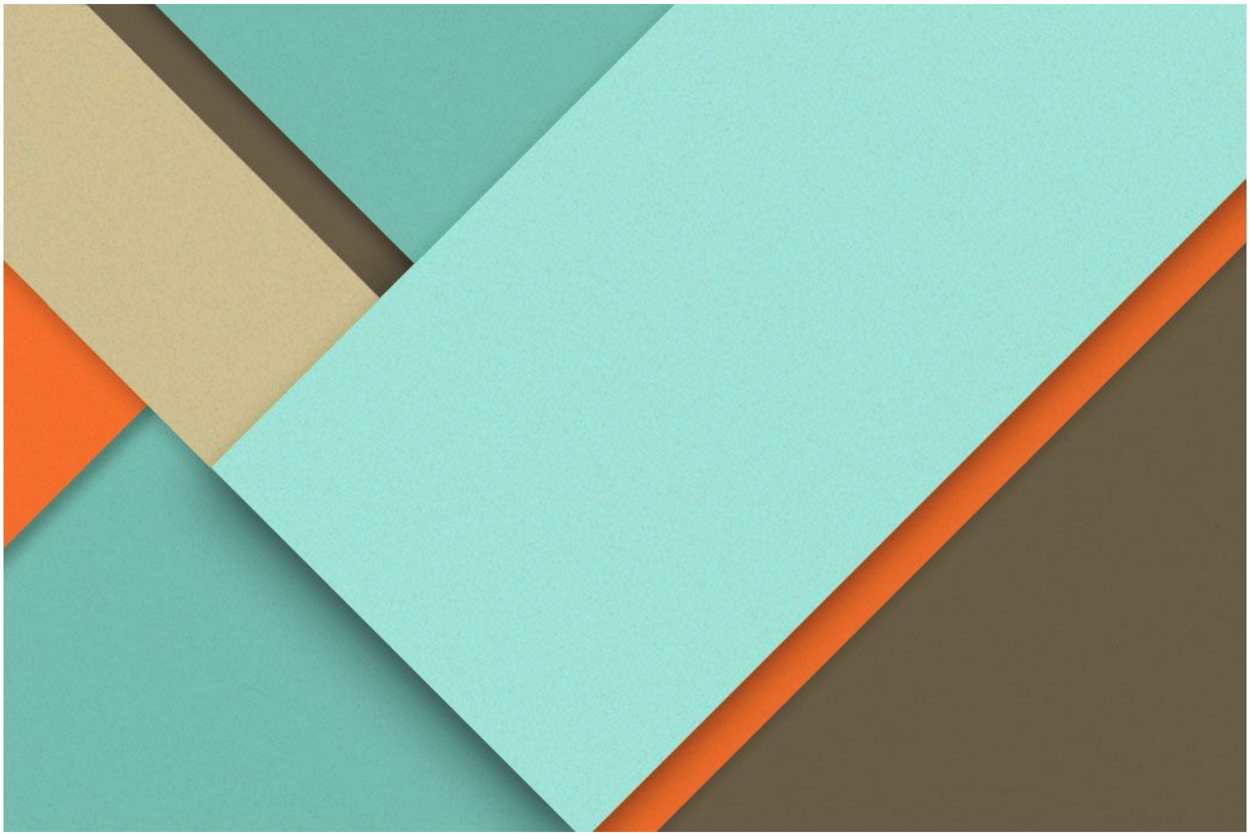


Semantic Analysis

28 march, 2018

Submitted by

R. RuhiTaj - 15CO239

Y. M . Greeshma - 15CO253

INDEX:

Overview

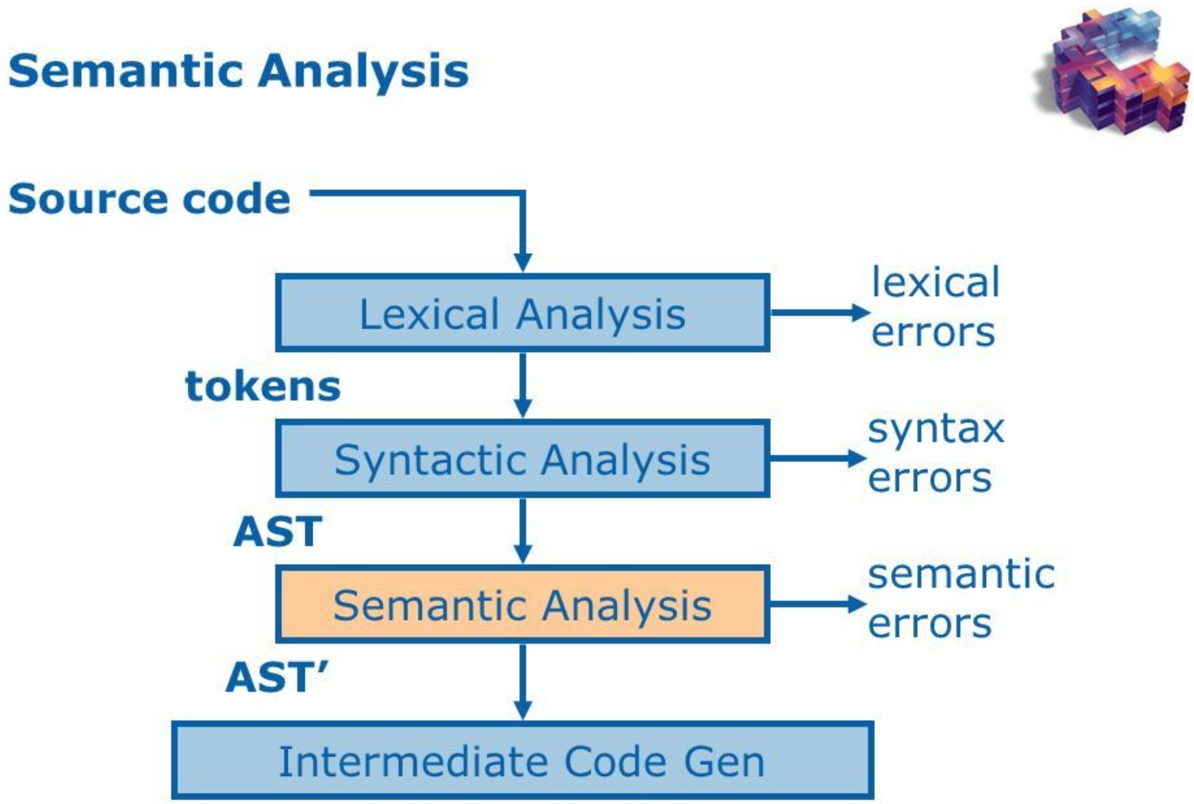
Semantic analysis or context sensitive analysis is a process in compiler construction, usually after parsing, to gather necessary semantic information from the source code. It usually includes type checking, or makes sure a variable is declared before use which is impossible to describe in the extended Backus–Naur form and thus not easily detected during parsing.

Semantic analysis is the task of ensuring that the declarations and statements of a program are semantically correct, i.e, that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used.

Compilers use semantic analysis to enforce the static semantic rules of a language .

It is hard to generalize the exact boundaries between semantic analysis and the generation of intermediate representations (or even just straight to final represenations); this demarcation is the logical boundary between the front-end of a compiler (lexical analysis and parsing) and the back-end of the compiler (intermediate representations and final code.)

Position in Compilation:



Semantic Processing

The compilation process is driven by the syntactic structure of the program as discovered by the parser Semantic routines:

- interpret meaning of the program based on its syntactic structure
- two purposes:

-- finish analysis by deriving context-sensitive information (e.g. type checking)

-- begin synthesis by generating the IR or target code

- associated with individual productions of a context free grammar or subtrees of a syntax tree

Alternatives for semantic processing

- one-pass analysis and synthesis
- one-pass compiler plus peephole
- one-pass analysis & IR synthesis + code generation pass
- multipass analysis (e.g. gcc)
- multipass synthesis (e.g. gcc)
- language-independent and retargetable (e.g. gcc) compilers Our focus in the assignments

One-pass analysis & IR synthesis + multipass analysis + multipass synthesis.

Goals

Semantic analysis is producing some sort of representation of the program, either object code or an intermediate representation of the program.

In semantic analysis, we delve even deeper to check whether they form a sensible set of instructions in the programming language.

We need to remember the type information and recall them as/where required – symbol table.

