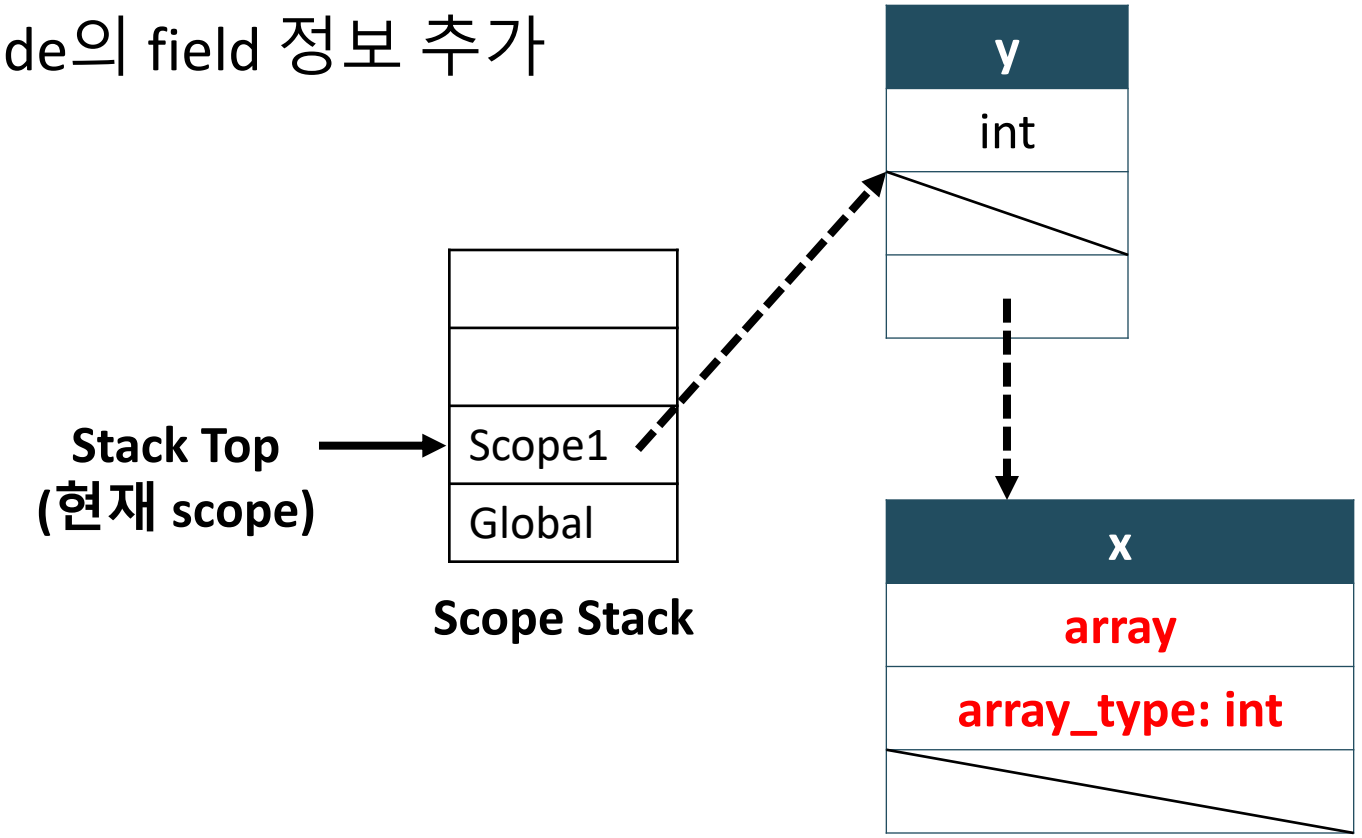
```
Int* x;
int y;
```



Scoped Symbol Table

- Array 처리
  - Array의 경우 역시 Array type에 대한 정보 추가 저장 필요
    - Symbol table(linked list) node의 field 정보 추가
    - 구현 방식은 자유!

