Project #3. Semantic Analysis

Symbol Table & Type Checker

2023 Compiler

Prof. Eul Gyu Lm

Project Goal

- **C-Minus Semantic Analyzer Implementation**
  - C-Minus parser with Lex and Yacc (in project 2) should be used.
    - **Start from the C-Minus Scanner & Parser (Uploaded in LMS)**
  - Find All Semantic Errors using **symbol table** & **type checker**
    - Semantic analyzer reads an input source string and generates AST (by tokenizing, parsing, …) as in the previous project.
    - After that, the semantic analyzer traverses the AST to find and print semantic errors and its line number
  - *symtab.c, analyze.c, … -> **cminus_semantic** (executable)*

Project Goal: Semantic Error Detection

- **Un/Redefined Variables and Functions**
  - Scope rules are same as C language
  - Function overloading is not allowed
- **Void type Variable**
  - Void is not allow for Variable’s type
- **Array Indexing Check**
  - Only *int* value can be used as an index
  - Index to Not Array
- **Operation’s LHS & RHS Type Check**
  - Operations such as *int*[] + *int*[], *int*[] + *int* and *void* + *void* are not allowed
    - *int* + *int* : *int*, *int* < *int* : *int*
  - *Assignment is not necessary for Operations*

- **Assignment Type**
  - LHS & RHS should have same type
- ***if/while* condition**
  - Only *int* value can be used for condition
- ***Function Call’s argument***
  - *Compare with Function Parameter*
- ***Return type***
  - *Compare with Function Type*
- **Built-in Functions**

Built-in Functions

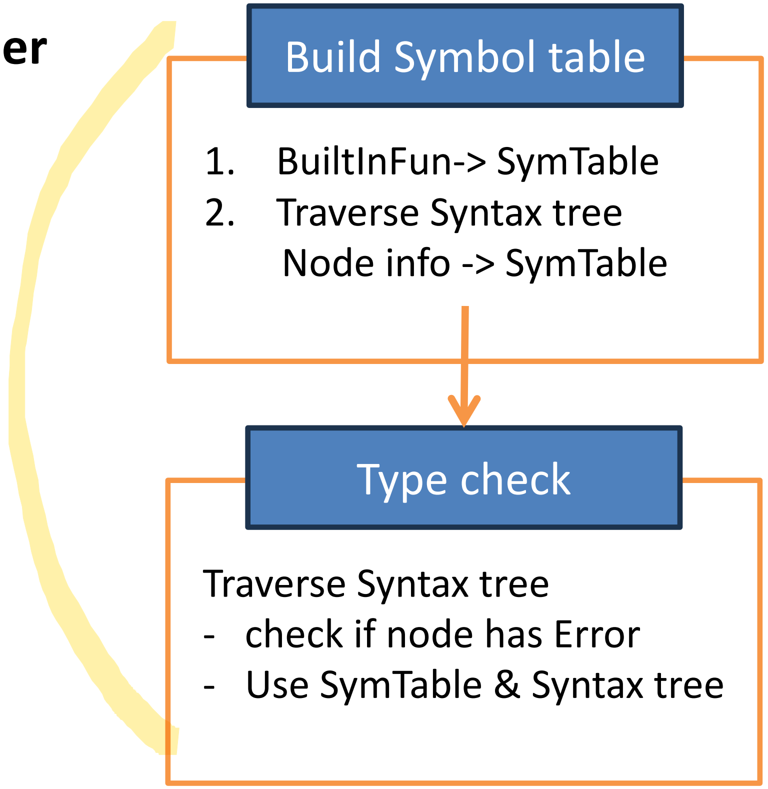
- ***Two Built-in Functions***
  - ***int input(void)***
    - Returns a value of the given integer value from the user.
  - ***void output(int value)***
    - Prints a value of the given argument.
- These two global functions are defined by default.

- The codes for Built-in Functions are provided.
  - See **declareBuiltInFunction()** & **buildSymtab()** in **anaylze.c**
    - While building symbol table, insert the symbols for built-in Functions as Global scope & lineno 0.

How to implement?

- **Implement symbol table and type checker**
- **Traverse syntax tree created by parser**

- **Files to check**
  - ***symtab.h, symtab.c***
  - ***analyze.h, analyze.c***
  - *globals.h : check the data structures*
  - *main.c : check the flow*
  - *etc*



Build Symbol table

1. BuiltInFun-> SymTable
2. Traverse Syntax tree
   Node info -> SymTable

Type check

Traverse Syntax tree
- check if node has Error
- Use SymTable & Syntax tree

Symbol Table in C-Minus

Example C-Minus Code

```
 1:   /* A program to perform Euclid's
 2:     Algorithm to computer gcd */
 3:
 4:   int gcd (int u, int v)
 5:   {
 6:       if (v == 0) return u;
 7:       else return gcd(v,u-u/v*v);
 8:      /* u-u/v*v == u mod v */
 9:   }
10:
11:   void main(void)
12:   {
13:       int x; int y;
14:       x = input(); y = input();
15:       output(gcd(x,y));
16:   }
```

Symbol Table

Name	Type	Location	Scope	Line Numbers
output	Void	0	global	0 15
Input	Integer	1	global	0 14 14
gcd	Integer	2	global	4 7 15
main	Void	3	global	11
u	Integer	0	gcd	4 6 7 7
v	Integer	1	gcd	4 6 7 7 7
x	Integer	0	main	13 14 15
y	Integer	1	main	13 14 15

- **Name**
  - The name of the symbol
  - Used in symbol identifications
- **Location**
  - Counter for memory locations of the variable
  - Never overlapped in a scope
- **Line Numbers**
  - Line numbers that the variable is defined and used

- **Scope**
  - The scope where the symbol is defined
- **Type**
  - The type of the symbol

Symbol Table in C-Minus

```
 1:  /* A program to perform Euclid's
 2:    Algorithm to computer gcd */
 3:
 4:  int gcd (int u, int v)
 5:  {
 6:      if (v == 0) return u;
 7:      else return gcd(v,u-u/v*v);
 8:     /* u-u/v*v == u mod v */
 9:  }
10:  int gcd (int x) { return x; }
11:
12:  void main(void)
13:  {
14:      int x; int y;
15:      x = input(); y = input();
16:      output(gcd(x,y));
17:      z = input();
18:  }
```

gcd?

z?

