Project #3. Semantic Analysis

Symbol Table & Type Checker

2022 Compiler

Prof. Yongjun Park

Hanyang University
Division of Computer Science & Engineering

Project Goal

- **C-Minus Semantic Analyzer Implementation**
  – **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**

  – C-Minus parser with Lex and Yacc (in project 2) should be used.
    - **Start from your source files of the previous parser project.**
    - You can implement in your own way.

  – *symtab.c, analyze.c, … -> cminus_semantic*

Hanyang University
Division of Computer Science & Engineering

Project Goal: Semantic Error Detection

- **Un/Redefined Variables and Functions**
  - Scope rules are same as C language
  - Function overloading is not allowed
- **Array Indexing Check**
  - Only *int* value can be used as an index
- **Built-in Functions**

- **Output Requirements**
  - **Error type** with its **line number**
  - **Output messages should be same as specified formats**

- **Type Check**
  - *void* variable
  - Operations such as *int*[] + *int*[], *int*[] + *int* and *void* + *void* are not allowed
    - *int* + *int* : *int*, *int* < *int* : *int*
  - assignment type
  - *if*/*while* condition
    - Only *int* value can be used for condition
  - function arguments
    - The number of parameters
    - Types
  - return type

Hanyang University
Division of Computer Science & Engineering

Output Formats

- **Please refer to 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"

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.

Output Examples

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

```
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    }
```

```
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  }
```

```
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   }
```

```
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   }
```

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

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 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

- **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

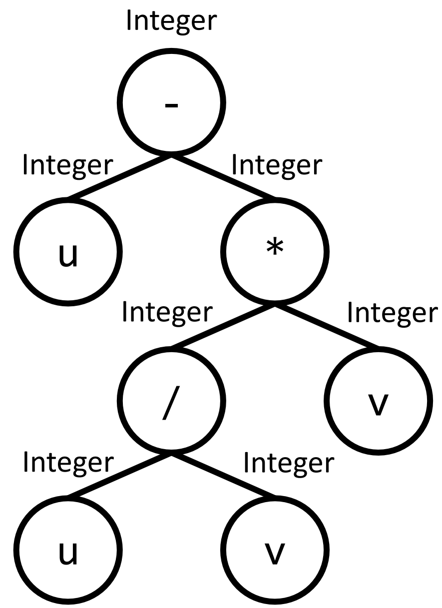
```
/* A program to perform Euclid's
  Algorithm to computer gcd */

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



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			
…	…			

Hanyang University
Division of Computer Science & Engineering

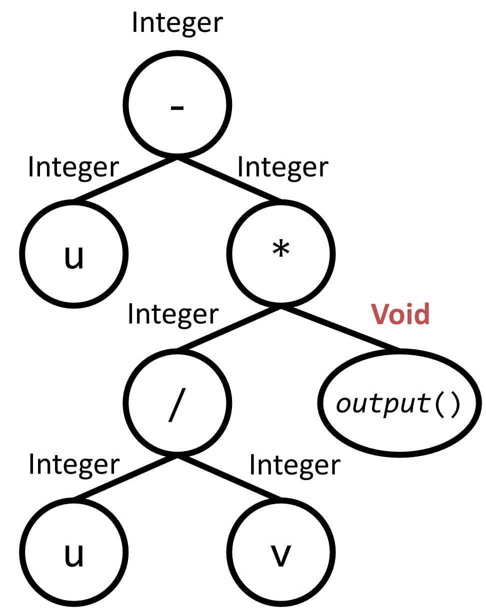
Type Checker

```
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()



Symbol Table

Name	Type			
u	Integer			
v	Integer			
output	Void			
…	…			

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!