Symbol tables

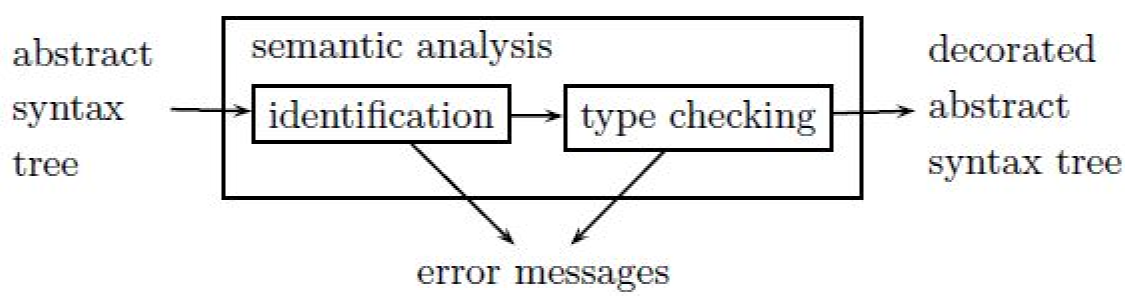
For compile-time efficiency, compilers use a symbol table:

- associates lexical names (symbols) with their attributes What items should be entered
- variable names
- defined constants
- procedure and function names

- literal constants and strings
- source text labels
- compiler-generated temporaries

Semantics

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.



CFG + semantic rules = Syntax Directed Definitions

For example: int value = “astring”;

should ***not*** issue an ***error*** in ***lexical*** and ***syntax analysis*** phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis.

The following tasks should be performed in semantic analysis:

- ***Scope resolution***
- ***Type checking :***

-- Data types are used in a manner that is consistent with their definition (i. e., only with compatible data types, only with operations that are defined for them, etc.)

- ***Label Checking*** **:**

-- Labels references in a program must exist.

- ***Array-bound checking***
- ***Flow control checks***

-- control structures must be used in their proper fashion (no GOTOs into a FORTRAN DO statement, no breaks outside a loop or switch statement, etc.)

Where Is Semantic Analysis Performed in a Compiler?

- Semantic analysis is not a separate module within a compiler. It is usually a collection of procedures called at appropriate times by the parser as the grammar requires it.
- Implementing the semantic actions is conceptually simpler in recursive descent parsing because they are simply added to the recursive procedures.
- Implementing the semantic actions in a tableaction driven LL(1) parser requires the addition of a third type of variable to the productions and the necessary software routines to process it.

About Analysis

Parsing only verifies that the program consists of tokens arranged in a syntactically valid combination. Now we'll move forward to semantic analysis, where we delve even deeper to check whether they form a sensible set of instructions in the programming language. Whereas any old noun phrase followed by some verb phrase makes a syntactically correct English sentence, a semantically correct one has subject verb agreement, proper use of gender, and the components go together to express an idea that makes sense. For a program to be semantically valid, all variables, functions, classes, etc. must be properly defined, expressions and variables must be used in ways that respect the type system, access control must be respected, and so forth. Semantic analysis is the front end's penultimate phase and the compiler's last chance to weed out incorrect programs. We need to ensure the program is sound enough to carry on to code generation.

A large part of semantic analysis consists of tracking variable/function/type declarations and type checking. In many languages, identifiers have to be declared before they're used. As the compiler encounters a new declaration, it records the type information assigned to that identifier. Then, as it continues examining the rest of the program, it verifies that the type of an identifier is respected in terms of the operations being performed. For example, the type of the right side expression of an assignment statement should match the type of the left side, and the left side needs to be a properly declared and assignable identifier. The parameters of a function should match the arguments of a function call in both number and type. The language may require that identifiers be unique, thereby forbidding two global declarations from sharing the same name. Arithmetic operands will need to be of numeric—perhaps even the exact same type (no automatic int to double conversion, for instance). These are examples of the things checked in the semantic analysis phase.

Some semantic analysis might be done right in the middle of parsing. As a particular construct is recognized, say an addition expression, the parser action could check the two operands and verify they are of numeric type and compatible for this operation. In fact, in a onepass compiler, the code is generated right then and there as well. In a compiler that runs in more than one pass (such as the one we are building for Decaf), the first pass digests the syntax and builds a parse tree representation of the program.

Types and Declarations

We begin with some basic definitions to set the stage for performing semantic analysis.

A type is a set of values and a set of operations operating on those values. There are three categories of types in most programming languages:

- ***Base types*** : int, float, double, char, bool, etc. These are the primitive types provided directly by the underlying hardware. There may be a facility for user defined variants on the base types (such as C enums).
- ***Compound types*** : arrays, pointers, records, struct s, union s, classes, and so on. These types are constructed as aggregations of the base types and simple compound types.
- ***Complex types*** : lists, stacks, queues, trees, heaps, tables, etc. You may recognize these as abstract data types. A language may or may not have support for these sort of higher level abstractions.

In many languages, a programmer must first establish the name and type of any data object (e.g., variable, function, type, etc). In addition, the programmer usually defines the lifetime. A declaration is a statement in a program that communicates this information to the compiler. The basic declaration is just a name and type, but in many languages it may include modifiers that control visibility and lifetime (i.e., static in C, private in Java). Some languages also allow declarations to initialize variables, such as in C, where you can declare and initialize in one statement.So, these three different types are operating by set of values and operations.The basic type are directly provided by the hardware so called as primitive types, Compound type build from the base type

Implementing Semantics Actions In Recursive-Descent Parsing:

- In a recursive-descent parser, there is a separate function for each nonterminal in the grammar.

–The procedures check the lookahead token against the terminals that it expects to find.

– The procedures recursively call the procedures to parse nonterminals that it expects to find.

– We now add the appropriate semantic actions that must be performed at certain points in the parsing process.

Processing Declarations

- Before any type checking can be performed, type must be stored in the symbol table. This is done while parsing the declarations.

- When processing the program's header statement:
  - – the program's identifier must be assigned the type program
  - – the current scope pointer set to point to the main program

- Processing declarations requires several actions:

- If the language allows for user-defined data types, the installation of these data types must have already occurred.

- The data types are installed in the symbol table entries for the declared identifiers.

- The identifiers are added to the abstract syntax tree.

What Does Semantic Analysis Produce?

- Part of semantic analysis is producing some sort of representation of the program, either object code or an intermediate representation of the program.
- One-pass compilers will generate object code without using an intermediate representation; code generation is part of the semantic actions performed during parsing.
- Other compilers will produce an intermediate representation during semantic analysis; most often it will be an abstract syntax tree or quadruples.

Semantic Errors

Semantic analyzer is expected to recognize:

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

Example:

E → E + T { E.value = E.value + T.value }

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories : synthesized attributes and inherited attributes.

- ### Synthesized attributes :

These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

S → ABC

If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

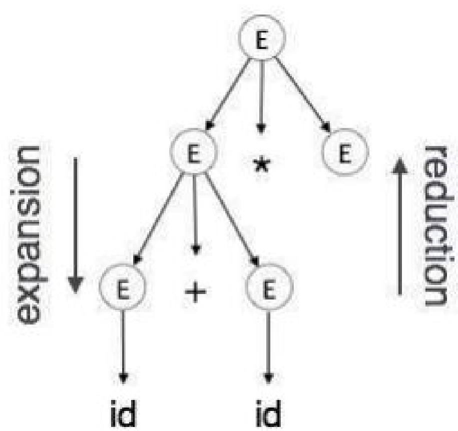
As in our previous example (E → E + T), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

- ## Inherited attributes:

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

```
S → ABC
```

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.



Expansion : When a non-terminal is expanded to terminals as per a grammatical rule

Reduction :

When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules actions.

Semantic analysis uses Syntax Directed Translations to perform the above tasks.

Semantic analyzer receives AST ***Abstract Syntax Tree*** from its previous stage ***syntax analysis***.

Semantic analyzer attaches attribute information with AST, which are called Attributed AST. Attributes are two tuple value, <attribute name, attribute value>

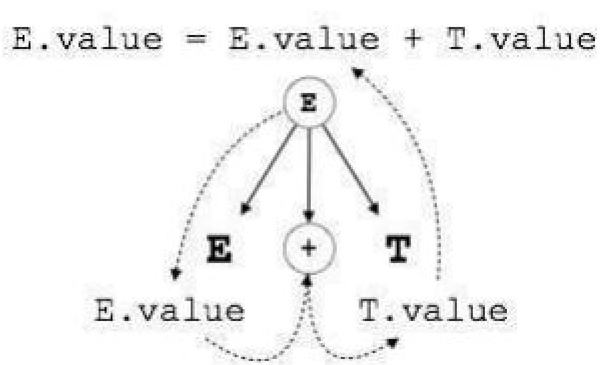
For example:

```
int value  = 5;
<type, "integer">
<presentvalue, "5">
```

For every production, we attach a semantic rule.