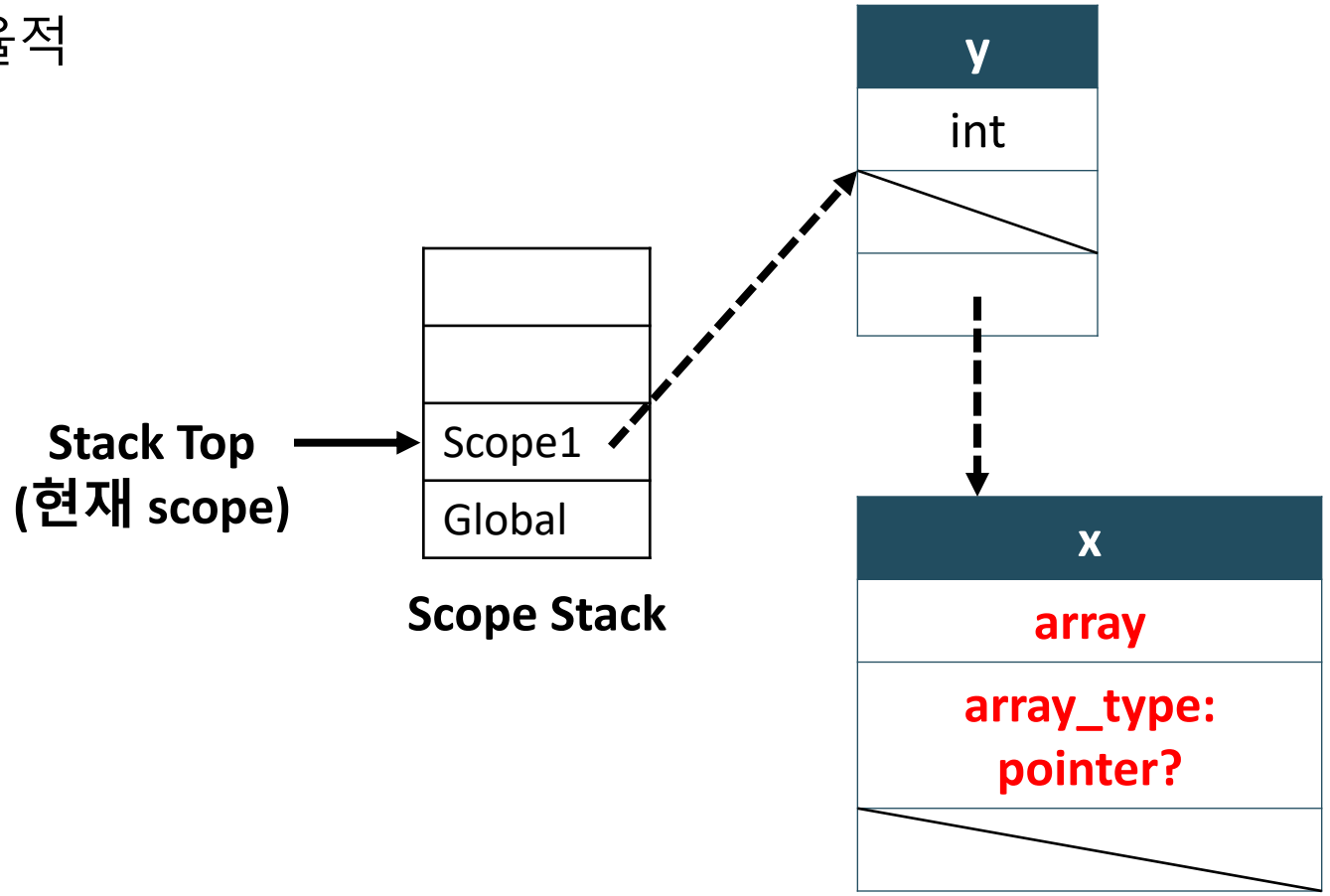
```
int x[10];
int y;
```



Scoped Symbol Table

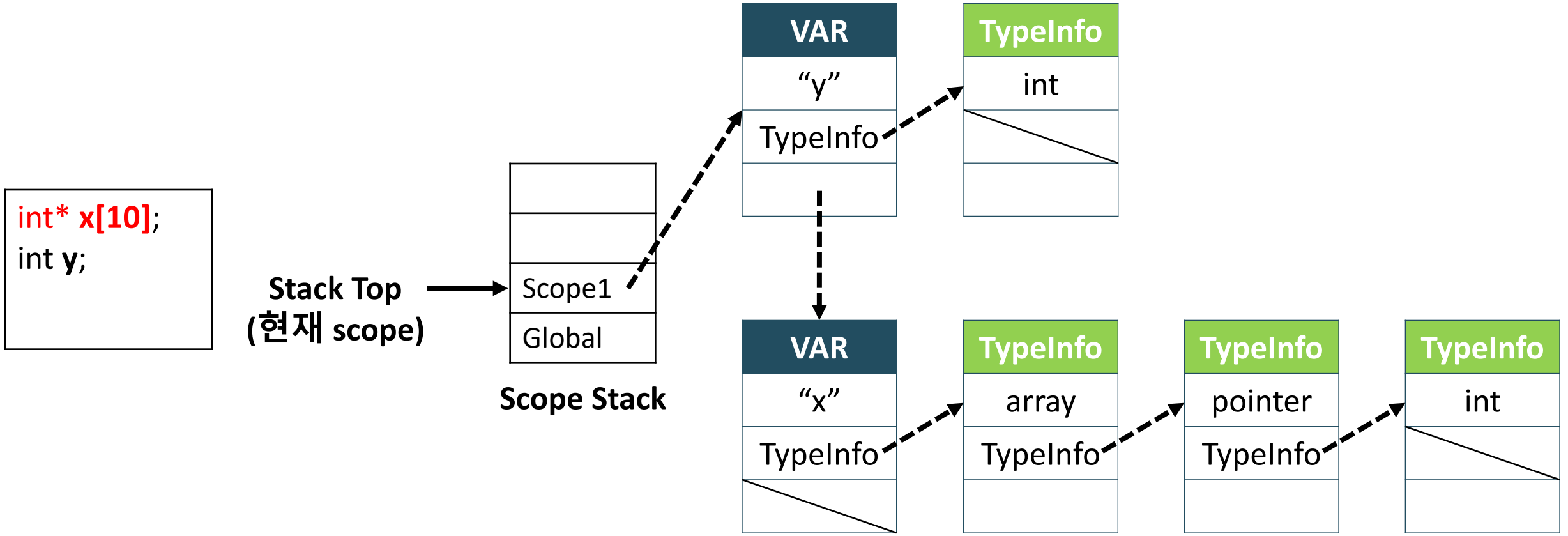
- Pointer에 대한 Array 처리??
  - 계속해서 node의 field 정보 추가 필요
    - Node 구성이 복잡해지고 비효율적

```
int* x[10];
int y;
```



Scoped Symbol Table

- Type 정보를 가진 별도의 구조체(TypeInfo)로 관리
  - **TypeInfo 의 linked list 형태로 연결**



Scoped Symbol Table

- Symbol 추가
  - Variable declaration production 에서 action 코드에 symbol 추가하는 코드 삽입
  - E.g. int a; int* x[10] 에 해당하는 production

Yacc file (.y)

```
def
  : type_specifier pointers ID '; ' { // symbol 추가하는 action code}
  | type_specifier pointers ID  ' [' INTEGER_CONST '] ' '; ' { // array symbol 추가}
  ;
```

  - ID
    - 토큰이므로 lex 에서 yylval을 통해 ID 의 값(string)을 전달
    - action 코드에서 **$3** 통해 ID 값을 얻고, 이를 symbol table 에 type 정보와 함께 추가
  - type_specifier
    - type 정보 (int, char, struct)
    - type_specifier production 의 action 코드에서 $$ 값에 type 정보 할당 필요

Scoped Symbol Table

- 새로운 Scope 추가
  - 새로운 Scope (block) 만나면 Scope stack 에 scope 추가
  - E.g. block { } 코드에 해당하는 production

Yacc file (.y)

```
compound_stmt
  : '{' {// scope 추가하는 action code} def_list stmt_list '}' { //scope 제거하는 action code}
  ;
```