Symbol Table

Name	Type	Location	Scope	Line Numbers
output	Void	0	global	0 15
Input	Integer	1	global	0 14 14
gcd	Integer	2	global	4 7 15
main	Void	3	global	11
u	Integer	0	gcd	4 6 7 7
v	Integer	1	gcd	4 6 7 7 7
x	Integer	0	main	13 14 15
y	Integer	1	main	13 14 15

- Line 10: The symbol defined as function is the same as already defined in symbol table.
  → Semantic Error: redefined function ‘gcd’ at line 10
- Line 17: The symbol used in main() are not defined in symbol table yet (both main and global scopes).
  → Semantic Error: undefined variable ‘z’ at line 17

Type Checker

```
/* A program to perform Euclid's
  Algorithm to computer gcd */

int gcd (int u, int v)
{
    if (v == 0) return u;
    else return gcd(v,u-u/v*v);
   /* u-u/v*v == u mod v */
}

void main(void)
{
    int x; int y;
    x = input(); y = input();
    output(gcd(x,y));
}
```

```
Op: -
  Variable: name = u
  Op: *
    Op: /
      Variable: name = u
      Variable: name = v
    Variable: name = v
```

Syntax Tree

왼쪽 child와 오른쪽 child의 type이 같은지 check

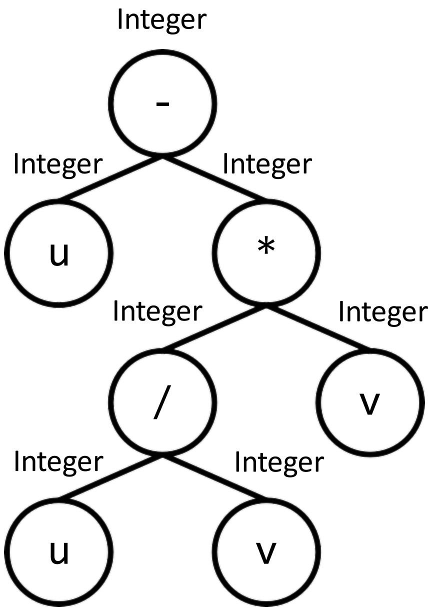
Type Checker

typeCheck()

case Binary Operator:

1) Check if LHS is an Integer
2) Check if RHS is an Integer
3) Then its result type is an Integer

Correct!



Symbol Table

Name	Type			
u	Integer			
v	Integer			
…	…			

Type Checker

```
 1:  /* A program to perform Euclid's
 2:    Algorithm to computer gcd */
 3:
 4:  int gcd (int u, int v)
 5:  {
 6:      if (v == 0) return u;
 7:      else return gcd(v,u-u/v*output());
 8:     /* u-u/v*v == u mod v */
 9:  }
10:
11:  void main(void)
12:  {
13:      int x; int y;
14:      x = input(); y = input();
15:      output(gcd(x,y));
16:  }
```

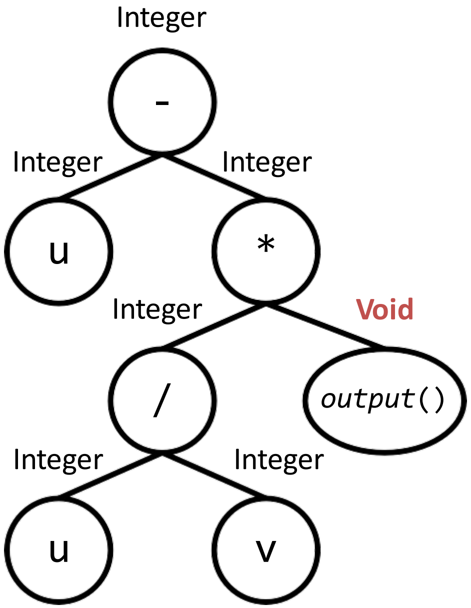
```
Op: –
  Variable: name = u
  Op: *
    Op: /
      Variable: name = u
      Variable: name = v
    Call: function name = output
```

Syntax Tree

Type Checker
typeCheck()

case Binary Operator:
1) Check if LHS is an Integer
2) Check if RHS is an Integer
3) Then its result type is an Integer

Incorrect!