S-attributed SDT :

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production ***right handside.***



As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

L-attributed SDT :

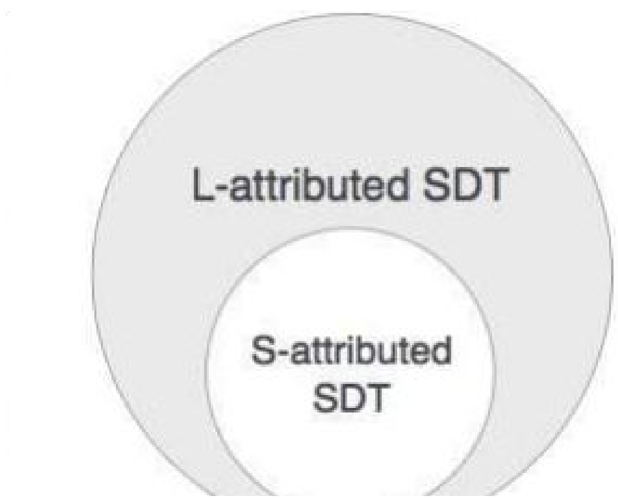
This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production

$S \rightarrow ABC$

S can take values from A, B, and C synthesized. A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner



Attribute Flow

- Pattern of information flow between attributes
- Necessary flow determined by language and parsing technique
- Example: arithmetic expressions – Can define S-attributed grammar from SLR grammar – LL(1) equivalent must have inherited attributes

Dependency between Attributes

- For each semantic action, if the action is of the following type

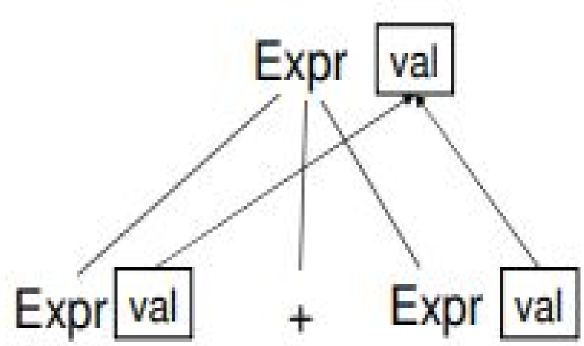
  b:= f(c1,...,cn)

  We shall say that b depends on c1,...,cn

- Looking at ALL semantic actions in the parse tree, we can build a directed dependency graph showing which attributes are dependent on which others.

Ordering rules using Dependency Graph

- For each attribute, we draw arrows to the other attributes that depend on this attribute.
- We end up with a graph detailing attribute dependency
- By convention, the graph is drawn over the parse tree



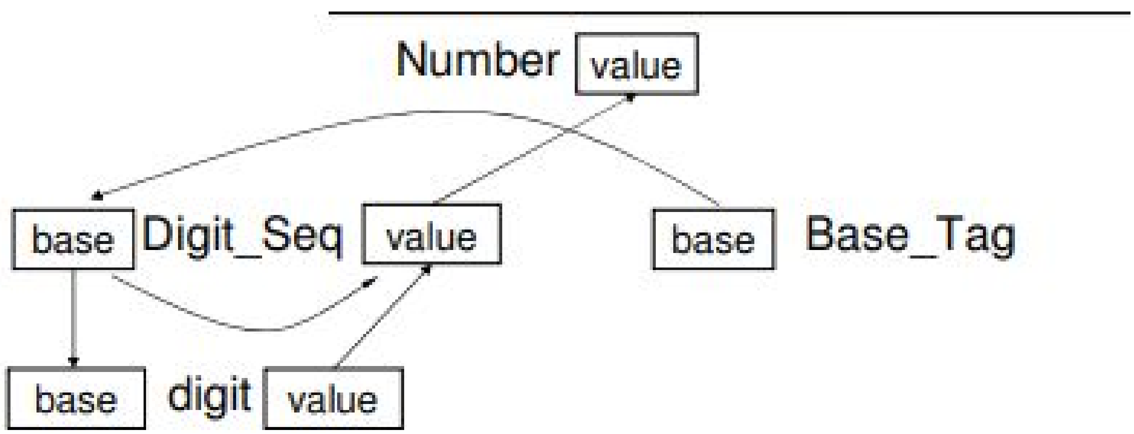
Ordering rules using Dependency Graph

- By convention, inherited nodes are drawn on the left of the symbol, and synthesized ones on the right.
- E.g., for the grammar:

Number :- Digit_Seq Base_Tag

Digit_Seq :- Digit_Seq1 digit

Digit_Seq :- digit



Circularity in Attribute Dependency Graph

- It is possible for the attribute dependency graph to have an infinite loop.
- Imagine the prior grammar with one extra dependency.
- There is now no way to order all attribute evaluations to allow resolution.
- Grammars are written so as to avoid such loops

CODE FOR SEMANTIC ANALYSIS:

Lex File:

```
%option yylineno
%{
	#include<stdio.h>
	# include <math.h>
	#include"y.tab.h"
	#include<math.h>
	#include "help.h"

%}

space[ ]
openBraces[(]
closeBraces[)]
openD[[]
closeD[]]

%%

("int"|"void")?{space}*main{space}*{openBraces}({space}?|"void"|""){closeBraces}
{the_medium_to_display(yytext,36, yylineno); return id; }

"#include"([ ]+)?((<(\\.|[^>])+>)|(\"(\\.|[^"])+\"))  { the_medium_to_display(yytext, 1,
yylineno);return HEADER;}
```

"#define"[ ]+[a-zA-z_][a-zA-z_0-9]* {the_medium_to_display(yytext, 2, yylineno); return DEFINE; }

"auto"|"register"|"static"|"extern"|"typedef" { the_medium_to_display(yytext, 3, yylineno); return storage_const;}

"short"|"long"|"signed"|"unsigned" { the_medium_to_display(yytext, 4, yylineno); return type_const;}

"void"|"char"|"int"|"float"|"double" { the_medium_to_display(yytext, 4, yylineno); return type_const;}

"const"|"volatile" { the_medium_to_display(yytext, 5, yylineno); return qual_const;}