    - 위 예제는 어디까지나 Hint 이며 실제 구현 시에는 function declaration 처리를 위해 위와 다르게 구현해야 할 수도 있음

Structure

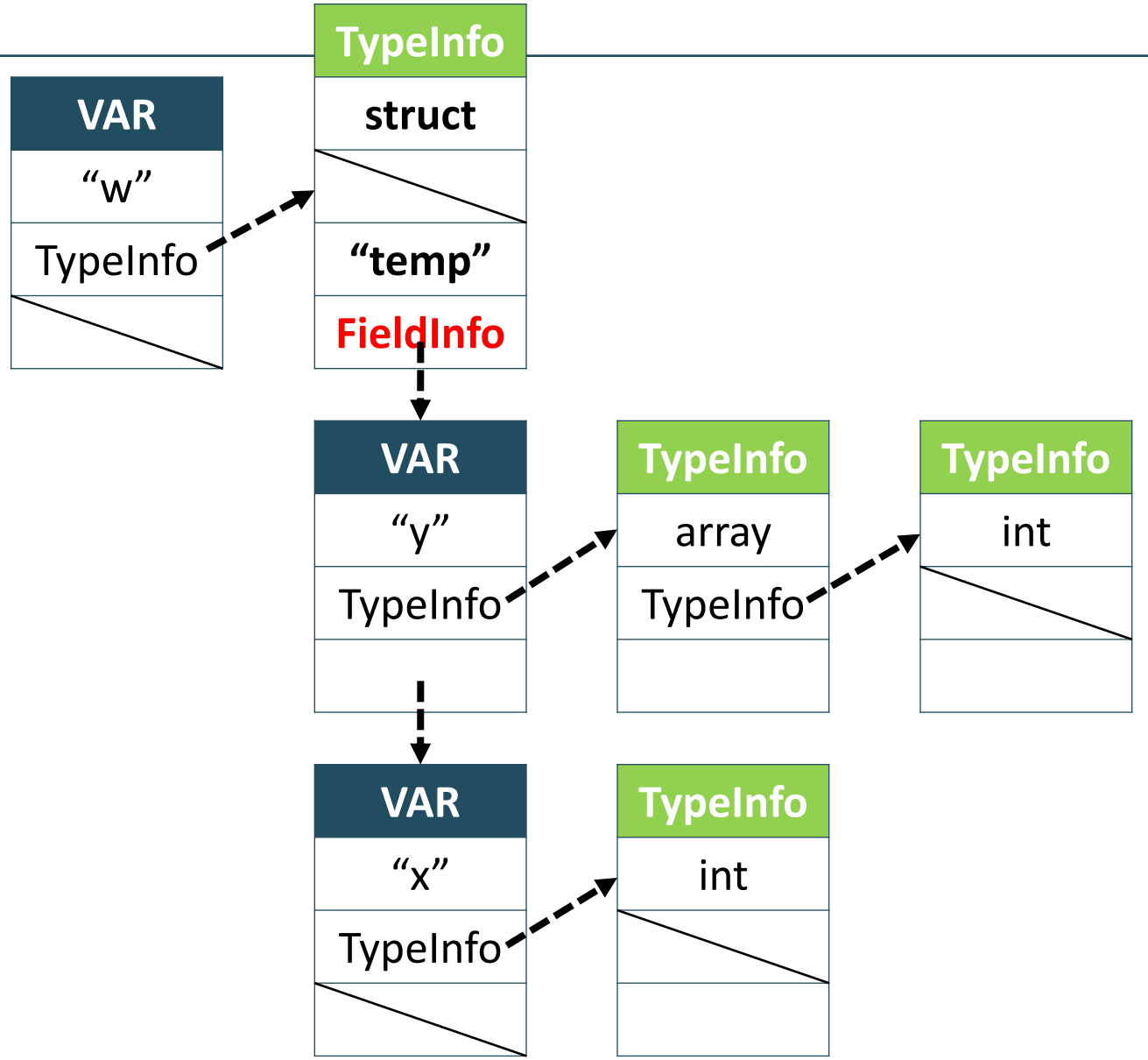
- Struct
  - 각 Struct 정의를 하나의 새로운 Type으로 처리

```
struct temp {
    int x;
    int y[20];
} w;
```

  - temp 라는 이름을 가진 struct type
  - w variable은 temp struct type 을 가짐
  - Struct 의 경우 field 정보(name, type) 같이 저장 필요