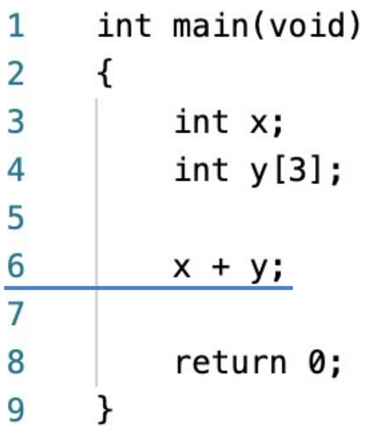


Symbol Table

Name	Type			
u	Integer			
v	Integer			
output	Void			
…	…			

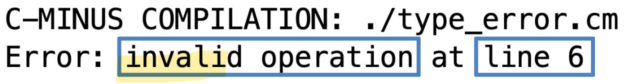
- Line 7: Type checker finds an error
  → Semantic Error: type error at line 7

Output Examples



```
1   int main(void)
2   {
3       int x;
4       int y[3];
5
6       x + y;
7
8       return 0;
9   }
```

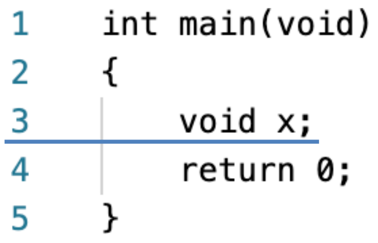
Interger + IntegerArray



```
C-MINUS COMPILATION: ./type_error.cm
Error: invalid operation at line 6
```

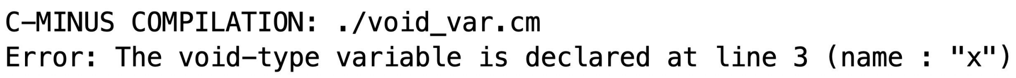
Error Type

Line Number



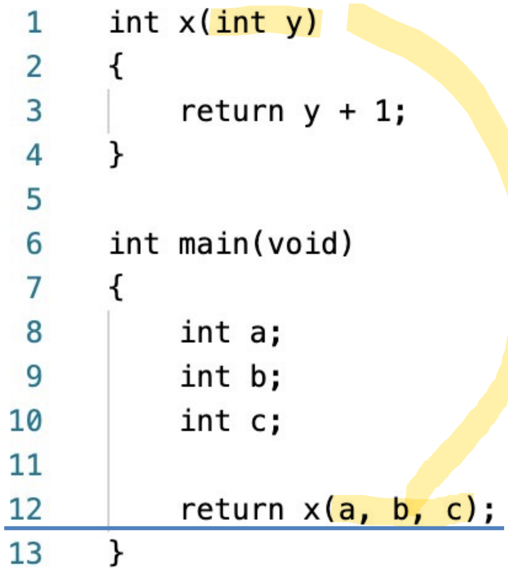
```
1   int main(void)
2   {
3       void x;
4       return 0;
5   }
```

No void type for variable



```
C-MINUS COMPILATION: ./void_var.cm
Error: The void-type variable is declared at line 3 (name : "x")
```

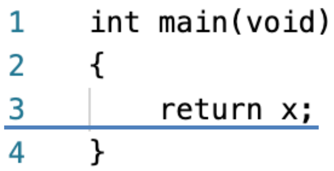
Output Examples



```
1     int x(int y)
2     {
3         return y + 1;
4     }
5
6     int main(void)
7     {
8         int a;
9         int b;
10        int c;
11
12        return x(a, b, c);
13    }
```

parameters vs.
arguments

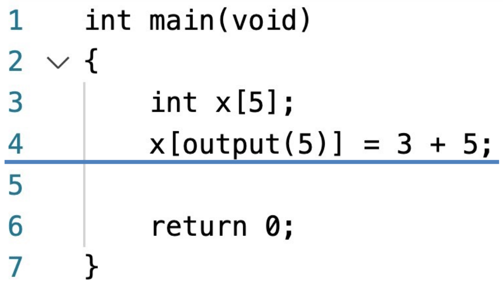
```
C-MINUS COMPILATION: ./invalid_func.cm
Error: Invalid function call at line 12 (name : "x")
```



```
1     int main(void)
2     {
3         return x;
4     }
```

```
C-MINUS COMPILATION: ./undeclared_var.cm
Error: undeclared variable "x" is used at line 3
Error: Invalid return at line 3
```

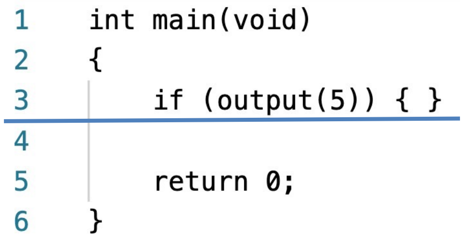
Output Examples



```
1  int main(void)
2  {
3      int x[5];
4      x[output(5)] = 3 + 5;
5
6      return 0;
7  }
```

Built-in function output’s return type is void

```
C-MINUS COMPILATION: ./invalid_index.cm
Error: Invalid array indexing at line 4 (name : "x").
indices should be integer
```

