Project #3: Semantic Analysis

2024 Fall

Hunjun Lee

Hanyang University

Project Goal

- **C-Minus Semantic Analyzer Implementation**
  - You should **find semantic errors** using a symbol table and type checker

- **What you should do:**
  - Start from the Lex and Yacc in Project 1 & 2
  - Modify the code if necessary (you may need to modify the AST structure for semantic analysis)
  - Traverse over the AST to generate a symbol table
  - Traverse over the AST again and use the symbol table to perform type checking

TODO #1: Scope Analysis

- **Un/Redefined Variables and Functions**
  - Scope rules are the same as C language (but unlike C, C-minus does not allow separate declaration)

C-Minus …

```
int foo(int x, int y) {
    …
}

int boo(int z) {
    int i, j;
    foo(i, j);
}
```

Declare the functions before usage

Python …

```
int boo(int z) {
    int i, j;
    foo(i, j);
}

int foo(int x, int y) {
    …
}
```

Can use functions before declaration

C …

```
int foo(int x, int y);
int boo(int z) {
    int i, j;
    foo(i, j);
}
int foo(int x, int y) {
    …
}
```

Separate body declaration

TODO #2: Built-in Functions

- **You have two built in functions which are defined by default**
- **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
- **Assume that these functions are declared at line 0**

TODO #3: Type Checking - 1

- **You cannot declare a void-type variable**

- **Only integer variables are compatible with arithmetic and logical operations**
  - int + int : int
  - int < int : int
  - Not allowed: int[] + int[], int[] + int, void + void, …

- **Assignment type (there is no type conversion)**
  - You are allowed to assign int [] to int [] (size does not matter)

TODO #3: Type Checking - 2

- **if / while: you are only allowed to use int value for condition**
- **function arguments / parameters:**
  - The number and types of the arguments and parameters should match
- **return type**
- **Array indexing check**
  - Only int value can be used as an index

Error Output Formats

- **Please refer to the attached file for output format specifications (error_messages.c)**
  - "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"

Basic Output Examples - 1

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

Invalid expression

```
C-MINUS COMPILATION: test_1.cm
Error: invalid operation at line 6
```

Error Type Error Type

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

```
C-MINUS COMPILATION: test_1.cm
Error: The void-type variable is declared at line 3 (name : "x")
```

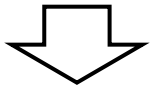
Error Type Error Type / Var Name

Basic Output Examples - 2

```
 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 }
```

```
C-MINUS COMPILATION: test_1.cm
Error: Invalid function call at line 12 (name : "x")
```

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



```
C-MINUS COMPILATION: test_1.cm
Error: undeclared variable "x" is used at line 3
Error: Invalid return at line 3
```

Basic Output Examples - 3

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

```
C-MINUS COMPILATION: test_1.cm
Error: Invalid array indexing at line 4 (name : "x"). indicies should be integer
```

Handling Undeclared Variables / Functions - 1

- **Assume that the variable is implicitly declared after there is an undeclared variable usage**

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

Undeclared variable @ line 3

Use implicitly declared variable @ line 3

```
// Symbol table
symbol kind   type          line no
-------------------------------------
x      var    undetermined  3 4
```

```
C-MINUS COMPILATION: test_1.cm
Error: undeclared variable "x" is used at line 3
```

Handling Undeclared Variables / Functions - 2

- **The undetermined type results in type checking error**
  - Condition, Add, Compare, …

```
1 int main (void)
2 {
3     int y;
4     x;
5     x + y;
6 }
```

Undeclared variable @ line 3

Use implicitly declared variable @ line 3

```
// Symbol table
symbol kind   type          line no
-------------------------------
x      var    undetermined 3 4
```

```
C-MINUS COMPILATION: test_1.cm
Error: undeclared variable "x" is used at line 4
Error: invalid operation at line 5
```

Handling Undeclared Variables / Functions - 3

- **The undetermined type results in type checking error**
  - Condition, Add, Compare, …

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

Undeclared variable @ line 4

Use implicitly declared variable @ line 4

```
// Symbol table
symbol kind   type           line no
-------------------------------------
y      var    int            3 4 5
x      var    undetermined   4 5
```

```
C-MINUS COMPILATION: test_1.cm
Error: undeclared variable "x" is used at line 4
Error: invalid operation at line 4
Error: invalid operation at line 5
```

Handling Undeclared Variables / Functions - 4

- **Assume that the function is implicitly declared after there is an undeclared function call**
  - Return type: undetermined & parameter type: undetermined

```
1 int main (void)
2 {
3     x(1, 2);
4
5     if(x()){}
6
7     return 0;
8 }
```

Undeclared function @ line 3

Use implicitly declared func @ line 3

```
C-MINUS COMPILATION: test_1.cm
Error: undeclared function "x" is called at line 3
Error: Invalid function call at line 3 (name : "x")
Error: Invalid function call at line 5 (name : "x")
Error: invalid condition at line 5
```

```
// Symbol table
symbol kind   type          line no
-------------------------------
x      func   undetermined 3 5
```

```
// Function Detail
Name   Return Type   Param Name    Param Type
-----------------------------------------
x      undetermined                undetermined
```

Modify the Line Number - 1

- **Modify the line number according to the following error lines ☹**
  - Function / Variable Declaration: follow the **identifier line number**

```
 1 int main (void)
 2 {
 3     int
 4         y;
 5     int y;
 6     int
 7         y
 8         ;
 9
10     return 0;
11 }
```

```
C-MINUS COMPILATION: test_1.cm
Error: Symbol "y" is redefined at line 5 (already defined at line 4)
Error: Symbol "y" is redefined at line 7 (already defined at line 4 5)
```