Structure

```
struct temp {
    int x;
    int y[20];
} w;
```

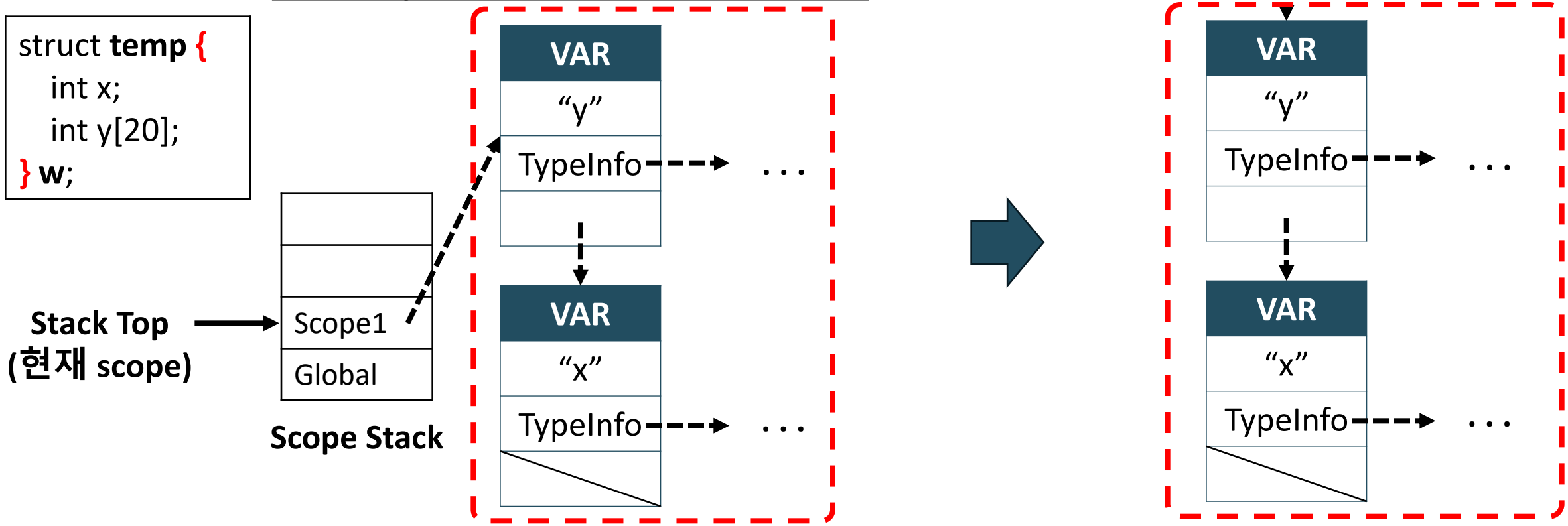


- Struct TypeInfo
  - TypeInfo 에 struct 관련 정보 추가
  - **Struct name**
  - **Struct field**
    - Variable declaration과 동일한 형태
    - 기존 variable 및 type 구조체 활용

Structure

- Struct Field 정보 취합
  - 기존 block statement 처리방식 활용
  - Struct body 도 하나의 block statement
    - 새로운 Scope 및 Symbol table 생성
    - **생성된 symbol table을 FieldInfo 에 삽입**

```
struct temp {
    int x;
    int y[20];
} w;
```

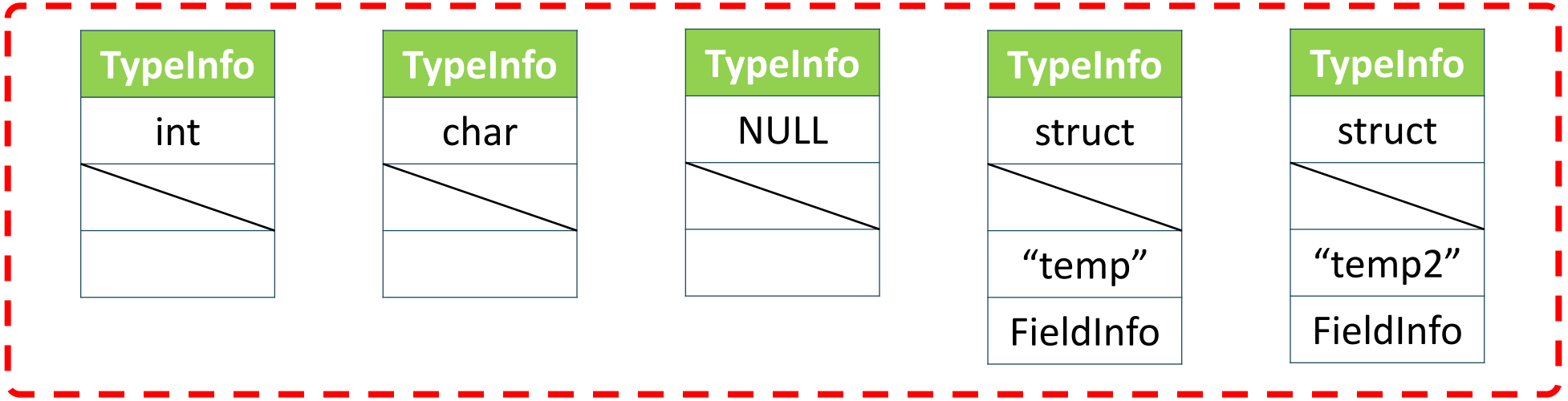


Type

- Project 에서 사용하는 Type 종류
  - int
  - char
  - NULL
  - 선언된 Struct
- Project 구현시 TypeInfo 구조체에서 구분하는 각 Type 종류
  - int
  - char
  - NULL
  - Pointer
  - Array
  - Struct

Type 관리

- Global Type List
  - 공유 가능한 TypeInfo 들의 Global Type List를 관리
    - Variable 마다 TypeInfo 를 생성하는 대신, Global TypeInfo 를 공유
    - **내부에서 다른 TypeInfo 정보를 참조하지 않는 Type들은 쉽게 공유 가능!**