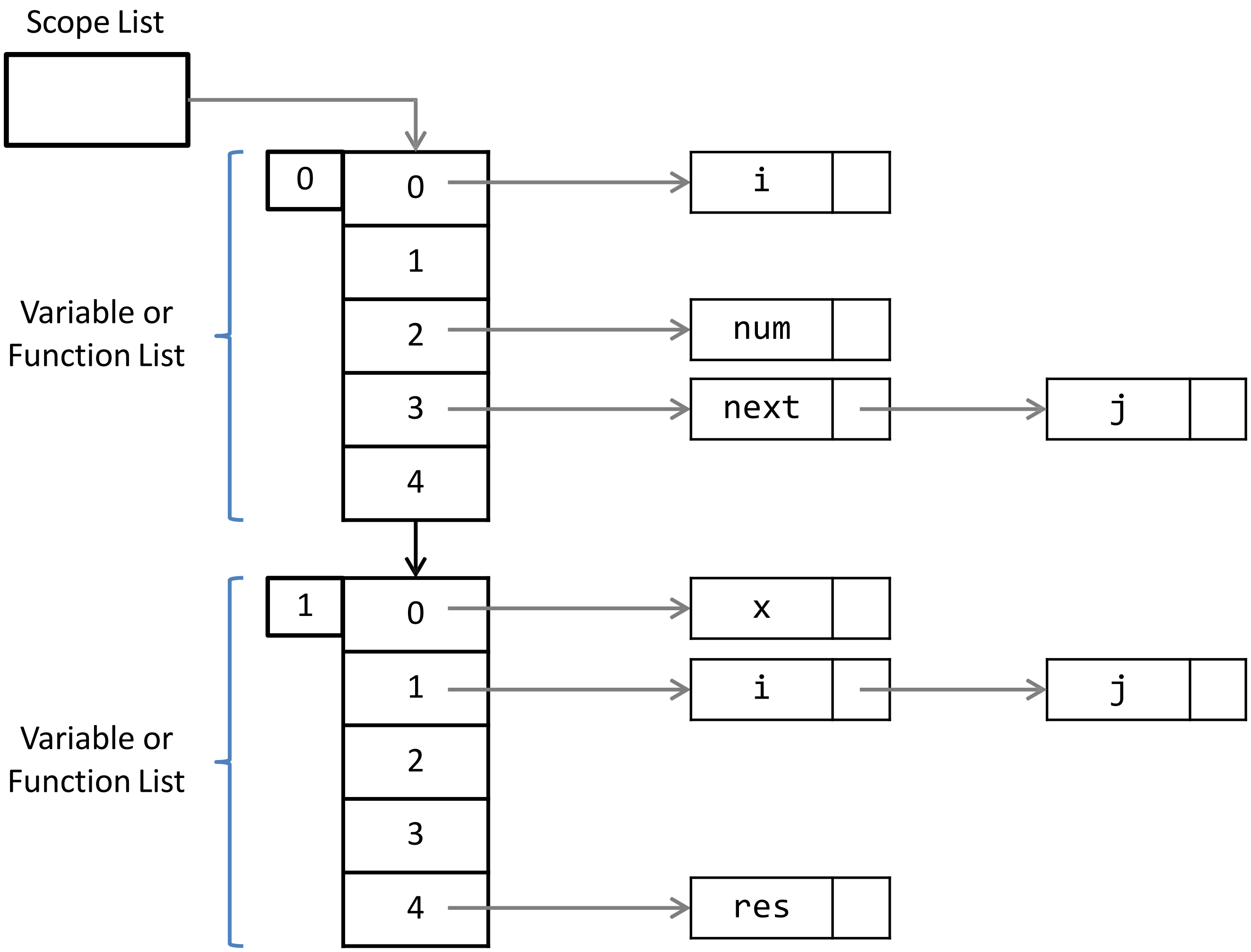


```
1  int main(void)
2  {
3      if (output(5)) { }
4
5      return 0;
6  }
```

Built-in function output’s return type is void

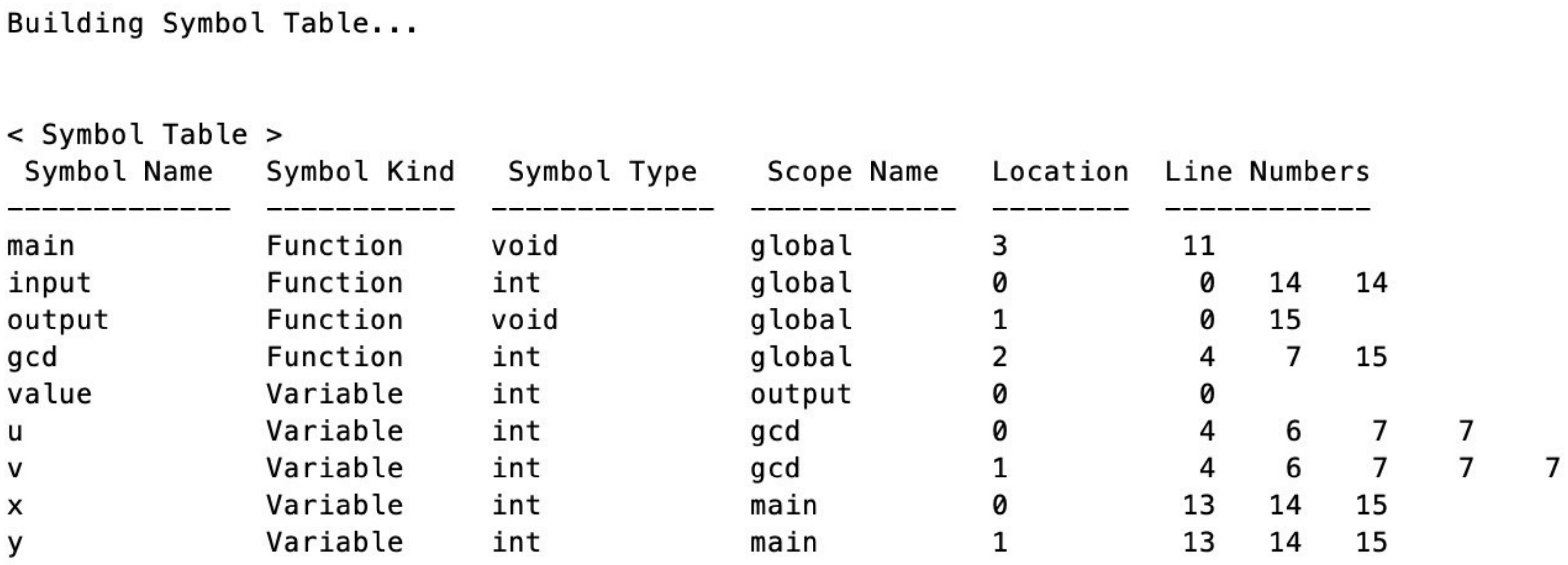
```
C-MINUS COMPILATION: ./invalid_condition.cm
Error: invalid condition at line 5
```