"enum" {the_medium_to_display(yytext, 6, yylineno); return enum_const; }

"case" { the_medium_to_display(yytext, 7, yylineno); return CASE;}

"default" { the_medium_to_display(yytext, 8, yylineno); return DEFAULT;}

"if" { the_medium_to_display(yytext, 9, yylineno); return IF;}

"switch" {the_medium_to_display(yytext, 10, yylineno); return SWITCH; }

"else" { the_medium_to_display(yytext, 11, yylineno);return ELSE;}

"for" { the_medium_to_display(yytext, 12, yylineno); return FOR;}

```
"do" { the_medium_to_display(yytext, 13, yylineno); return DO;}

"while"{ the_medium_to_display(yytext, 14, yylineno); return WHILE;}

"goto"          { the_medium_to_display(yytext, 15, yylineno); return GOTO;}

"continue"      {the_medium_to_display(yytext, 16, yylineno); return CONTINUE; }

"break"                 { the_medium_to_display(yytext, 17, yylineno); return BREAK;}

"struct"|"union"        { the_medium_to_display(yytext, 18, yylineno); return struct_const;}

"return"        {the_medium_to_display(yytext, 19, yylineno); return RETURN; }

"sizeof"        { the_medium_to_display(yytext, 20, yylineno); return SIZEOF;}

"||"  {the_medium_to_display(yytext, 21, yylineno); return or_const; }

"&&"    {the_medium_to_display(yytext, 22, yylineno); return and_const; }

"..."     { the_medium_to_display(yytext, 23, yylineno); return param_const;}

"=="|"!="       { the_medium_to_display(yytext, 24, yylineno); return eq_const;}

"<="|">="       {the_medium_to_display(yytext, 25, yylineno); return rel_const; }
```

```
">>"|"<<"       { the_medium_to_display(yytext, 26, yylineno); return shift_const;}

"++"|"--"       { the_medium_to_display(yytext, 27, yylineno); return inc_const;}

"->"            {the_medium_to_display(yytext, 28, yylineno); return point_const; }

";" { the_medium_to_display(yytext, 29, yylineno);set = set + 1; return yytext[0];}

"="|","|"("|")"|"["|"]"|"*"|"+"|"-"|"/"|"?"|":"|"&"|"|"|"^"|"!"|"~"|"%"|"<"|">"       {
the_medium_to_display(yytext, 29, yylineno); return yytext[0];}

"{" { letsPush(); the_medium_to_display(yytext, 29, yylineno); return yytext[0];}

"}" { letsPop(); the_medium_to_display(yytext, 29, yylineno); return yytext[0];}

"*="|"/="|"+="|"%="|">>="|"-="|"<<="|"&="|"^="|"|=" {the_medium_to_display(yytext, 30,
yylineno); return PUNC; }

[0-9]+ { the_medium_to_display(yytext, 31, yylineno); return int_const;}

[0-9]+"."[0-9]+         {the_medium_to_display(yytext, 32, yylineno); return float_const; }

"'"."'"|"'"[a-zA-z_][a-zA-z_0-9]*"'"    { the_medium_to_display(yytext, 33, yylineno);  return
char_const ;}

[a-zA-z_][a-zA-z_0-9]*("["([1-9][0-9]+)|([a-zA-z_][a-zA-z_0-9]*)"]")+   {
the_medium_to_display(yytext, 37, yylineno); return id;}
```

```
"printf"|"getchar" { the_medium_to_display(yytext, 38, yylineno); return id;}


[a-zA-z_][a-zA-z_0-9]*       { int k = the_medium_to_display(yytext, 34, yylineno);
if(k==1){ printf("\n\n%s-in\n\n",yytext); return int_id; }else if(k==2){ return float_id; } else
if(k==3){ return char_id; }  else { printf("\n\n%s-out\n\n",yytext); return id;}}



\"(\\.|[^\"])*\"   { the_medium_to_display(yytext, 35, yylineno);  return string;}


"//"(\\.|[^\n])*[\n]                ;
[/][*]([^*]|[*]*[^*/])*[*]+[/]  ;
[ \t\n]                             ;


%%


int yywrap(void)
{
   return 1;
}
```

Yacc File :

```
%{
        #include<stdio.h>
        # include <string.h>
        # include <stdlib.h>
        int yylex(void);
        int yyerror(const char *s);
```

```
int success = 1;
void check(int a, int b){
if(a==2 || b== 2){
printf("Semantic error\n");
printf("Cannot add expression containing character variable\n\n");
}
else if(a!=b){
printf("Semantic error\n");
printf("Cannot add floating and integer type variables\n\n");
}
}
void checkAssign(int a, int b){
if(a== b-10){}
else if((a==2 || b-10==2) && (a==1 && b-10==3)){
printf("Semantic error\n");
printf("Cannot assign different variables variables\n\n");
}
}
%}

%token int_const char_const float_const int_id float_id char_id id string enumeration_const storage_const type_const qual_const struct_const enum_const DEFINE

%token IF FOR DO WHILE BREAK SWITCH CONTINUE RETURN CASE DEFAULT GOTO SIZEOF PUNC or_const and_const eq_const shift_const rel_const inc_const

%token point_const param_const ELSE HEADER

%left '+' '-'

%left '*' '/'
```

```
%nonassoc "then"
%nonassoc ELSE
%define parse.error verbose
%start program_unit

%%

program_unit            :       HEADER program_unit
                                | DEFINE primary_exp program_unit

                                | translation_unit {$$ = $1;}

                                ;
translation_unit        : external_decl {$$ = $1;}

                                | translation_unit external_decl

                                ;
external_decl           : function_definition
                                | decl {$$ = $1;}
                                ;
function_definition     : decl_specs declarator decl_list compound_stat
                                | declarator decl_list compound_stat
                                | decl_specs declarator
compound_stat
                                | declarator compound_stat
                                ;
decl                            : decl_specs init_declarator_list ';' {$$ = $1;}
```

| decl_specs ';'

;

decl_list : decl

| decl_list decl

;

decl_specs : storage_class_spec decl_specs

| storage_class_spec

| type_spec decl_specs

| type_spec

| type_qualifier decl_specs

| type_qualifier

;

storage_class_spec : storage_const

;

type_spec : type_const

| struct_or_union_spec

| enum_spec

| typedef_name

;

type_qualifier : qual_const

;

struct_or_union_spec : struct_or_union id '{' struct_decl_list '}'

| struct_or_union int_id '{' struct_decl_list '}'

