Semantic

0. Environment

- WSL Ubuntu 20.04
- gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
- flex 2.6.4
- bison (GNU Bison) 3.5.1
- GNU Make 4.2.1(Built for x86_64-pc-linux-gnu)

1. TODO

1) 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

2) 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
  - Under below
- 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

2. Output Format

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"`

3. Built-int Functions

- int input(void)
  - Returns a value of the given integer value from the user.
- void output(int value)
  - Print a value of the given argument.
- These two global functions are defined by default.

4. Symbol Table Differences

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

Tiny Symbol Table

Tiny의 Symbol Table은 따로 Scope가 존재하지 않아서 모두 Global Symbol Table로 구현이 가능하고 따로 선언도 하지 않기에, 나오는 즉시 Symbol Table을 확인할 수 있다.

C-Minus

C-Minus의 Symbol Table은 Scope도 존재하고 선언과 사용이 구별되어 있기 때문에 바로 Symbol Table을 확인해서는 안된다. 그래서 traverse함수를 이용해서 AST를 순회하면서 확인해야한다.

5. Implementation

- Symbol Table의 계층구조를 구현하기 위해서 Linked List를 사용하고, 내부의 변수도 Linked List를 이용해서 해당 Symbol Table 내부에서 존재하도록 구현해야한다. → 몇 개까지 나올 수 있을지 모르기 때문
- traverse를 진행하면서 내 Scope의 위치를 알아야 한다. Scope가 계속해서 변하기 때문에 Scope의 구조를 Symbol Table에 맞게 Linked List 등의 구조를 이용해서 구현해야한다.
- st_insert 함수는 내 현재 Scope에 맞는 Symbol Table에 원하는 변수 or 함수를 넣기 위한 함수이다. 만약 이미 존재한다면 에러를 출력한다.
- st_lookup_excluding_parent 함수는 내 Scope에서만 변수 or 함수가 존재하는지 확인하는함수로 이 함수를 이용해서 st_insert로 넣을지 말지를 결정한다.
- st_lookup 함수는 내 Scope와 내 위의 Scope들 모두를 돌면서 맞는 변수 or 함수 이름이 있는지 확인하는 함수이다. 이 함수를 사용했을 때 존재하지 않으면, 해당 프로그램에 선언되지 않았으므로 에러를 출력한다.
- 일반적으로 변수, 상수 등 type이 정해져 있는 경우에는 Type Check가 편하다. 그러나 다른 노드들에 대해서는 조금 다른 방법으로 Type Check를 해야한다. 자식노드들의 Type을 확인해서 둘 다 Int가 맞다면 해당 노드의 Type도 Int로 결정하는 방법으로 하면 Type Check를 쉽게 할 수 있다.
- Return의 경우에는 정확히 생각하지 못했는데, 우선적으로 생각한 방법은 Return이 존재하는 Scope를 확인해서 Scope의 이름을 단순히 함수 이름으로 했다고 가정했을 때, Global Symbol Table에서 함수의 선언 Type을 보고 Return과 비교할 수 있을 것 같다.
- 그 외에는 특별히 어려운 경우의 수가 없을 것 같고 단순히 구현하면 될 것 같다.