Hint: Symbol Table Implementation (Case 1)



Hint: Symbol Table Implementation (Case 2)

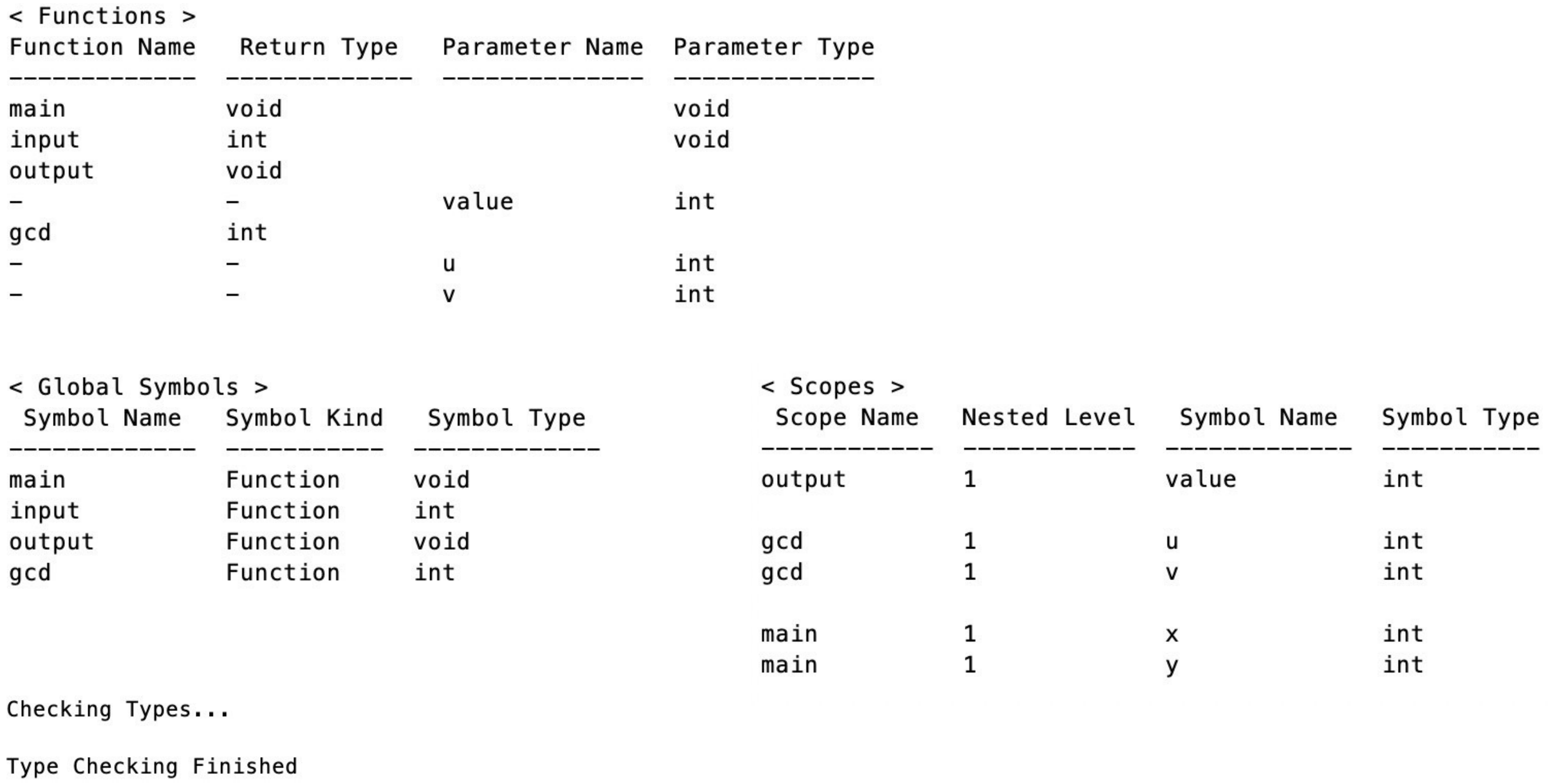
- Build with ***TraceAnalyze = TRUE*** in *main.c*



```
Building Symbol Table...


< Symbol Table >
 Symbol Name   Symbol Kind   Symbol Type    Scope Name   Location  Line Numbers
-------------  -----------  -------------  ------------  --------  ------------
main           Function     void           global        3          11
input          Function     int            global        0           0   14   14
output         Function     void           global        1           0   15
gcd            Function     int            global        2           4    7   15
value          Variable     int            output        0           0
u              Variable     int            gcd           0           4    6    7    7
v              Variable     int            gcd           1           4    6    7    7    7
x              Variable     int            main          0          13   14   15
y              Variable     int            main          1          13   14   15
```

Hint: Symbol Table Implementation (Case 2)

- Build with ***TraceAnalyze = TRUE*** in *main.c*



```
< Functions >
Function Name   Return Type   Parameter Name  Parameter Type
-------------  -------------  --------------  --------------
main           void                           void
input          int                            void
output         void
-              -              value           int
gcd            int
-              -              u               int
-              -              v               int


< Global Symbols >                                   < Scopes >
 Symbol Name   Symbol Kind   Symbol Type              Scope Name   Nested Level   Symbol Name   Symbol Type
-------------  -----------  -------------            ------------  ------------  -------------  -----------
main           Function     void                     output        1             value          int
input          Function     int
output         Function     void                     gcd           1             u              int
gcd            Function     int                      gcd           1             v              int

                                                     main          1             x              int
                                                     main          1             y              int

Checking Types...

Type Checking Finished
```

Type Checker

- **Type checking for functions and variables**
  - Check the number and types of arguments for function call.
  - Check return type.
  - The type *void* is only available for functions.