Modify the Line Number - 2

- **Modify the line number according to the following error lines ☹**
  - Expressions (including assignments): the line number should be set according to the **starting line**

```
 1 int main (void)
 2 {
 3     int x;
 4     int y[5];
 5
 6     x +
 7         y
 8         +
 9         5;
10
11     return 0;
12 }
```

```
C-MINUS COMPILATION: test_1.cm
Error: invalid operation at line 6
```

Modify the Line Number - 3

- **Modify the line number according to the following error lines ☹**
  - Expressions (including assignments): the line number should be set according to the **starting line**

```
 1 int main (void)
 2 {
 3     int x;
 4     int y[5];
 5
 6     y
 7         + y
 8         + y;
 9
10     return 0;
11 }
```

```
C-MINUS COMPILATION: test_1.cm
Error: invalid operation at line 6
Error: invalid operation at line 6
```

Modify the Line Number - 4

- **Modify the line number according to the following error lines ☹**
  - Statement: the line number should be set according to the **ending line**

```
 1 int main (void)
 2 {
 3     int x;
 4     int y[5];
 5
 6     if(x +
 7         y)
 8     {}
 9
10     return 0;
11 }
```

```
C-MINUS COMPILATION: test_1.cm
Error: invalid operation at line 6
Error: invalid condition at line 8
```

Handling Invalid Operations

- **Invalid binary operation returns an undetermined type**
  - Ex) After adding x + y, it returns undetermined type ➔ Therefore, assigning x + y to z results in an assignment error

```
 1 int main (void)
 2 {
 3     int x;
 4     int y[5];
 5     int z;
 6
 7     z = x + y;
 8
 9     return 0;
10 }
```

```
C-MINUS COMPILATION: test_1.cm
Error: invalid operation at line 7
Error: invalid assignment at line 7
```

Symbol Table in Tiny

```
// Example Code (for Tiny)
 1: { Sample program
 2: in TINY language -
 3: computes factorial
 4: }
 5: read x; { input an integer }
 6: if 0 < x then { don't compute if x <= 0 }
 7: fact := 1;
 8: repeat
 9: fact := fact * x;
10: x := x - 1
11: until x = 0;
12: write fact { output factorial of x }
13: end
```

Symbol Table

Variable Name	Location	Line Numbers
x	0	5 6 9 10 10 11
fact	1	7 9 9 12

- **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

Symbol Table in C-Minus

```
// Example Code (for Tiny)
 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

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

Symbol Table in C-Minus

```
// Example Code (for Tiny)
 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: }
```

Redefined

Undeclared

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

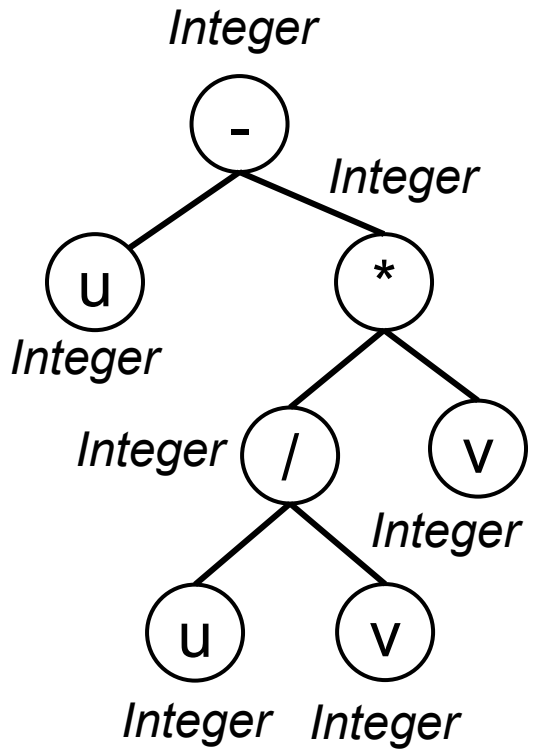
Type Checker

```
// Example Code (for Tiny)
 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: }
```

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

Type Checker
typeCheck()

case Binary Operator:
1) check if LHS is an integer
2) check if RHS is an interger
3) Then its return type is integer



Name	Type
output	Void
u	Integer
v	Integer

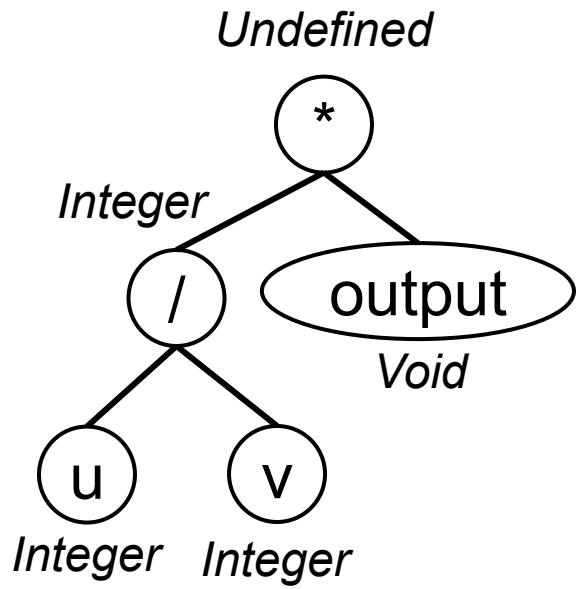
Type Checker