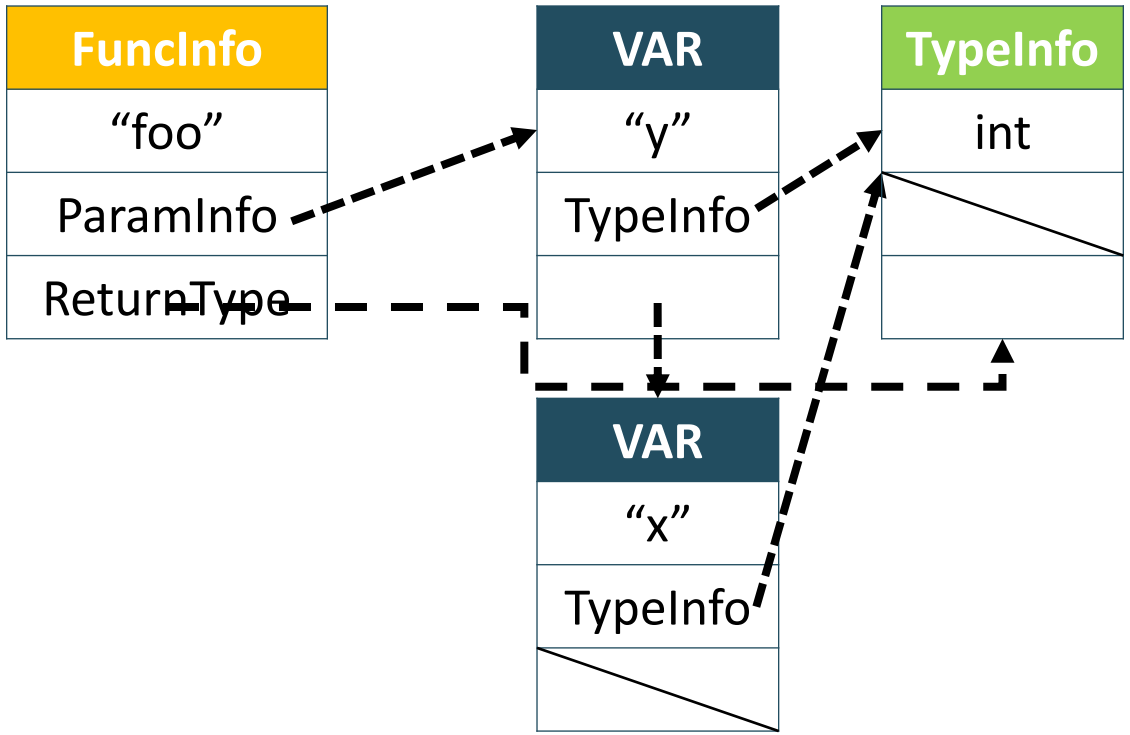


    - Struct 의 경우 선언 때마다 이 Global Type List 를 탐색하여 이미 동일한 이름의 Struct 있는지 체크, 없는 경우 Struct TypeInfo 생성하여 Global List에 추가

Function

- Function 구조체
  - Function 은 타입이 아니므로 variable 의 타입으로 사용될 수 없음
    - 별도의 Function 구조체로 관리
  - **Function Name, Parameter, Return type** 정보 저장 필요

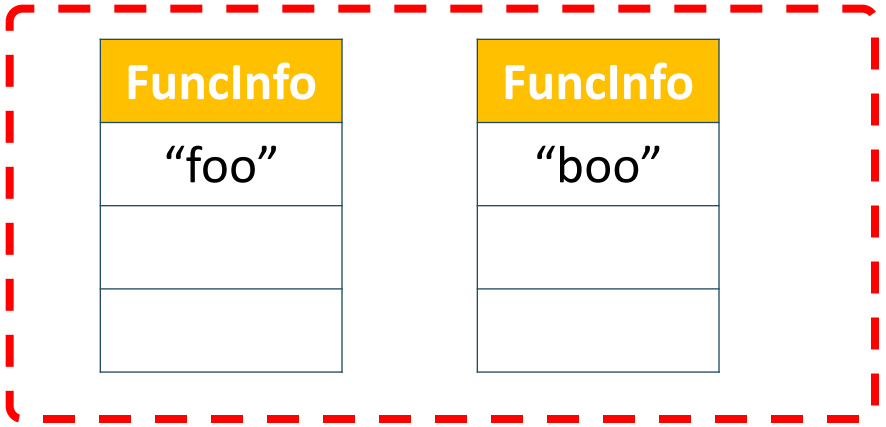
```
int foo (int x, int y)
{
   char z;
}
```



Function

- Global Function List
  - FuncInfo를 Global List로 관리
    - Function 정의마다 FuncInfo 생성하고 Global Function List 에 추가

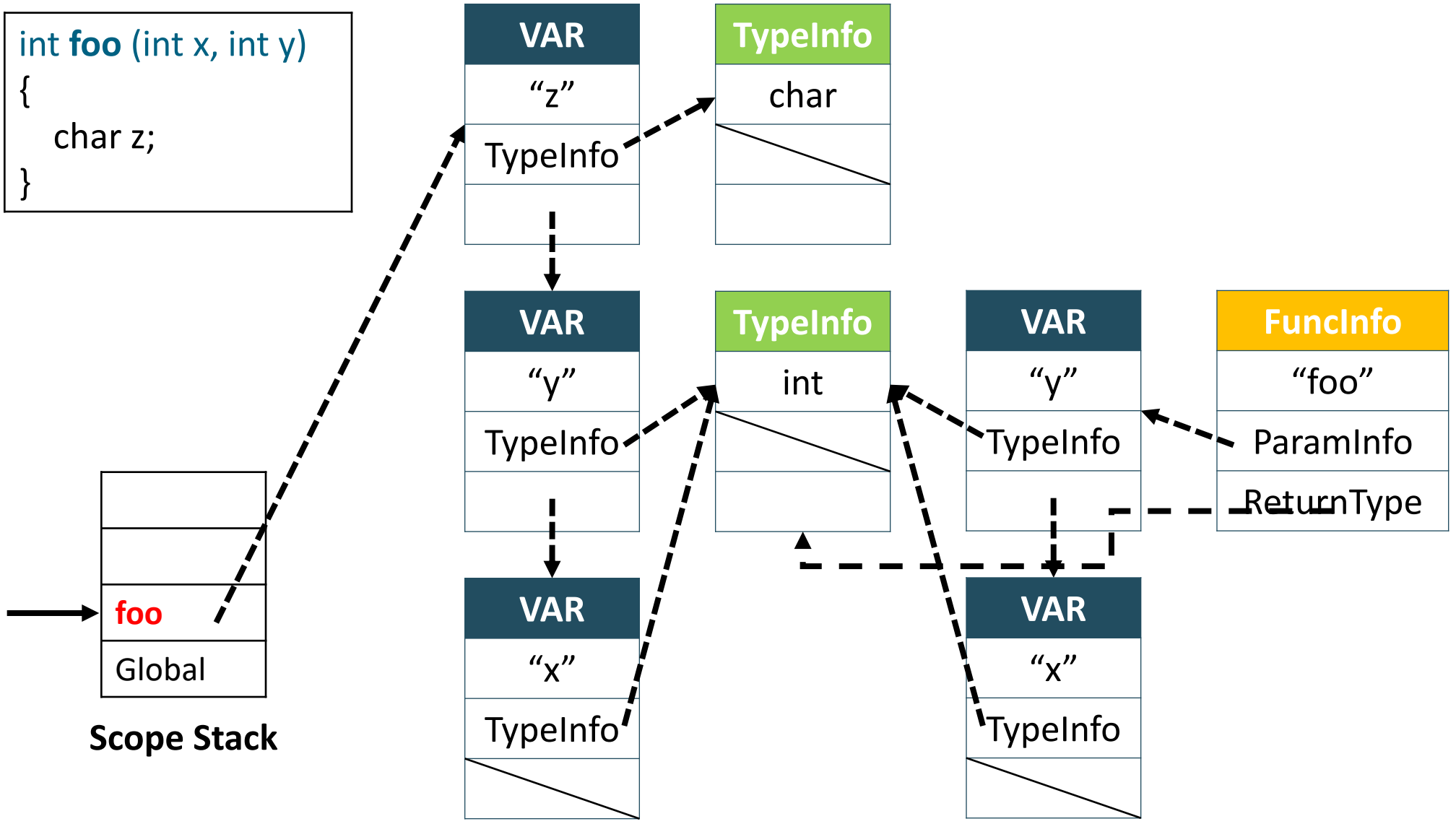
```
int foo (int x, int y) {
   return 0;
}

char boo () {
   return 'a';
}
```