| struct_or_union float_id '{' struct_decl_list '}'

```
                                        | struct_or_union char_id '{'
struct_decl_list '}'
                                        | struct_or_union '{' struct_decl_list '}'
                                        | struct_or_union id
                                        | struct_or_union char_id
                                        | struct_or_union float_id
                                        | struct_or_union int_id
                                        ;
struct_or_union                     : struct_const
                                        ;
struct_decl_list                : struct_decl
                                        | struct_decl_list struct_decl
                                        ;
init_declarator_list        : init_declarator {$$ = $1;}
                                        | init_declarator_list ',' init_declarator
                                        ;
init_declarator                     : declarator {$$ = $1;}
                                        | declarator '=' initializer {
checkAssign($1,$3);}
                                        ;
struct_decl                         : spec_qualifier_list struct_declarator_list ';'
                                        ;
spec_qualifier_list             : type_spec spec_qualifier_list
                                        | type_spec
                                        | type_qualifier spec_qualifier_list
                                        | type_qualifier
                                        ;
struct_declarator_list          : struct_declarator
```

```
                                    | struct_declarator_list ','
struct_declarator
                                    ;
struct_declarator           : declarator { $$ = $1;}
                                    | declarator ':' const_exp
                                    | ':' const_exp
                                    ;
enum_spec                           : enum_const id '{' enumerator_list '}'
                                    | enum_const int_id '{' enumerator_list '}'
                                    | enum_const char_id '{' enumerator_list
'}'
                                    | enum_const float_id '{' enumerator_list
'}'
                                    | enum_const '{' enumerator_list '}'
                                    | enum_const id
                                    | enum_const int_id
                                    | enum_const float_id
                                    | enum_const char_id
                                    ;
enumerator_list                     : enumerator
                                    | enumerator_list ',' enumerator
                                    ;
enumerator                          : id
                                    | int_id
                                    | char_id
                                    | float_id
                                    | id '=' const_exp  { checkAssign($1,$3);}
```

```
                                        | int_id '=' const_exp
{checkAssign($1,$3);}
                                        | float_id '=' const_exp
{checkAssign($1,$3);}
                                        | char_id '=' const_exp
{checkAssign($1,$3);}
                                        ;
declarator                          : pointer direct_declarator
                                        | direct_declarator {$$ = $1;}
                                        ;
direct_declarator               : id
                                        | int_id {  $$ = 1;}
                                        | char_id {$$ = 2;}
                                        | float_id {$$ = 3;}

                                        | '(' declarator ')'

                                        | direct_declarator '[' const_exp ']'

                                        | direct_declarator '[' ']'
                                        | direct_declarator '(' param_type_list ')'

                                        | direct_declarator '(' id_list ')'

                                        | direct_declarator '(' ')'

                                        ;
pointer                                 : '*' type_qualifier_list
                                        | '*'
                                        | '*' type_qualifier_list pointer
```

```
                                    | '*' pointer
                                    ;
type_qualifier_list         : type_qualifier
                                    | type_qualifier_list type_qualifier
                                    ;
param_type_list                 : param_list
                                    | param_list ',' param_const
                                    ;
param_list                      : param_decl
                                    | param_list ',' param_decl
                                    ;
param_decl                      : decl_specs declarator
                                    | decl_specs abstract_declarator
                                    | decl_specs
                                    ;
id_list                         : id
                                    | int_id
                                    | char_id
                                    | float_id
                                    | id_list ',' id
                                    | id_list ',' int_id
                                    | id_list ',' float_id
                                    | id_list ',' char_id
                                    ;
initializer                     : assignment_exp
                                    | '{' initializer_list '}'
                                    | '{' initializer_list ',' '}'
```

```
                                                ;
initializer_list            : initializer
                                                | initializer_list ',' initializer
                                                ;
type_name                               : spec_qualifier_list abstract_declarator
                                                | spec_qualifier_list
                                                ;
abstract_declarator                 : pointer
                                                | pointer direct_abstract_declarator
                                                |        direct_abstract_declarator
                                                ;
direct_abstract_declarator  : '(' abstract_declarator ')'
                                                | direct_abstract_declarator '[' const_exp
']'
                                                | '[' const_exp ']'
                                                | direct_abstract_declarator '[' ']'
                                                | '[' ']'
                                                | direct_abstract_declarator '('
param_type_list ')'
                                                | '(' param_type_list ')'
                                                | direct_abstract_declarator '(' ')'
                                                | '(' ')'
                                                ;
typedef_name                            : 't'
                                                ;
stat                                    : labeled_stat
```

```
                                        | exp_stat {$$ = $1;}

                                        | compound_stat {$$ = $1;}

                                        | selection_stat

                                        | iteration_stat
                                        | jump_stat
                                        ;
labeled_stat                  : id ':' stat
                                        | int_id ':' stat
                                        | char_id ':' stat
                                        | float_id ':' stat
                                        | CASE const_exp ':' stat
                                        | DEFAULT ':' stat
                                        ;
exp_stat                           : exp ';' {$$ = $1;}
                                        | ';'
                                        ;
compound_stat                      : '{' decl_list stat_list '}'

                                        | '{' stat_list '}'{$$ = $2;}

                                        | '{' decl_list   '}'

                                        | '{' '}'


                                        ;
stat_list                          : stat {$$ = $1;}
```

```
                                        | stat_list stat

                                        ;
selection_stat                  : IF '(' exp ')' stat
                          %prec "then"
                                        | IF '(' exp ')' stat ELSE stat
                                        | SWITCH '(' exp ')' stat
                                        ;
iteration_stat              : WHILE '(' exp ')' stat
                                        | DO stat WHILE '(' exp ')' ';'
                                        | FOR '(' exp ';' exp ';' exp ')' stat
                                        | FOR '(' exp ';' exp ';'       ')' stat
                                        | FOR '(' exp ';' ';' exp ')' stat
                                        | FOR '(' exp ';' ';' ')' stat
                                        | FOR '(' ';' exp ';' exp ')' stat
                                        | FOR '(' ';' exp ';' ')' stat
                                        | FOR '(' ';' ';' exp ')' stat
                                        | FOR '(' ';' ';' ')' stat
                                        ;
jump_stat                       : GOTO id ';'
                                        | GOTO int_id ';'
                                        | GOTO char_id ';'
                                        | GOTO float_id ';'
                                        | CONTINUE ';'
                                        | BREAK ';'
                                        | RETURN exp ';'
                                        | RETURN ';'
                                        ;
```

```
exp                         : assignment_exp {$$ = $1;}
                            | exp ',' assignment_exp
                            ;
assignment_exp              : conditional_exp {$$ = $1;}
                            | unary_exp assignment_operator
assignment_exp {printf("\n\nEntered\n\n");  checkAssign($1,$3);}
                            ;
assignment_operator         : PUNC
                            | '='
                            ;
conditional_exp             : logical_or_exp {$$ = $1;}
                            | logical_or_exp '?' exp ':'
conditional_exp
                            ;
const_exp                   : conditional_exp
                            ;
logical_or_exp              : logical_and_exp {$$ = $1;}
                            | logical_or_exp or_const
logical_and_exp
                            ;
logical_and_exp             : inclusive_or_exp {$$ = $1;}
                            | logical_and_exp and_const
inclusive_or_exp
                            ;
inclusive_or_exp            : exclusive_or_exp {$$ = $1;}
                            | inclusive_or_exp '|' exclusive_or_exp
                            ;
exclusive_or_exp            : and_exp {$$ = $1;}
```

```
                                        | exclusive_or_exp '^' and_exp
                                        ;
and_exp                                 : equality_exp {$$ = $1;}
                                        | and_exp '&' equality_exp
                                        ;
equality_exp                  : relational_exp {$$ = $1;}
                                        | equality_exp eq_const relational_exp
                                        ;
relational_exp                     : shift_expression {$$ = $1;}
                                        | relational_exp '<' shift_expression
                                        | relational_exp '>' shift_expression
                                        | relational_exp rel_const
shift_expression
                                        ;
shift_expression              : additive_exp {$$ = $1;}
                                        | shift_expression shift_const
additive_exp
                                        ;
additive_exp                  : mult_exp {$$ = $1;}
                                        | additive_exp '+' mult_exp {
check($1,$3);}
                                        | additive_exp '-' mult_exp
                                        ;
mult_exp                           : cast_exp {$$ = $1;}
                                        | mult_exp '*' cast_exp
                                        | mult_exp '/' cast_exp
                                        | mult_exp '%' cast_exp
                                        ;
```

```
cast_exp            : unary_exp {$$ = $1;}
                        | '(' type_name ')' cast_exp
                        ;
unary_exp           : postfix_exp {$$ = $1;}
                        | inc_const unary_exp
                        | unary_operator cast_exp
                        | SIZEOF unary_exp
                        | SIZEOF '(' type_name ')'
                        ;
unary_operator      : '&' | '*' | '+' | '-' | '~' | '!'
                        ;
postfix_exp         : primary_exp       {$$ = $1;}
                        | postfix_exp '[' exp ']'
                        | postfix_exp '(' argument_exp_list ')'
                        | postfix_exp '(' ')'
                        | postfix_exp '.' id
                        | postfix_exp '.' int_id
                        | postfix_exp '.' char_id
                        | postfix_exp '.' float_id
                        | postfix_exp point_const id
                        | postfix_exp point_const int_id
                        | postfix_exp point_const float_id
                        | postfix_exp point_const char_id
                        | postfix_exp inc_const
                        ;
primary_exp         : id {$$ = $1; }
                        | int_id { $$ = 1;}
```

```
                                    | char_id  { $$ = 2;}
                                    | float_id  { $$ = 3;}

                                    | consts {$$ = $1; }

                                    | string

                                    | '(' exp ')'
                                    ;
argument_exp_list           : assignment_exp
                                    | argument_exp_list ',' assignment_exp
                                    ;
consts                              : int_const     {$$ = 11;}

                                    | char_const {$$ = 12;}
                                    | float_const {$$ = 13;}
                                    | enumeration_const
                                    ;
%%

int yyerror(const char *msg)
{
        extern int yylineno;
        printf("There is an error and failed to parse.\nLine Number: %d
%s\n",yylineno,msg);
        success = 0;
        return 0;
}
```

```
int main()
{
    yyparse();
    if(success)
            printf("\n\n\nParsing Successful\n");
theTable();
    return 0;
}
```

Symbol Table Code - help.h :

```
char* everything[45][50];
int i,count[45] = {0};
char* dataType[50][20];
int lineNo[500];
int set=0;
float scope= 0.0 ;
int whoIsInSet[500] = {0};
int assign[500];
float IdScope[500] = {-1};
int lineNoOfId[600];
int Dim[700] = {-1};
int dimk[500];
int dimGlobe=-1;
```

```
char littleScope[500];
int scopeTrack = -1;

float sendToInc(float n){

return round(n)+1.000000;

}

void letsPush(){


scopeTrack = scopeTrack + 1;
littleScope[scopeTrack] = "{";


if(scopeTrack==0){


scope = sendToInc(scope);

}

scope = scope + 0.01;

}
```

```
void letsPop(){


scopeTrack = scopeTrack -1;
scope = scope - 0.01;

}

int checkC(float a, float b){

if((round(a)== round(b)) && ( a >= b)){
return 1;
}

return -1;

}
/*

1. int
2. float
3. char
4. double
5. void
6. struct
7. union
```

8. int*

9. char*

10. float*

11. double*

12. struct*

13. int[]

14. float[]

15. char[]