```
// Example Code (for Tiny)
 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/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: *
    Op: /
      Variable: name = u
      Variable: name = v
    Call: function name =
output
```

Type Checker
typeCheck()

case Binary Operator:
1) check if LHS is an integer
2) check if RHS is an interger
3) Then its return type is integer



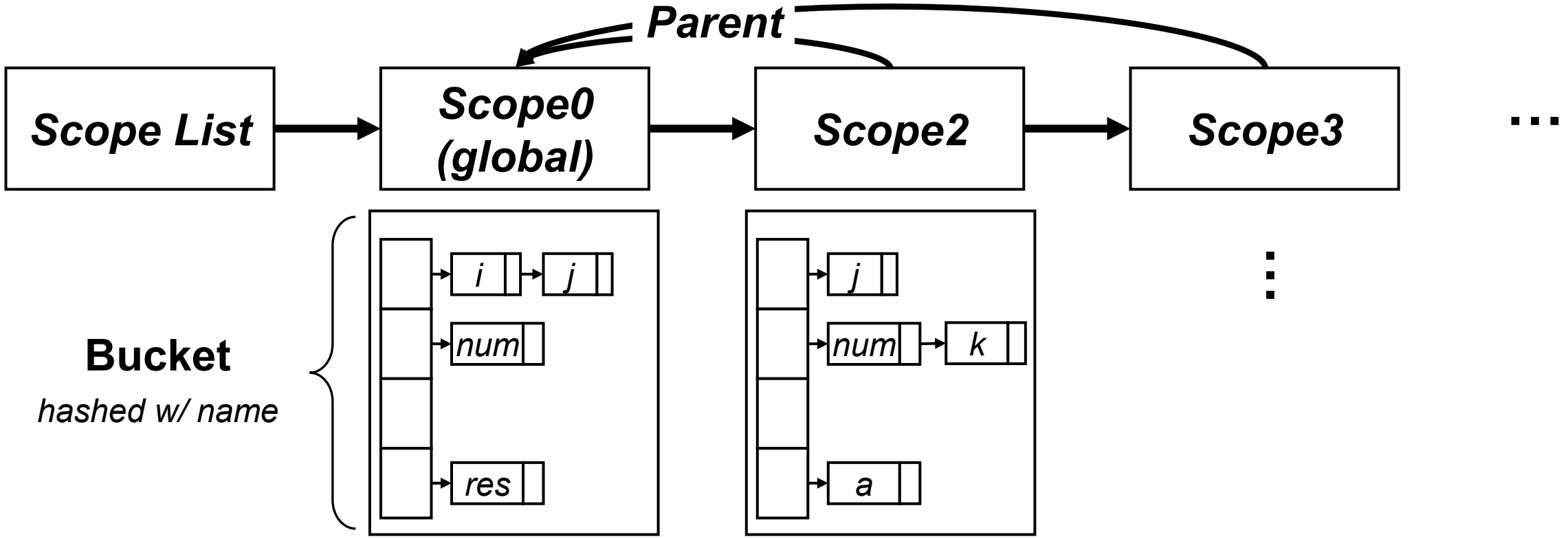
Name	Type
output	Void
u	Integer
v	Integer

Implementation

- **Implement symbol table and type checker**
- **Traverse syntax tree created by parser**
- **Files to modify**
  - globals.h
  - main.c
  - util.h, util.c
  - scan.h scan.c
  - parse.h, parse.c
  - symtab.h, symtab.c
  - analyze.h, analyze.c

Symbol Table Implementation Example

- You can implement in your own way if you want



Printing Symbol Table - 1

- **You can implement a print functions by setting and building with TraceAnalyze = TRUE in main.c**
  - We will provide example symbol table print results using the format in the following slides

- **You do not necessarily implement the print function codes**

Printing Symbol Table - 2

- **The baseline print result for symbol table**

```
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
```

Printing Symbol Table - 3

- You may implement additional print functions for debugging

```
< 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 >
 Symbol Name   Symbol Kind   Symbol Type
-------------  -----------  -------------
main           Function     void
input          Function     int
output         Function     void
gcd            Function     int


< Scopes >
 Scope Name   Nested Level   Symbol Name   Symbol Type
------------  ------------  -------------  -----------
output        1             value          int

gcd           1             u              int
gcd           1             v              int

main          1             x              int
main          1             y              int
```

Modify: main.c

- Modify code to print only semantic errors
- NO_ANALYZE, NO_CODE, TraceParse, and TraceAnalyze

```
10  /* set NO_PARSE to TRUE to get a scanner-only compiler */
11  #define NO_PARSE FALSE
12  /* set NO_ANALYZE to TRUE to get a parser-only compiler */
13  #define NO_ANALYZE FALSE
14
15  /* set NO_CODE to TRUE to get a compiler that does not
16   * generate code
17   */
18  #define NO_CODE TRUE
19
20  #include "util.h"
21  #if NO_PARSE
22      #include "scan.h"
23  #else
24      #include "parse.h"
25      #if !NO_ANALYZE
26          #include "analyze.h"
27          #if !NO_CODE
28              #include "cgen.h"
29          #endif
30      #endif
31  #endif
32
33  /* allocate global variables */
34  int lineno = 0;
35  FILE *source;
36  FILE *listing;
37  FILE *code;
38
39  /* allocate and set tracing flags */
40  int EchoSource = FALSE;
41  int TraceScan = FALSE;
42  int TraceParse = FALSE;
43  int TraceAnalyze = FALSE;
44  int TraceCode = FALSE;
```

```
10    /* set NO_PARSE to TRUE to
11    #define NO_PARSE FALSE
12    /* set NO_ANALYZE to TRUE
13    #define NO_ANALYZE FALSE
```