  - Check if the types of two operands can be matched when assigning.
  - Check if the condition for *if* or *while* can be evaluated to *int*.

  - Check other things by referring to C-Minus syntax.
  - *Note)* Types in C-Minus → void, int, int[]

Hint: Build with Makefile

```
# Makefile for C-Minus
#
# ./lex/tiny.l        --> ./cminus.l (from Project 1)
# ./yacc/tiny.y       --> ./cminus.y
# ./yacc/globals.h    --> ./globals.h

CC = gcc

CFLAGS = -W -Wall

OBJS = main.o util.o lex.yy.o y.tab.o

.PHONY: all clean
all: cminus_parser

clean:          rm -vf cminus_parser *.o lex.yy.c y.tab.c y.tab.h y.output
                rm -vrf temporary_for_grading

cminus_parser: $(OBJS)
                $(CC) $(CFLAGS) $(OBJS) -o $@ -lfl

main.o: main.c globals.h util.h scan.h parse.h y.tab.h
                $(CC) $(CFLAGS) -c main.c

util.o: util.c util.h globals.h y.tab.h
                $(CC) $(CFLAGS) -c util.c

scan.o: scan.c scan.h util.h globals.h y.tab.h
                $(CC) $(CFLAGS) -c scan.c

lex.yy.o: lex.yy.c scan.h util.h globals.h y.tab.h
                $(CC) $(CFLAGS) -c lex.yy.c

lex.yy.c: cminus.l
                flex cminus.l

y.tab.h:  y.tab.c

y.tab.o: y.tab.c parse.h

                $(CC) $(CFLAGS) -c y.tab.c

y.tab.c: cminus.y
                yacc -d -v cminus.y
```

Use *–ll* instead of *–lfl*
for MacOS

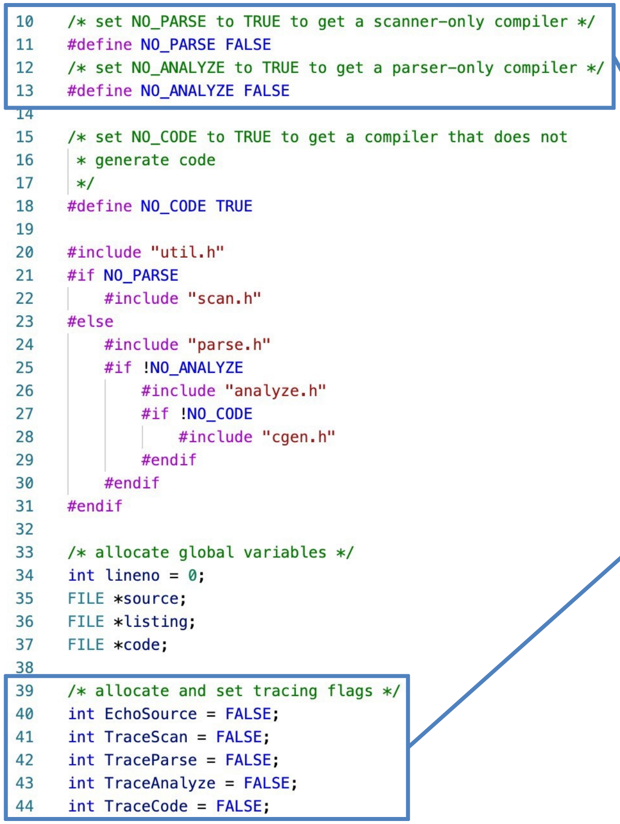
You can also use the Shellscript testcase_result.sh
- build with makefile
- make output files for all of the testcase in './my_result' directory

---------commandline-----------------

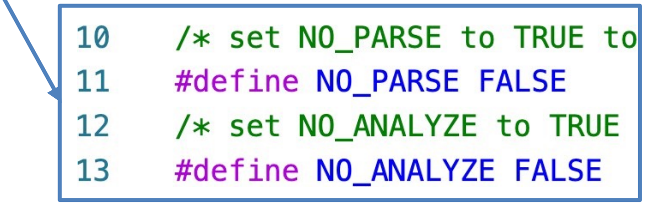
```
chmod +x testcase_result.sh
./testcase_result.sh
```