16. double[]

```
void setLineNo(int line, char *word){

if(word == "int"){

lineNo[0]= line;

}
else if(word == "float"){

lineNo[1]= line;

}
else if(word == "char"){

lineNo[2]= line;
```

```
}
else if(word == "double"){

lineNo[3]= line;

}
else if(word == "void"){

lineNo[4]= line;

}

else if(word == "struct"){

lineNo[5] = line;

}

}

*/


void namethe(int no){

switch(no){
```

```
case 1: printf("\n|\t\t\t    |\n|\t\t\t    |\n|  Header_File           ");
                    break;
case 2: printf("\n|\t\t\t    |\n|\t\t\t    |\n|  Define                ");
                    break;
case 3: printf("\n|\t\t\t    |\n|\t\t\t    |\n|  Storage_Constant      ");
                    break;
case 4:             printf("\n|\t\t\t    |\n|\t\t\t    |\n|  Type_Constant         ");
                    break;
case 5: printf("\n|\t\t\t    |\n|\t\t\t    |\n|  Qual_Constant         ");
                    break;
case 6: printf("\n|\t\t\t    |\n|\t\t\t    |\n|  Enum_Constant         ");
                    break;
case 7: printf("\n|\t\t\t    |\n|\t\t\t    |\n|  Case                  ");
                    break;
case 8: printf("\n|\t\t\t    |\n|\t\t\t    |\n|  Default               ");
                    break;
case 9: printf("\n|\t\t\t    |\n|\t\t\t    |\n|  If                    ");
                    break;
case 10:printf("\n|\t\t\t    |\n|\t\t\t    |\n|  Switch                ");
                    break;
case 11:printf("\n|\t\t\t    |\n|\t\t\t    |\n|  Else                  ");
                    break;
case 12:printf("\n|\t\t\t    |\n|\t\t\t    |\n|  For                   ");
                    break;
case 13:printf("\n|\t\t\t    |\n|\t\t\t    |\n|  Do                    ");
                    break;
case 14:printf("\n|\t\t\t    |\n|\t\t\t    |\n|  While                 ");
```

```
                break;

case 15:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  Goto              ");
                break;

case 16:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  Continue          ");
                break;

case 17:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  Break             ");
                break;

case 18:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  Struct_Constant   ");
                break;

case 19:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  Return            ");
                break;

case 20:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  Size_Of           ");
                break;

case 21:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  Or_Const          ");
                break;
case 22:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  And_Const         ");
                break;
case 23:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  Param_Const       ");
                break;
case 24:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  Equi_Const        ");
```

```
            break;
case 25:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  Rel_Const            ");
            break;
case 26:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  Shift_Const          ");
            break;
case 27:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  Inc_Const            ");
            break;
case 28:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  Point_Const          ");
            break;

case 29:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  Just_Symbol          ");
            break;

case 30:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  Punc                 ");
            break;
case 31:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  int_Const            ");
            break;
case 32:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  float_Const          ");
            break;
case 33:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  Char_Const           ");
            break;
case 34:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  id                   ");
            break;
case 35:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  string               ");
            break;
case 36:printf("\n|\t\t\t   |\n|\t\t\t   |\n|  Main_Function        ");
            break;
```

```
case 37:printf("\n|\t\t\t    |\n|\t\t\t    |\n|  Array                 ");
                    break;
case 38:printf("\n|\t\t\t    |\n|\t\t\t    |\n|  Library_Function       ");
                    break;
default: break;
}

}

void thename(int no){



printf("\n|\t\t\t    |\n|\t\t\t    |\n|  ");
int k  = strlen (everything[34][no]);
printf("%s",everything[34][no]);
for(i=0;i<25-k;i++){
printf(" ");}

}

void theScope(int n){




}
```

```
int the_medium_to_display(char* word, int no, int lineAtCode){



 int y = strlen(word);

everything[no][count[no]] = (char*)malloc(y*sizeof(char));

int track = -1;

if(no == 37){
the_medium_to_display( word, 34, lineAtCode);
track = 1;
}

int i,stop=0;

for(i=0;i<count[no];i++){

if( strcmp(everything[no][i], word) ==0 ){

if( no !=34){
stop=1;
return whoIsInSet[lineNo[i]];
break;
i=count[no];
}
```

```
else

{
if(checkC(scope, IdScope[i])==1){

if(whoIsInSet!=0){
printf("Semantic Error\n");
printf("Redeclaration of the identifier");
printf
}

stop=1;
return whoIsInSet[lineNo[i]];
break;
i=count[no];
}
}
}

}

if(stop==0){

int g;

        for(g=0;g<y;g++){
```

```
    everything[no][count[no]][g] = word[g];
    }

if (no == 34){

lineNo[count[no]] = set;
if(whoIsInSet[set] == 0){

printf("\n\nSemantic Error found at line No - %d!\n",lineAtCode);
printf("%s is not defined\n\n",word);

}

IdScope[count[no]] = scope;

if(count[no]!=0){

/*
if(IdScope[count[no]-1] > scope ){

scope = scope+1.0;
IdScope[count[no]] = scope;
}
*/
}
}
if(track == 1){
```

```
printf("\n\n\n\n");

int countIt = 0;
dimGlobe = dimGlobe+1;
for(int h=0;h<strlen(word);h++){
if(word[h] == '[' ){
printf("Hey");
countIt = countIt+1;
}
}
dimk[dimGlobe] = countIt;
printf("%d---%d\n",dimGlobe, dimk[dimGlobe]);
}
 count[no] = count[no]+1;
 //namethe(no);

 //printf(":\t%s\n", everything[no][count[no]-1]);
}

if(strcmp(word , "int") == 0){
whoIsInSet[set] = 1;
}
else if(strcmp(word , "float") == 0){
whoIsInSet[set] = 2;

}
else if(strcmp(word , "char") == 0){
```

```
whoIsInSet[set] = 3;

}
else if(strcmp(word , "double") == 0){
whoIsInSet[set] = 4;
}
else if(strcmp(word , "void") == 0){

whoIsInSet[set] = 5;

}
else if(strcmp(word , "struct") == 0){

whoIsInSet[set] = 6;

}
else if(strcmp(word , "union") == 0){

whoIsInSet[set] = 7;

}
else if(strcmp(word , "*") == 0){

 if(whoIsInSet[set] == 1){
          whoIsInSet[set] = 8;
 }
```

```
 else if(whoIsInSet[set] == 2){
          whoIsInSet[set] = 9;
}
  else if(whoIsInSet[set] == 3){
          whoIsInSet[set] = 10;
}
  else if(whoIsInSet[set] == 4){
          whoIsInSet[set] = 11;
}
  else if(whoIsInSet[set] == 6){
          whoIsInSet[set] = 12;
}


}


if(track == 1){


  if(whoIsInSet[set] == 1){
          whoIsInSet[set] = 13;
 }
 else if(whoIsInSet[set] == 2){
          whoIsInSet[set] = 14;
}
  else if(whoIsInSet[set] == 3){
          whoIsInSet[set] = 15;
}
  else if(whoIsInSet[set] == 4){
```

```
        whoIsInSet[set] = 16;
}


}

return whoIsInSet[set];

}

int DataType(int no){

switch(no){

case 1 : printf("integer"); return strlen("integer");
        break;
case 2 : printf("float"); return strlen("float");
        break;
case 3 : printf("character"); return strlen("character");
        break;
case 4 : printf("double"); return strlen("double");
        break;
case 5 : printf("void"); return strlen("void");
        break;
case 6 : printf("struct"); return strlen("struct");
        break;
case 7 : printf("union"); return strlen("union");
```

```
        break;
case 8 : printf("integer_pointer"); return strlen("integer_pointer");
        break;
case 9 : printf("float_pointer"); return strlen("float_pointer");
        break;
case 10 : printf("char_pointer"); return strlen("char_pointer");
        break;
case 11 : printf("double_pointer"); return strlen("double_pointer");
        break;
case 12 : printf("struct_pointer"); return strlen("struct_pointer");
        break;
case 13 : printf("integer_array"); return strlen("integer_array");
        break;
case 14 : printf("float_array"); return strlen("float_array");
        break;
case 15 : printf("char_array"); return strlen("char_array");
        break;
case 16 : printf("double_array"); return strlen("double_array");
        break;
default : printf("UNDEFINED_DATA_TYPE"); return
strlen("UNDEFINED_DATA_TYPE");
        break;

}

}
```

```
void theTable(){

        int m,n;
        FILE * fp;
        int h;



  /* open the file for writing*/
        //fp = fopen ("/home/ubuntu/Desktop/correctTestCase/test3/output.txt","w");

        printf("\n\n\n------------------------ Symbol Table ----------------------\n\n\n\n");



                for(h=0;h<20;h++){
        printf("___");
        }



        for(m=1;m<39;m++){

        if(count[m]>0 && (m<30 || m>35) ){



        namethe(m);
        for(n=0;n<count[m];n++)
                {
```

```
        int p = strlen(everything[m][n]);
        if(n==0 && n==count[m]-1){
        printf("  | %s\n|\t\t\t    |\n|\t\t\t    |\n|\t\t\t    |\n",everything[m][n]);

        }
        else if(n==0)
        {printf("  | %s\n",everything[m][n]);}
        else if(n== count[m]-1)
        {printf("|\t\t\t    | %s\n|\t\t\t    |\n|\t\t\t    |\n|\t\t\t    |\n|",everything[m][n]);
        }
        else{
        printf("|\t\t\t    | %s\n",everything[m][n]);
        }

        }
    if(count[m]==1){
    printf("|");
    }
    for(h=0;h<60;h++){
    if (h==27){
        printf("|");}
    else{
    printf("_");
    }
    }

    }
```

```
        }

printf("\n\n\n\n\n\n");

printf("---------------------- Constant Table ----------------------\n\n\n\n");




                for(h=0;h<20;h++){
        printf("___");
        }




                for(m=30;m<36;m++){


        if(count[m]>0){



        namethe(m);
        for(n=0;n<count[m];n++)
                {
                int p = strlen(everything[m][n]);
                if(n==0 && n==count[m]-1){
                printf("  | %s\n|\t\t\t    |\n|\t\t\t    |\n|\t\t\t    |\n",everything[m][n]);
```

```
        }
        else if(n==0)
        {printf("  | %s\n",everything[m][n]);}
        else if(n== count[m]-1)
        {printf("|\t\t\t    | %s\n|\t\t\t    |\n|\t\t\t    |\n|\t\t\t    |\n|",everything[m][n]);
        }
        else{
        printf("|\t\t\t    | %s\n",everything[m][n]);
        }

        }
if(count[m]==1){
printf("|");
}
for(h=0;h<60;h++){
if (h==27){
        printf("|");}
else{
printf("_");
}
}

}



}
```

```
printf("\n\n\n\n");

printf("------------------------------ ID  ----------------------------\n\n\n\n");

                    for(h=0;h<77;h++){
          printf("_");
          }

m = 34;

int dimTrack = 0;

printf("\n|\t\t\t    |\t\t\t    |\t\t\t    |\n|       VALUE             |        SCOPE         |       DATA
TYPE       |");
printf("\n|\t\t\t    |\t\t\t    |\t\t\t    |\n");
printf("|");

                    for(h=0;h<76;h++){
                              if(h==27 || h==51 || h==75)
                              {printf("|");}
                              else{
```

```
                                printf("_");}
        }
for(n=0;n<count[m];n++){

        if(count[m]>0){



printf("\n|\t\t\t    |\t\t\t    |\t\t\t    |\n|");
int slen = strlen(everything[m][n]);
printf("%s",everything[m][n]);

int j;
for(j=0;j<27-slen;j++){
printf(" ");
}

printf("|");

char buf[5];
//gcvt (IdScope[n], 4,buf);
slen = 7;
printf("%f",IdScope[n]);

for(j=0;j<23-slen;j++){
printf(" ");
}
printf("|");
```

```
/*
if(Dim[n]!=-1){
printf("%dD-",Dim[n]);
}
slen = 3;

*/

slen =0;

if( whoIsInSet[lineNo[n]] >= 13 && whoIsInSet[lineNo[n]] <= 16){

printf("%dD-",dimk[dimTrack]);
dimTrack = dimTrack +1;
slen =  3;

}

slen = slen + DataType(whoIsInSet[lineNo[n]]);



for(j=0;j<23-slen;j++){
printf(" ");
}
printf("|");
```

```
printf("\n|\t\t\t    |\t\t\t    |\t\t\t    |\n");
printf("|");

                    for(h=0;h<76;h++){
                              if(h==27 || h==51 || h==75)
                              {printf("|");}
                              else{
                                        printf("_");}
          }

          }

          }

printf("\n\n\n\n");

}
```

```
// 1 Real Numbers
//2 String Literal
// 3 Separator
//4 Header Files
// 5 printf
// 6 keyWord
// 7 identifier
// 8 operator
// 9 format specifier
//     struct node{

//         char* text;
//         int type;
//         struct node* link;
//         struct node* down;

//     };



//     struct node* root = (struct node*) malloc(sizeof(struct node));
//     root->link = NULL;
//     root->down = NULL;



// void the_medium_to_display(char* the_text, int category, struct node* head){

//         struct node* temp = (struct node*) malloc(sizeof(struct node));
```

```
//	temp->text = the_text;
//	temp->type = category;
//	temp->link=NULL;
//	temp->down=NULL;

// if(head->down == NULL){


//	head->down = temp;

// }

// else{

//	struct node* tmp;
//	tmp = head;
//	int note=0;

//	while(tmp->down!=NULL){

//		tmp = tmp->down;

//		if(tmp->type==category){
//			note=1;

//			struct node* tempr;
```