```
39    /* allocate and set tracing flags */
40    int EchoSource = FALSE;
41    int TraceScan = FALSE;
42    int TraceParse = FALSE;
43    int TraceAnalyze = FALSE;
44    int TraceCode = FALSE;
```

* *TraceAnalyze* helps to debug semantic analyzer

Modify: symtab.c & symtab.h

- **Symbol table implementations in Tiny**
  - Symbol table consists of BucketListRec, which has LineListRec as line number list of the symbols.
  - st_insert() inserts symbols to the table and st_lookup() returns the location of the symbol entries in the table by name (char*)

- **Scope and type information is required in C-Minus**
  - Modify st_insert and st_lookup **considering the scopes**
    - Scope has a hierarchical structure. New scopes are added within compound statements (child of upper scope) and function declarations (child of global scope)

Build Symbol Table

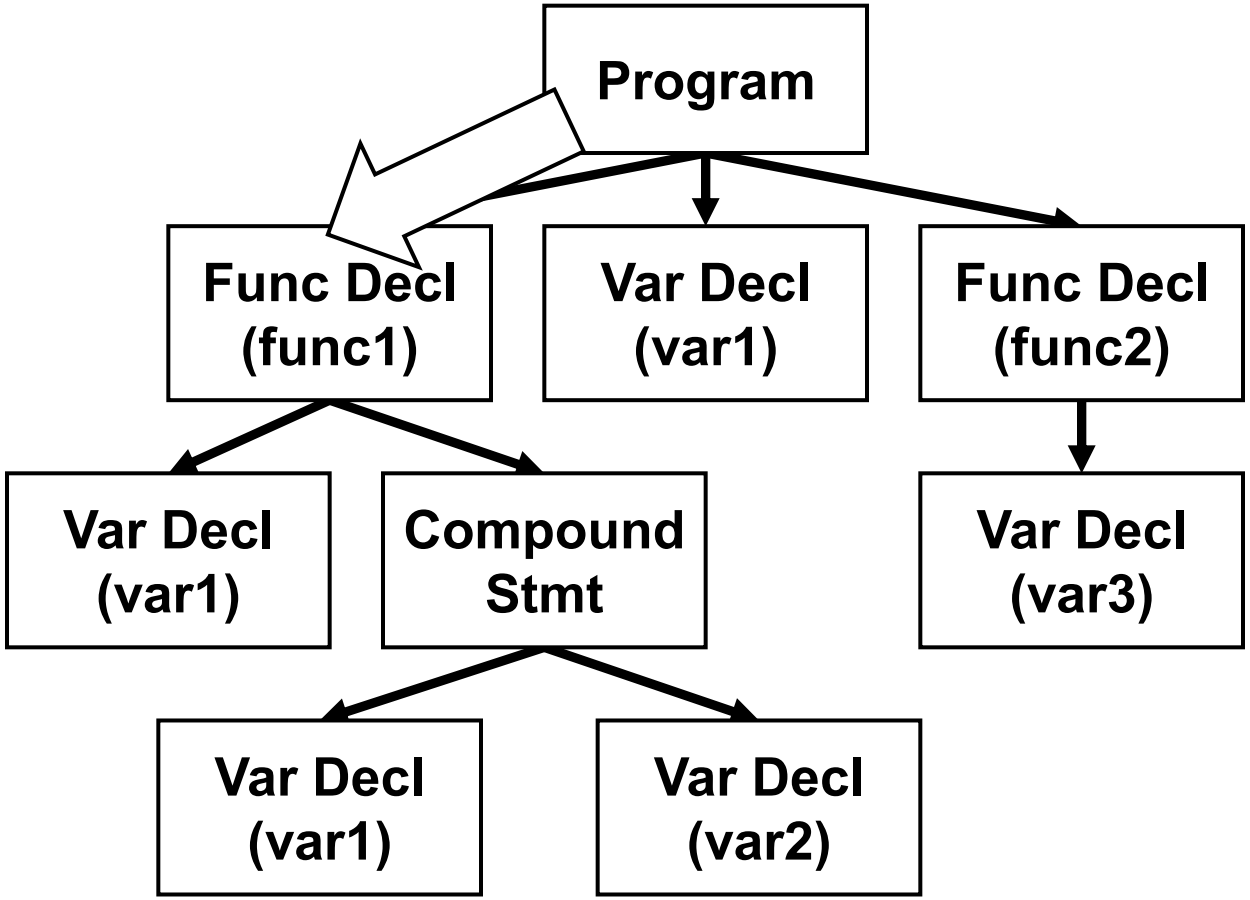
- **Five operations:**
  - Insert scope: start a new nested scope
  - Exit scope: exit the current scope
  - Find symbol(x): Search for x in the hierarchy
  - Add symbol(x): Add a symbol x to the table
  - Check scope(x): Check if x is defined in the current scope (optional)

- **We can build the symbol tables during parsing (after construction the AST)**

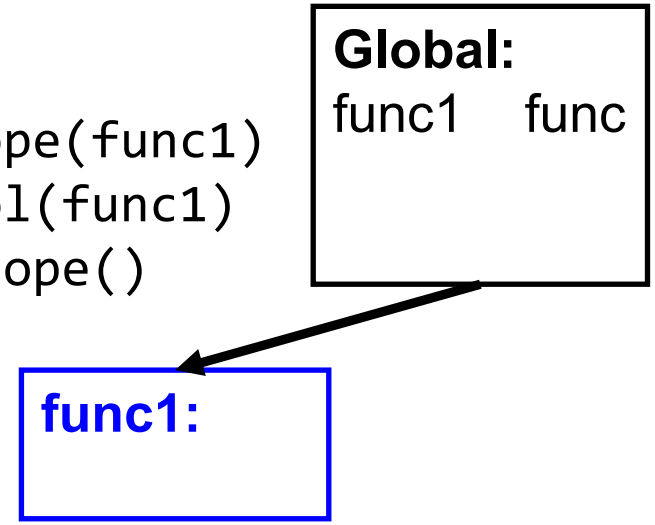
- **We should generate the symbol tables before semantic analysis**

Recall: Symbol Table Impl

- We generate the symbol table when traversing over the AST

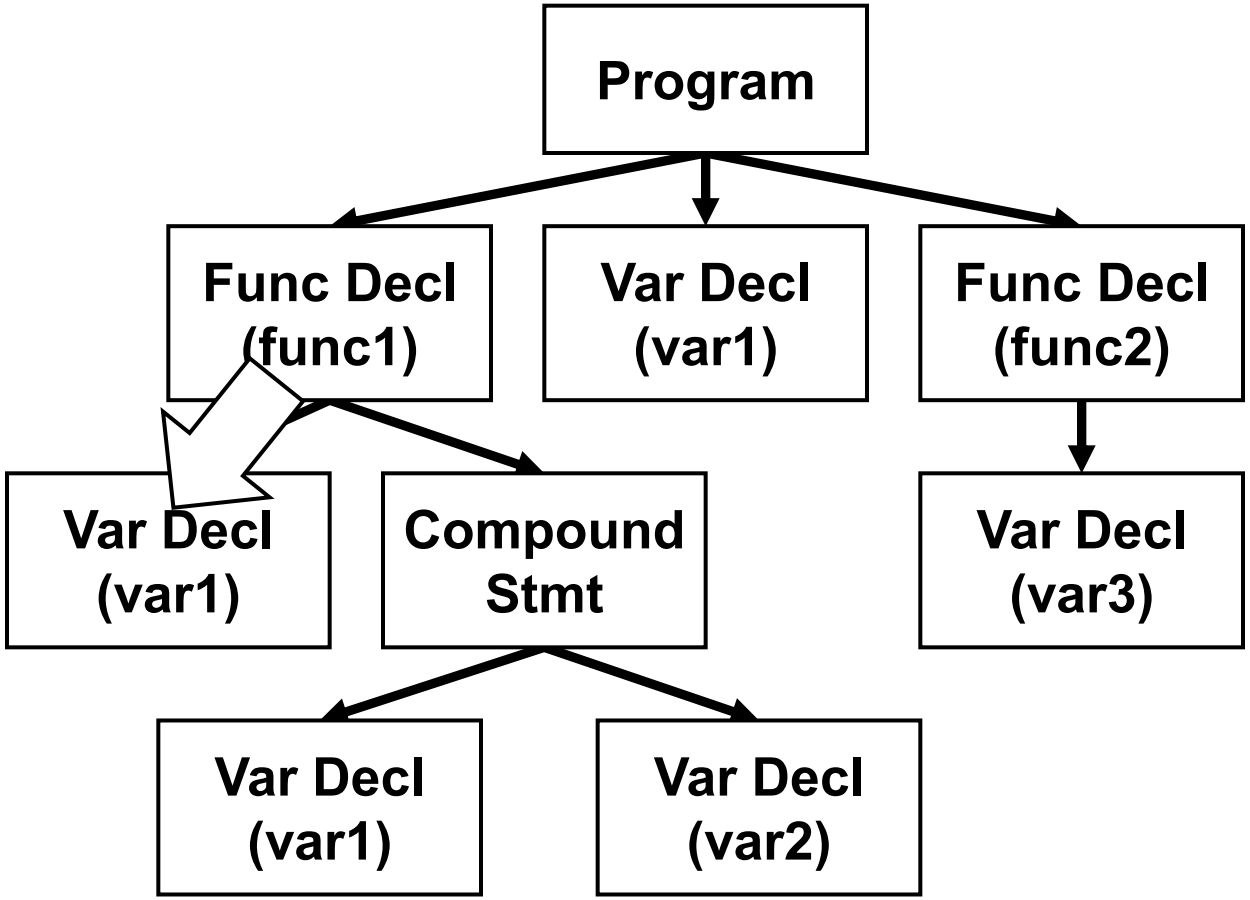


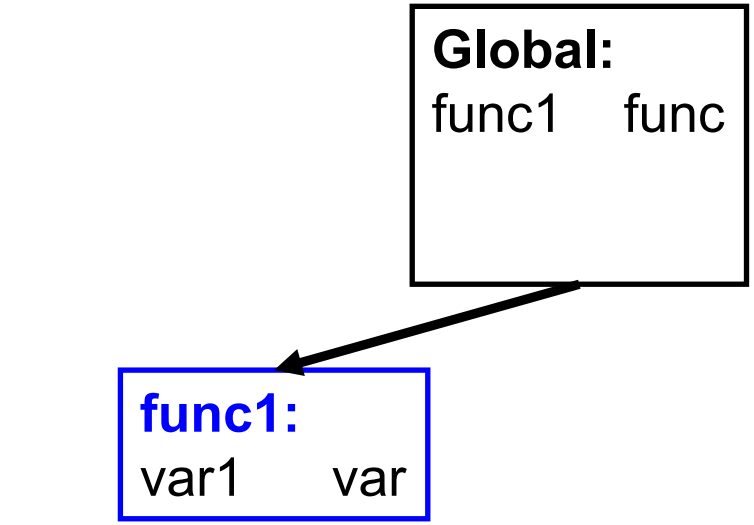