Main.c

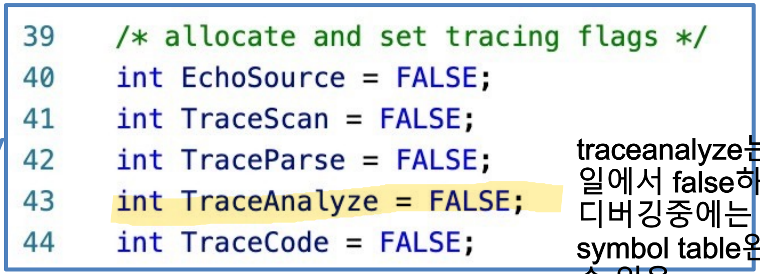
- ***main.c***
  - Modify code to print only semantic errors
  - *NO_ANALYZE*, *NO_CODE*, *TraceParse*, and *TraceAnalyze*



```c
/* set NO_PARSE to TRUE to get a scanner-only compiler */
#define NO_PARSE FALSE
/* set NO_ANALYZE to TRUE to get a parser-only compiler */
#define NO_ANALYZE FALSE

/* set NO_CODE to TRUE to get a compiler that does not
 * generate code
 */
#define NO_CODE TRUE

#include "util.h"
#if NO_PARSE
    #include "scan.h"
#else
    #include "parse.h"
    #if !NO_ANALYZE
        #include "analyze.h"
        #if !NO_CODE
            #include "cgen.h"
        #endif
    #endif
#endif

/* allocate global variables */
int lineno = 0;
FILE *source;
FILE *listing;
FILE *code;

/* allocate and set tracing flags */
int EchoSource = FALSE;
int TraceScan = FALSE;
int TraceParse = FALSE;
int TraceAnalyze = FALSE;
int TraceCode = FALSE;
```



```c
/* set NO_PARSE to TRUE to
#define NO_PARSE FALSE
/* set NO_ANALYZE to TRUE
#define NO_ANALYZE FALSE
```



```c
/* allocate and set tracing flags */
int EchoSource = FALSE;
int TraceScan = FALSE;
int TraceParse = FALSE;
int TraceAnalyze = FALSE;
int TraceCode = FALSE;
```

traceanalyze는 최종파일에서 false하고 근데 디버깅중에는 true하면 symbol table완성시 볼 수 있음

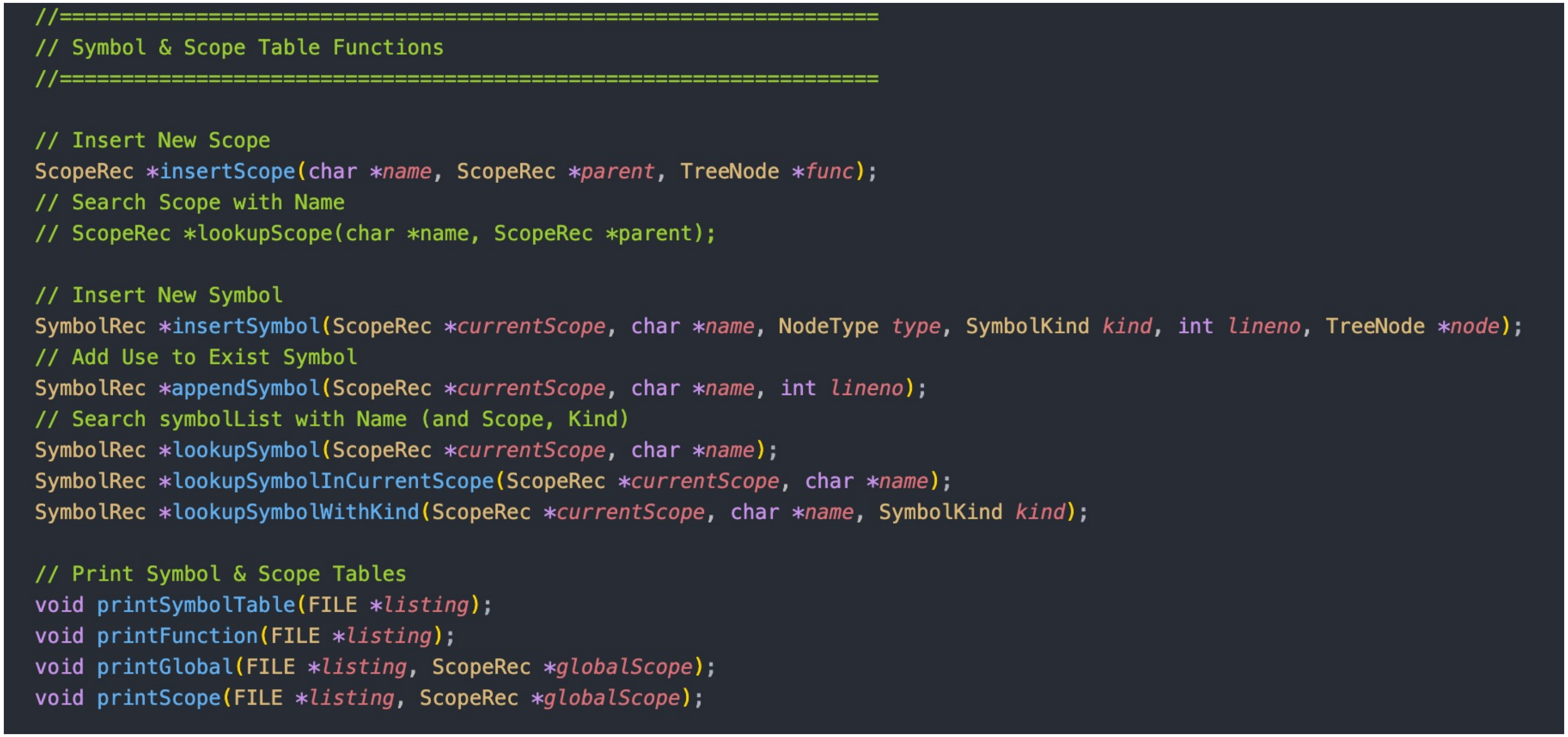
* *TraceAnalyze* helps to debug semantic analyzer

Where to See?

- ***symtab.h* & *symtab.c***
  - Symbol table implementations
  - Sample Codes are provided
  - Scope and type information is required in C-Minus
    - Or you can define multiple table structures to describe whole C-Minus semantics as in case 2.
    - Scope has a hierarchical structure. New scopes are added within compound statements (child of upper scope) and function declarations (child of global scope).

Hint: Where to See?