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

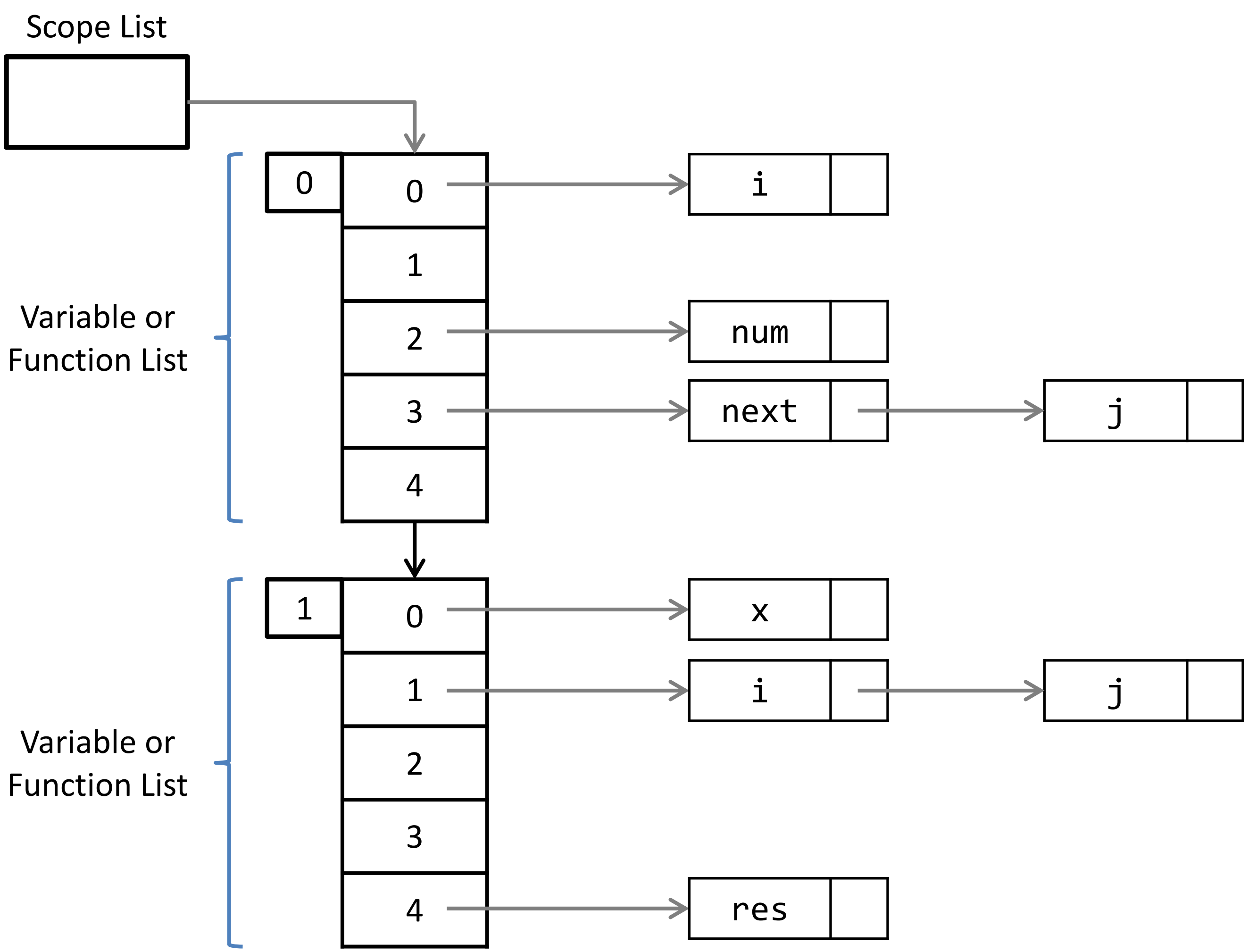
Hanyang University
Division of Computer Science & Engineering

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*

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 >
 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
```

```
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

Hint: Where to See?

- ***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;
```


* *TraceAnalyze* helps to debug semantic analyzer

Hint: Where to See?

- ***symtab.h & symtab.c***
  - 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
    - 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***
  - Implementation Hints (not mandatory, use in your own way)

```
-void st_insert( char * name, int lineno, int loc );
+void st_insert( char * scope, char * name, ExpType type, int lineno, int loc );

 /* Function st_lookup returns the memory
  * location of a variable or -1 if not found
  */
-int st_lookup ( char * name );
+BucketList st_lookup ( char * scope, char * name );
+BucketList st_lookup_excluding_parent ( char * scope, char * name );
```

```
typedef struct BucketListRec
   { char * name;
     ExpType type;
     LineList lines;
     int memloc ; /* memory location for variable
     struct BucketListRec * next;
   } * BucketList;

/* The record for each scope,
 * including name, its bucket,
 * and parent scpoe.
 */
typedef struct ScopeListRec
   { char * name;
     BucketList bucket[SIZE];
     struct ScopeListRec * parent;
```

Hint: 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
    - ***input*()**, ***output*()**
  - Implement error messages in 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 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 ($\leq$ 5 pages)**
  - Compilation environment and method
  - Brief explanations about how to implement and how it operates
  - Examples and corresponding result screenshots

- **Format**
  - Any visible formats such as PDF, MS Word, HWP, … are allowed
    - **PDF format is recommended**
  - GitLab wiki is not allowed
    - Instead, write in markdown format and submit as PDF

Submission

- **Deadline: 12/21 (Wed.) 23:59:59**

- **Submission**
  - Submit all the **source codes** and **the report** in the **3_Semantic** directory
  - https://hconnect.hanyang.ac.kr/2022_ele4029_12271/2022_ele4029_**[Student ID]**.git

- **Questions**
  - E-mail: **compiler.teachingassistant@gmail.com**
    - Please provide all questions related with projects to TAs.

Q&A