```
check_scope(func1)
add_symbol(func1)
Insert_scope()
```



Recall: Symbol Table Impl

- We generate the symbol table when traversing over the AST

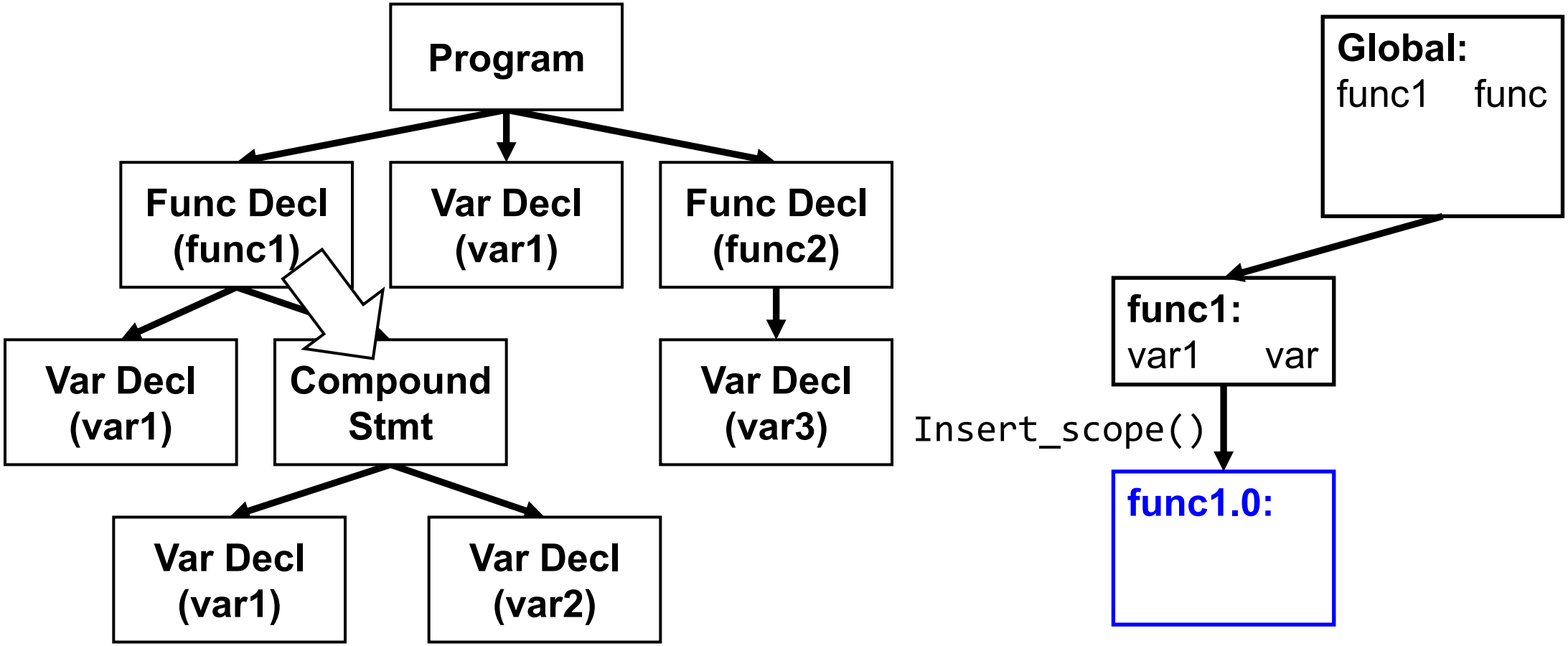