- Function 정의 때마다 Global Function List 를 탐색하여 이미 동일한 이름의 Function 있는지 체크, 없는 경우 FuncInfo 생성하여 Global Function List에 추가
- **Parameter 의 경우 local variable 이기도 함!**

Function

```
int foo (int x, int y)
{
    char z;
}
```



Type Check 예제

- Binary operation : 사칙연산

Yacc file (.y)

```
binary
  :  binary '+' binary
  | binary '-' binary
  | binary '*' binary
  | binary '/' binary
  | unary
```

- binary symbol 의 value($i)는 unary로부터 전달
- 사칙연산 시 2개의 binary 의 type 이 같은지 확인이 필요

Type Check 예제

- Binary operation : 사칙연산

Yacc file (.y)

```
%{
#include “subc.h”       // subc.h 헤더파일에 필요한 구조체(e.g. TypeInfo) 정의
%}

%union {
   TypeInfo* typeInfo;
}

%token <typeInfo> unary binary  // unary, binary symbol value 를 TypeInfo* 타입으로 정의
```

- binary symbol 의 value 가 **TypeInfo** 에 대한 정보(pointer) 가져야 함
  - Action 코드에서 TypeInfo 토대로 2 개의 binary symbol 의 타입 비교
- **subc.h**
  - Global variable 선언 및 구현에 필요한 구조체 정의 포함

Lvalue

- Lvalue 고려
  - Left-value
  - 메모리 주소를 갖고 있어서 **값을 저장할 수 있는 value**
  - Assignment 의 LHS 에 올 수 있는 값

```
int a;
int b[10];
5 = a; /* error: lvalue is not assignable */
b = &a; /* error: lvalue is not assignable */
```

    - 상수 (integer const) 에는 값 저장이 불가능
    - Array name 은 const 값으로 저장이 불가능

  - **Parsing 에서 symbol 값에 lvalue 정보도 포함 필요!**

Lvalue

- Lvalue 고려한 ExtendedTypeInfo 구조체

Yacc file (.y)

```
%{
#include “subc.h”
%}

%union {
   ExtendTypeInfo* typeInfo;
}

%token <typeInfo> unary binary
```

ExtendedTypeInfo
TypeInfo*
lvalue (0/1)

- binary symbol 의 value 가 lvalue 정보를 포함한 **ExtendedTypeInfo** 포인터 저장

Semantic Check List

- Undeclared Variables & Functions
- Re-declaration
- Type Checking
- Structure & Structure pointer declaration
- Function Declaration

Undeclared Variables & Functions

- 선언되지 않은 변수 사용 및 정의되지 않은 함수 호출 에러
  - variable (undeclared)

    ```
    a = 0; /* error: use of undeclared identifier */
    ```

  - variable (out of scope)

    ```
    { int a; }
    a = 0; /* error: use of undeclared identifier */
    ```

  - function call (undeclared)

    ```
    foo(); /* error: use of undeclared identifier */
    ```

Re-declaration

- 동일한 이름의 Variable, Struct, Function 정의 에러
  - 구현 편의상 Variable, Struct, Function 등 서로 다른 종류끼리 이름 겹치는 경우는 고려하지 않음
    - E.g. int foo; int foo(int x, int y) {} 변수와 함수 이름 겹치는 경우는 고려하지 않음

```
{
   int a; /* OK */
   int a; /* error: redeclaration */
   char a; /* error: redeclaration */
}
{
   int a; /* OK */
   {
      int a; /* OK */
   }
}
```

Type Check (Assignment)

- Assignment Semantic Check
  - 다음과 같은 순서로 Semantic Check
  1. LHS가 lvalue인지 체크
     - Number 나 array name 의 경우 값 할당 불가능 (Const)
  2. RHS가 NULL이고 LHS가 포인터 타입인지 체크
  3. LHS와 RHS의 타입이 같은지 체크
     - **암묵적 형변환은 허용되지 않음** (e.g. char a = 1; 은 에러 발생)

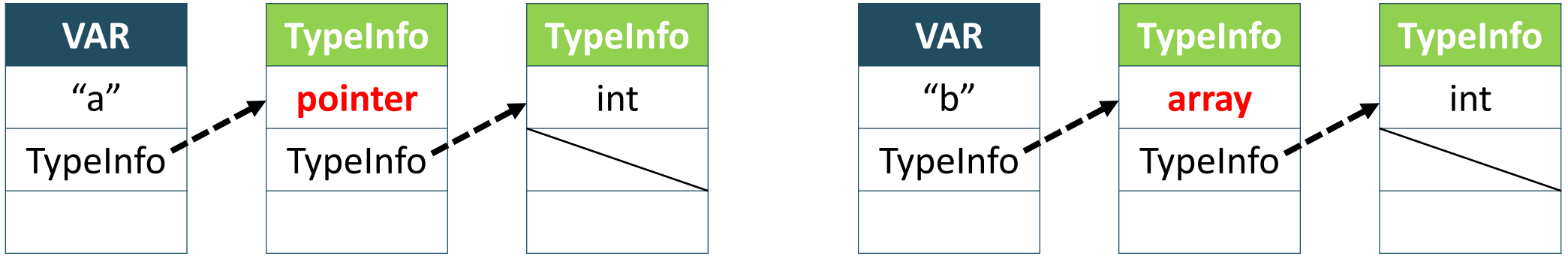
```
int a;
char b;
a = b; /* error: incompatible types for assignment operation*/
5 = a; /* error: lvalue is not assignable */
a = NULL; /* error: cannot assign 'NULL' to non-pointer type */
a = 5; /* OK */
```

Type Check (Assignment)

- Assignment Semantic Check

```
int *a;
int b[10];
a = b; /* error: incompatible types for assignment operation*/
b = a; /* error: lvalue is not assignable */
```

  - C 문법상 a = b 허용되나, 구현 편의상 a 와 b 의 TypeInfo 가 다르므로 다른 타입으로 처리!