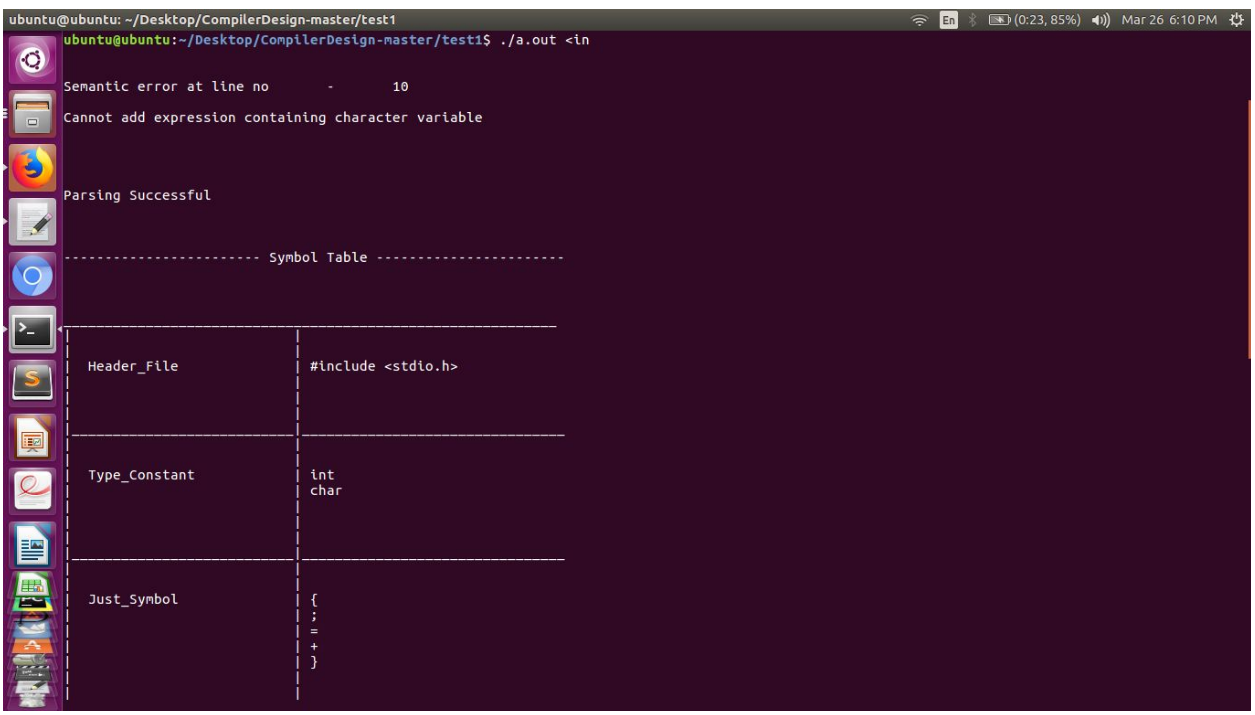
```
//                      tempr = tmp;

//                      while(tempr->link!=NULL){
//                              tempr= tempr->link;
//                      }

//                      tempr->link = temp;

//              }

//      }

//      if(note==0){
//              tmp->down = temp;
//      }

// }

// printf("%d : %s\n", category, the_text);


// }
```