```
check_scope(var1)
add_symbol(var1)
```

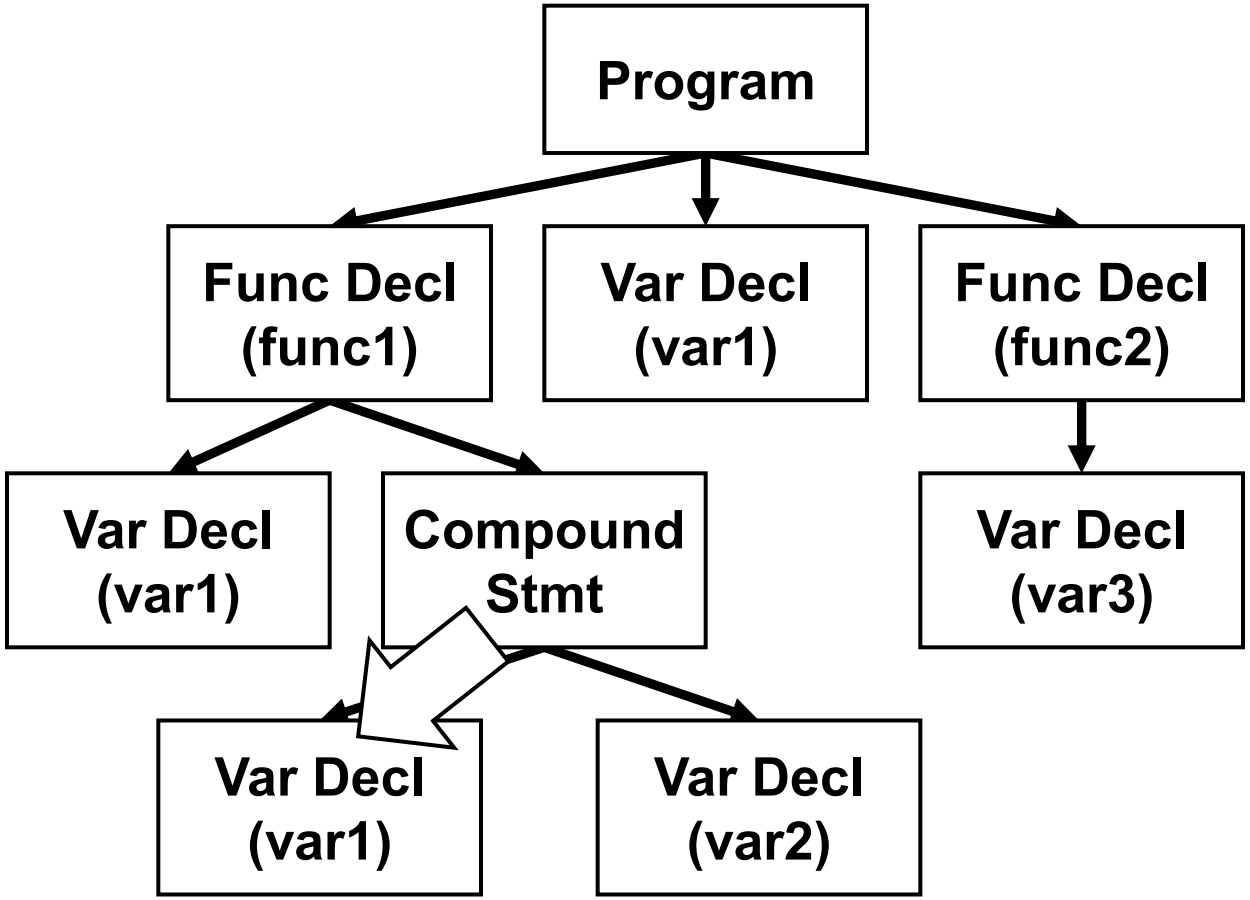
Recall: Symbol Table Impl

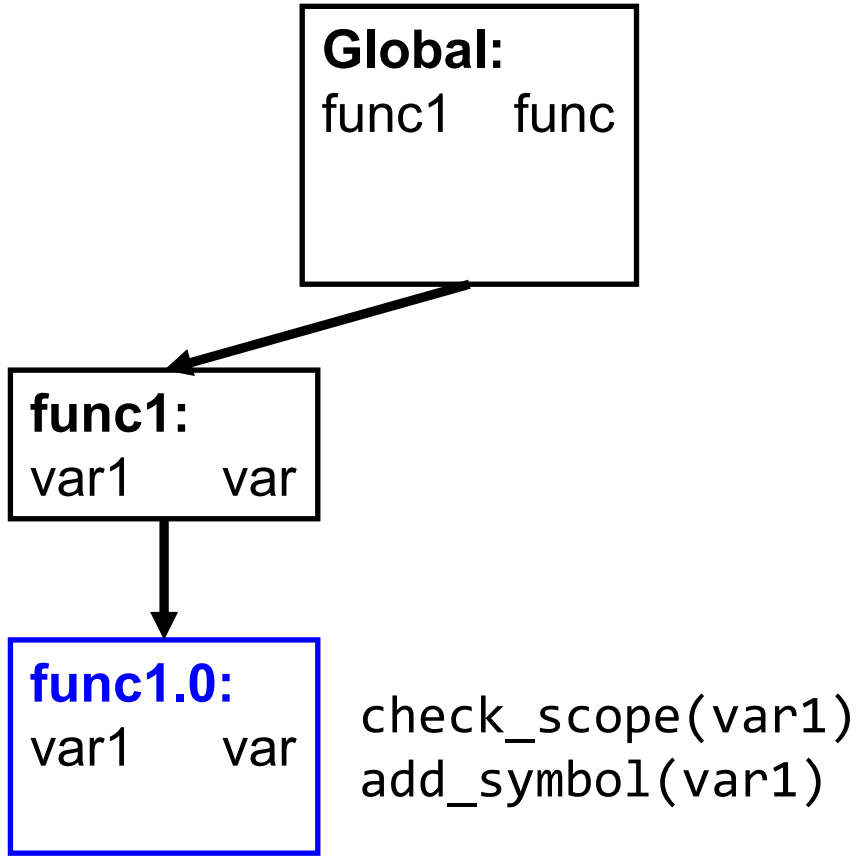
- We generate the symbol table when traversing over the AST



Recall: Symbol Table Impl