Type Check (Assignment)

- Assignment Semantic Check

```
int *a[5];
int *b;
int c[10];
struct temp1 { int a; } *s1;
struct temp1 s2;
struct temp2 { int b; } *s3;

a = b;      /* error: lvalue is not assignable */
b = c;      /* error: incompatible types for assignment operation */
s1 = s3;  /* error: incompatible types for assignment operation */
s1 = s2;  /* error: incompatible types for assignment operation */
s1 = &s2; /* OK */
```

  - Struct 의 경우 구조가 동일해도 다른 이름으로 선언되어 있는 경우 다른 타입으로 처리

Type Check (Binary 연산)

- +, -, *, /, %
  - Integer type 에만 허용, 그 외 타입은 에러
- &&, || (logical)
  - Integer type 에만 허용, 그 외 타입은 에러

```
char c1;
char c2;
char c3;
c1 = ‘a’;
c2 = ‘b’;

c3 = c1 + c2 /* error: invalid operands to binary expression*/
c1 && c2; /* error: invalid operands to binary expression*/
```

Type Check (Binary 연산)

- >=, >, <=, < (Relop)
  - int, char 에만 허용, 그 외 타입은 에러
  - **연산 결과는 int type 으로 처리**

```
int result;
int a;
int b;
char c;
result = (a > 5) || ( a <= b ); /* OK */
result = (a < c) /* error: types are not comparable in binary expression */
```

Type Check (Binary 연산)

- ==, != (Equop)
  - int, char, pointer type 에만 허용, 그 외 타입은 에러
  - **연산 결과는 int type 으로 처리**

```
int result;
int *a;
int *b;
char *c;
result = ( a == b ); /* OK */
result = ( a == c ); /* error: types are not comparable in binary expression */
```

Type Check (Unary 연산)

- -, !
  - Integer type 에만 허용, 그 외 타입은 에러

```
int a;
char b;
a = 10;
b = 'a';
a = -a; /* OK */
b = -b; /* error: invalid argument type to unary expression*/
b = !b; /* error: invalid argument type to unary expression*/
```

Type Check (Unary 연산)

- ++, --
  - int, char 에만 허용, 그 외 타입은 에러

```
int a;
char b;
int* c;
char d[10];
struct temp { int a;} e;
a++;
--a;
b++;
c++; /* error: invalid argument type to unary expression*/
--d; /* error: invalid argument type to unary expression*/
++e; /* error: invalid argument type to unary expression*/
```

Type Check (Pointer)

- Pointer Operator : *, &
  - * 오른쪽 (RHS) 에는 pointer type 만 가능
  - & 오른쪽 (RHS) 에는 rvalue 만 가능: 구현 편의상 non-lvalue (number, array, string) 불가능
- NULL
  - 0 은 NULL 로 사용될 수 없음
  - NULL 은 pointer type 에만 할당 가능

```
int *a;
int b;
int c[10];
a = 0; /* error: incompatible types for assignment operation */
a = NULL; /* OK */
a = &b; /* OK */
a = *b; /* error: indirection requires pointer operand */
&b = a; /* error: lvalue is not assignable */
b = &c; /* error: cannot take the address of an rvalue */
b = 0; /* OK */
b = *a; /* OK */
```

Type Check (Struct)

- Struct operator : ., ->
  - . 왼쪽 (LHS) 에는 struct type 만 가능
  - -> 왼쪽 (LHS) 에는 struct pointer type 만 가능
  - ., -> 오른쪽의 identifier 는 structure 의 field name 만 가능

```
struct str1 {int i; char c;};
struct str1 st1;
struct str1 *pst1;
int main() {
    int i;
    i = st1.i;
    i = st1.i2; /* error: no such member in struct */
    i = st1->i; /* error: member reference base type is not a struct pointer */
    i = pst1->i;
    i = pst1.i; /* error: member reference base type is not a struct */
}
```

Type Check (Array)

- Array operator: [ ]
  - a[i]
  - a 는 array type 만 가능, i 는 int type 만 가능

```
int a[5];
int b;
char c;

b = a[1];
a[1] = b;
a[1] = b[1]; /* error: subscripted value is not an array */
a[b];
a[c]; /* error: array subscript is not an integer */
```

Structure & Structure pointer declaration

- Structure
  - Struct type 은 struct instance (사용) 전에 정의되어야 함
  - 구현 편의상 Struct 는 항상 global declaration 으로 처리
    - Scope 은 struct 에 적용 안됨

- Structure Pointer
  - Struct pointer 선언 시 global type list 탐색하여 해당 struct 탐색
  - Global type list 에 없는 경우 오류

Structure & Structure pointer declaration

- Example

```
struct a {
    struct b x; /* error: incomplete type */
    struct b* p; /* error: incomplete type */
    struct b { } y; /* OK */
};

struct b { /* error: redeclaration */
};

int func() {
    struct b { } x; /* error: redeclaration */
}
```

Function Declaration

- Function Check
  - Return type 체크
  - Function call 에서 argument type 과 function 정의의 parameter type 비교
  - Function call 이후 return 값 타입 체크

```
int func1(int a, char b) { return 0; }
int func2(int a, char b) { return 'c'; } /* error: incompatible return types */
int func1() { return 0; } /* error: redeclaration */

int main() {
    int a;
    int b;
    char c;
    b = func1(a, b); /* error: incompatible arguments in function call */
    b = func1(a, c);
    c = func1(a, c); /* error: incompatible types for assignment operation */
    b = a(); /* error: not a function */
}
```