- ***symtab.h* & *symtab.c***
  - Samples codes are provided.



```
//==================================================================
// Symbol & Scope Table Functions
//==================================================================

// Insert New Scope
ScopeRec *insertScope(char *name, ScopeRec *parent, TreeNode *func);
// Search Scope with Name
// ScopeRec *lookupScope(char *name, ScopeRec *parent);

// Insert New Symbol
SymbolRec *insertSymbol(ScopeRec *currentScope, char *name, NodeType type, SymbolKind kind, int lineno, TreeNode *node);
// Add Use to Exist Symbol
SymbolRec *appendSymbol(ScopeRec *currentScope, char *name, int lineno);
// Search symbolList with Name (and Scope, Kind)
SymbolRec *lookupSymbol(ScopeRec *currentScope, char *name);
SymbolRec *lookupSymbolInCurrentScope(ScopeRec *currentScope, char *name);
SymbolRec *lookupSymbolWithKind(ScopeRec *currentScope, char *name, SymbolKind kind);

// Print Symbol & Scope Tables
void printSymbolTable(FILE *listing);
void printFunction(FILE *listing);
void printGlobal(FILE *listing, ScopeRec *globalScope);
void printScope(FILE *listing, ScopeRec *globalScope);
```

Where to See?

- ***analyze.c***
  - Modify symbol table generation
    - *buildSymtab*(), ***insertNode*()**: actual symbol table generation implementation
  - Modify type checker
    - *typeCheck*(), ***checkNode*()**: actual type checker implementation
  - Insert built-in function (Sample Codes are provided)
    - ***input*()**, ***output*()**
  - Implement error messages in semantic errors (Sample Codes are provided)

2 ways to implementation

1. Make your Own : Building symbol tables is just an intermediate process for semantic analysis, so you can implement them however you want.

두 가지 방법이 있다
기존에 있는거 사용하거나 자기가 만들어서 하거나

2. Use the sample codes

1. Analyze **Flow** : Start from the main.c
2. Analyze the **Data Structures & Function** of Sample Codes

   - symtab.c ,symtab.h, analyze.c , analyze.h, etc (global.h, util.c, util.h, scan.c ,scan.h)
3. **Fill the Code** of given area in following functions **(analyze.c)**
   1. static void insertNode(TreeNode *t)
      1. The Section of insert Symbols with Traverse SyntaxTree (Period of building SymbolTable)
      2. Use the Data structures & Functions in **symtab.c & symtab.h**
      3. Take care of **Void-Type variables & Redefinition errors & undeclared variable Error**
   2. static void checkNode(TreeNode *t)
      1. Type check Period
      2. Use the syntax tree & Symbol tree

Output Formats

- **Error type** with its **line number , Output messages should be same as follwing formats**
  - "Error: Undeclared function \"%s\" is called at line %d\n”
  - "Error: Undeclared variable \"%s\" is used at line %d\n”
  - "Error: Symbol \"%s\” is redefined at line %d\n"
  - "Error: Invalid array indexing at line %d (name : \"%s\"). Indices should be integer\n"
  - "Error: Invalid array indexing at line %d (name : \"%s\"). Indexing can only be allowed for int[] variables\n”
  - "Error: Invalid function call at line %d (name : \"%s\")\n"
  - "Error: The void-type variable is declared at line %d (name : \"%s\")\n"
  - "Error: Invalid operation at line %d\n”
  - "Error: Invalid assignment at line %d\n"
  - "Error: Invalid condition at line %d\n”
  - "Error: Invalid return at line %d\n”

- **How to grades the project?**
  - Compare output files (Answer files vs. output files from your code)
  - The output format is important when grading.
  - Don’t worry, defined functions for outputs are given in **analyze.c**

Output Formats

- Recommend to use the following **defined functions** in **analyze.c**

```
// Error Handlers
static void RedefinitionError(char *name, int lineno, SymbolList symbol);
static SymbolRec *UndeclaredFunctionError(ScopeRec *currentScope, TreeNode *node);
static SymbolRec *UndeclaredVariableError(ScopeRec *currentScope, TreeNode *node);
static void VoidTypeVariableError(char *name, int lineno);
static void ArrayIndexingError(char *name, int lineno);
static void ArrayIndexingError2(char *name, int lineno);
static void InvalidFunctionCallError(char *name, int lineno);
static void InvalidReturnError(int lineno);
static void InvalidAssignmentError(int lineno);
static void InvalidOperationError(int lineno);
static void InvalidConditionError(int lineno) {

        fprintf(listing, "Error: invalid condition at line %d\n", lineno);
        Error = TRUE;
}

static void TypeError(int lineno, char *message);
```

Evaluation

- **Evaluation Items**
  - **Compilation** (Success / Fail): **20%**
    - Please describe in the report how TA can build your project.
  - **Correctness** check for several testcases: **70%**
    - Note: Make sure there are no segmentation fault or infinite loop on any inputs.
  - **Report** : **10%**

Report

Guideline (≤ 5 pages but no limits)

1. Compilation environment and method
2. Brief explanations about how to implement and how it operates
3. Examples and corresponding result screenshots

Format

PDF format

Submission

Deadline: 12/20 **(Wed.) 23:59:59**

Submission

Place all the **source codes** in the **StudentID/3_Semantic** directory

Place **report** in the **StudentID** directory

Zip the **StudentID** directory

Upload the zip file to the **LMS system**

*Renaming **StudentID** directory names based on your StudentID

- No name, Only **StudentID**
- **it's related with the grading script**

Questions

E-mail: **ted6345@hanyang.ac.kr**

Please provide all questions related with projects to TAs.

Q&A