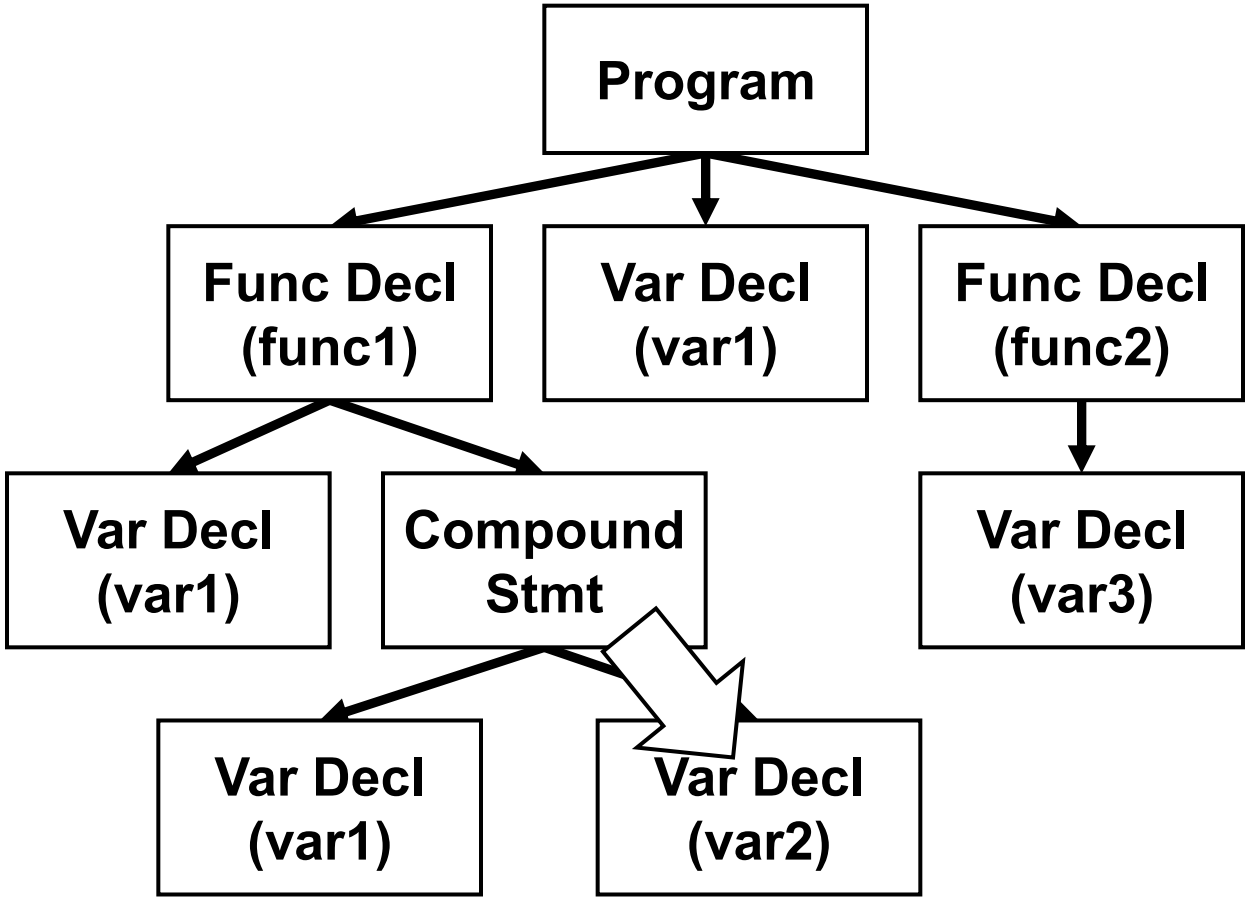
- We generate the symbol table when traversing over the AST

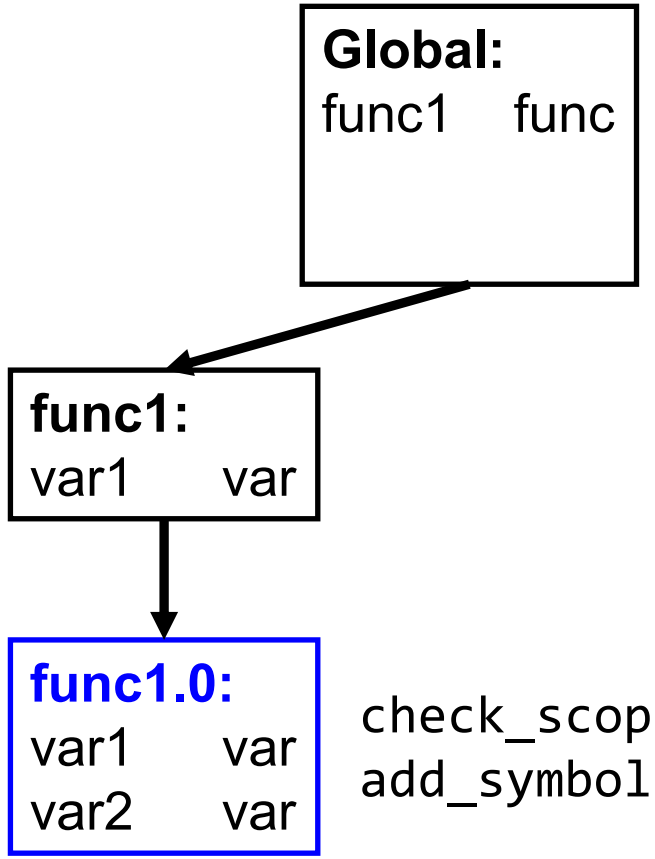




Recall: Symbol Table Impl

- We generate the symbol table when traversing over the AST