Grammar

- Project에 적용되는 Grammar
  - subc.y (skeleton code) 및 grammar.txt 에 전체 grammar 제공됨
    - 그대로 사용 혹은 필요에 따라 수정 가능!
  - Syntax Error는 고려하지 않음 (채점용 테스트 코드에는 syntax error가 없음)
  - 제공된 Grammar 특징 (기존 C 문법에 위배되지만, 구현의 편의를 위해 단순화)
    - Variable 를 declaration과 동시에 초기화 할 수 없음
      - int a = 0; /* syntax error */
    - Anonymous structure declaration은 지원되지 않음
      - struct { int x; int y; } w; /* syntax error */
    - 모든 Variable declaration은 scope (block) 의 첫 부분에 존재

```
int a;
a = 5;
int b;     /* syntax error */
{ int a; } /* OK */
```

Grammar

- 고려하지 않아도 되는 사항
  - Function
    - 자기 자신을 call 하는 함수 (self-recursive)
    - Return statement 가 없는 함수 (e.g. void func() {})
    - Function Overloading (동일한 이름의 함수가 2개 이상 정의될 수 없음)
  - Struct
    - 자기 자신을 멤버로 갖는 struct
  - Variable, Struct, Function 등 서로 다른 종류끼리 이름 겹치는 경우는 고려하지 않음
  - 문자열 assign (e.g. char* a = “hello”;)

Memory 오류 체크 – 추가 점수 항목

- AddressSanitizer (Asan)
  - C,C++ 언어에서 메모리 오류 탐지하기 위한 런타임 도구
  - 메모리 오류 탐지
    - Heap Buffer Overflow
    - Stack Buffer Overflow
    - Use-After-Free (해제된 메모리 접근)
    - **메모리 누수(leak)**
  - 컴파일 시 gcc 옵션 추가 (-fsanitize=address -static-libasan -g)
    - $ gcc **-fsanitize=address -static-libasan -g** -o myprog main.c
  - 필수 구현 사항은 아니지만, Asan으로 메모리 오류 없는 경우 가산점

Memory 오류 체크 – 추가 점수 항목

- AddressSanitizer (Asan)

C Source Code

```
int main (int argc, char** argv)
{
   . . .
   int* x = malloc(sizeof(int));   // memory leak
   return 0;
}
```

수행 결과 에러 메시지 (**memory leak**) 출력

```
==4399==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 4 byte(s) in 1 object(s) allocated from:
   #0 0x5620be1c30d7 in malloc (/temp/testprog+0x980d7)
   #1 0x5620be208c7a in main /temp/main.c:8
   #2 0x7f720f362d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58

SUMMARY: AddressSanitizer: 4 byte(s) leaked in 1 allocation(s).
```

Output

- 에러 메시지에 File name 과 line number 포함하여 출력
  - <filename>:<line_num>:(SPACE)error:(SPACE)<description>
  - “error” 전, 후에 빈칸 하나씩 삽입
  - EX) test.c:5 error: redeclaration
  - subc.l 에 정의된 **get_lineno()** 활용

  - 에러 메시지
    - 스켈레톤 subc.y 내에 제공된 메시지 출력 함수 활용
    - “error_” 로 시작하는 함수들
    - 미완성인 error_preamble() 함수에서 line number 및 file name 출력하도록 구현

Output

- 여러 semantic error 가 존재하는 경우
  - 각각의 error 를 순차적으로 모두 출력
  - Error 가 발생한 statement parsing 에서는 **에러 메시지 출력 후 symbol value 에 NULL을 할당**
  - 이전 reduce 에서 받은 symbol 값이 NULL인 경우 action 코드 수행없이 진행

```
int a;
char a; /* error */
a = 1;   /* OK */
a = 'c'; /* error */
```

Yacc file (.y)

```
ext_def
  : type_specifier pointers ID ';'   {
                                        if (check_is_declared($3)) {
                                           error_message();
                                           $$ = NULL;
                                        } else {
                                           $$ = addVariable($1, $2, $3);
                                        }
                                     }
```

Output

- 하나의 소스 코드 라인에서 여러 에러가 존재하는 경우
  - Parsing 할 때 먼저 찾는 에러 1개만 출력

```
int main() {
   int a;
   int b[10];
   b = a[1];     /* error: lvalue is not assignable */
                 /* error: subscripted value is not an array */
   return 1;
}
```

    - 위 예제 경우
      - expr -> unary ‘=‘ expr 통한 REDUCE 보다
      - unary -> unary ‘[‘ expr ‘]’ 가 먼저 REDUCE 되므로
      - subscripted value is not an array 에러만 출력

  - 그 외 하나의 REDUCE 에서 2개 이상 에러 존재하는 경우는 고려하지 않음

Tips

- 구현 편의를 위해 global variable 활용 필요
  - subc.h 에 정의하면 충돌 발생 -> 별도의 c file 내부에 global variable 선언
  - E.g. Scope stack 의 top 포인터, 현재 parsing 중인 함수의 return type, …
- 여러 action code 에서 자주 사용되는 코드는 C 함수로 만들고 호출하면 코드 유지 보수 ↑
  - 함수들은 subc.h 에 선언
  - 함수들을 별도 c 파일에 구현
  - 해당 c 파일 내부에서 global variable 사용

Submission

- 제출기한
  - **6/15 (일요일) 23:59**
  - 늦은 제출(delay) 없음

- 제출방법
  - lms 의 레포트 Project3 통해 제출

- 제출파일
  - ‘src’ directory 안의 작성한 파일들(**Makefile 포함**)을 zip 으로 압축하여 제출
  - Asan 적용할 시 Makefile 안에 Asan 빌드 옵션 추가하여 제출
  - Zip 파일 이름: project3_학번.zip
    - Ex) project3_202012345.zip