```
check_scope(var2)
add_symbol(var2)
```

Modify: analyze.c - 1

- **Modify symbol table generation (buildSymtab)**
  - Modify both preprocessing and postprocessing functions
  - The preprocessing and postprocessing should support scope analysis
  - You learned the basic methods in the class!

```
void buildSymtab(TreeNode * syntaxTree)
{ traverse(syntaxTree,insertNode,nullProc);
  if (TraceAnalyze)
  { fprintf(listing,"\nSymbol table:\n\n");
    printSymTab(listing);
  }
}
```

- **Make sure to insert built-in functions**
  - input() and output()

Modify: analyze.c - 2

- **Modify type checker (typeCheck)**
  - Modify both preprocessing and postprocessing functions
  - Define and implement the type inference rules based on the descriptions in the previous slides

- **Implement error messages for each semantic errors**

Implementation Notes

- **Building symbol tables is just an intermediate process for semantic analysis, so you can implement them however you want**
- **Variables follow scope of each compound statement.**
- **Built-in functions should be always accessible.**

Evaluation

- **Evaluation Items**
  - **Compilation** (Success / Fail): **20%**
    - Please describe in the report how to 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 ($\leq$ 5 pages)**
  - Compilation environment and method
  - Brief explanations about how to implement and how it operates
  - Examples and corresponding result screenshots

- **Format**
  - Use PDF with the filename as follows

Submission

- **Deadline: 12/01 23:59:00**
  - You cannot submit Project #1 / #2 (late submission) after the deadline
  - You do not have a late submission for Project #3
- **Submission**
  - Submit all the **source codes** in a single zip file and **report** as a pdf file
  - Format + Name:
    - Report: [Student No]_Project3.pdf
    - Code: do not modify any name and compress all the codes into a single zip file and the name should be
      - [Student No]_Project3.zip