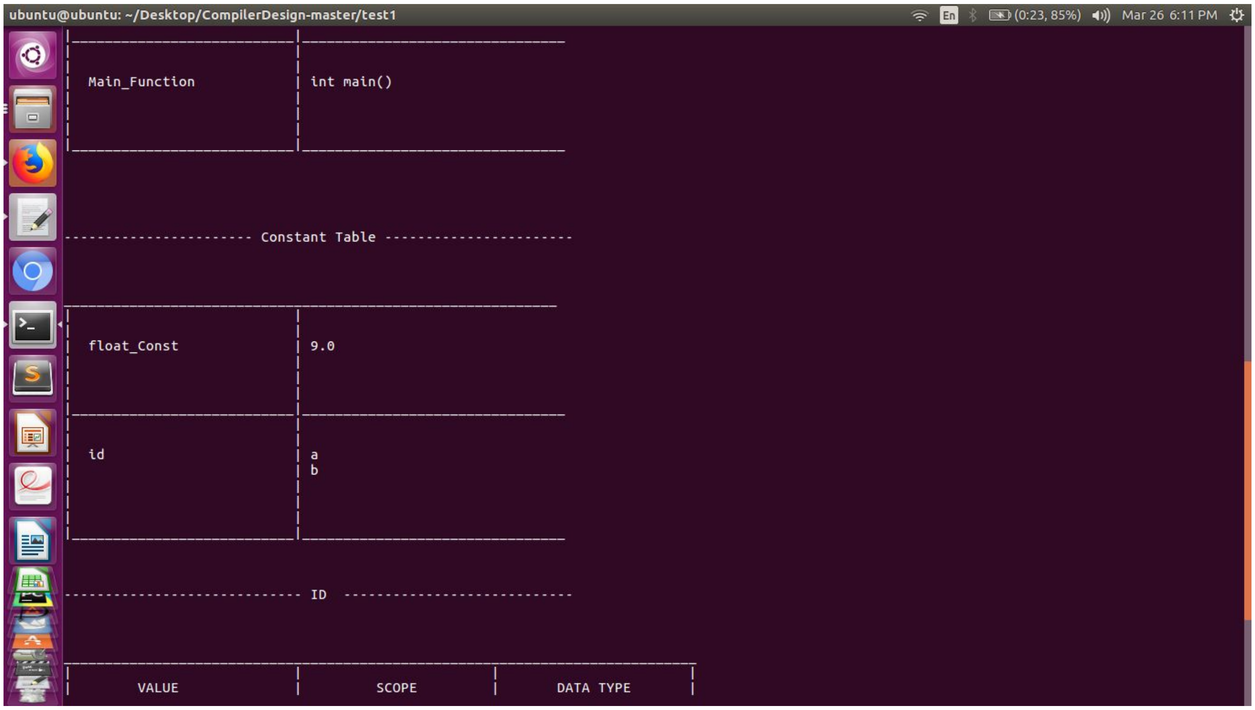
Input Test Cases:

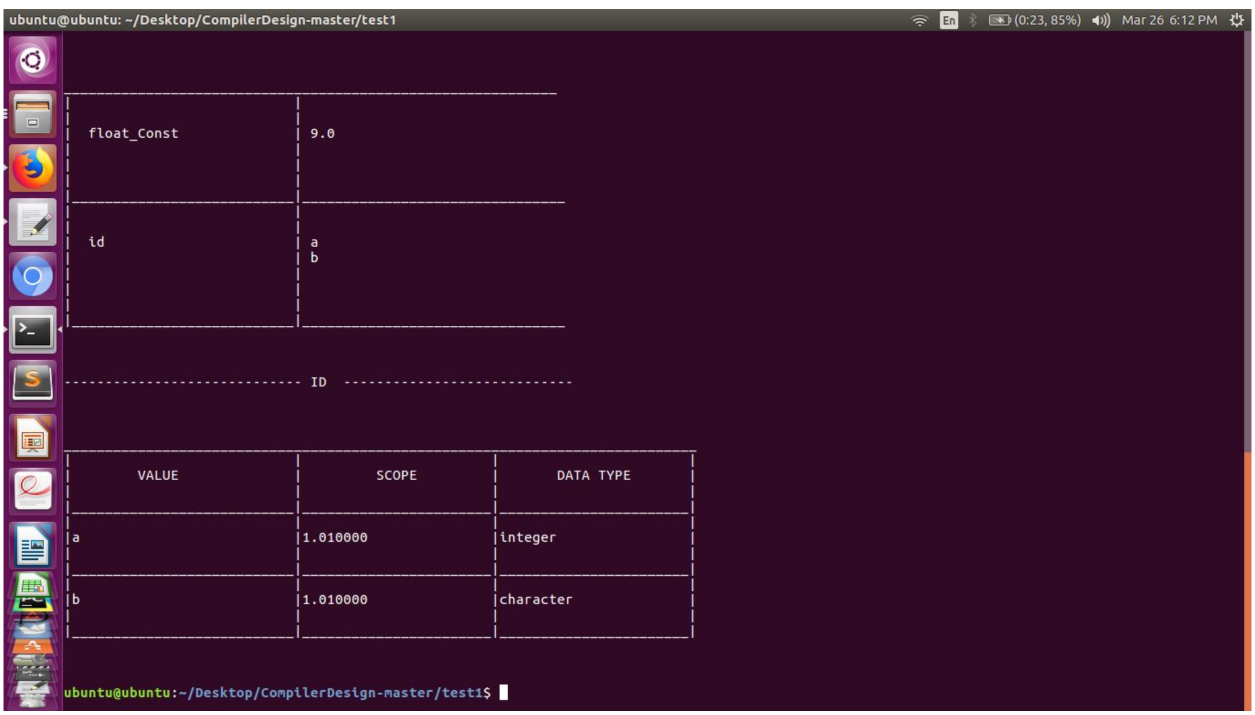
Test1:

```
#include <stdio.h>

int main(){

int a;
char b;

a= 9.0;

a = b + a;

}
```

Output:

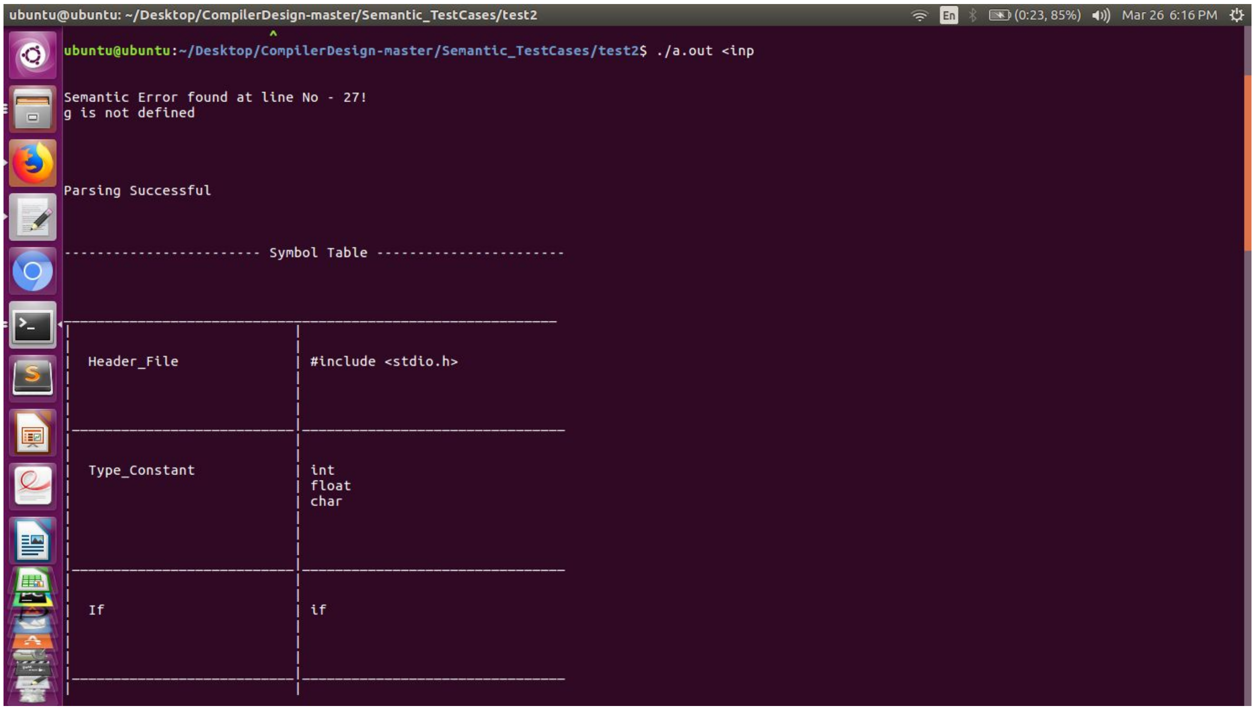



ubuntu@ubuntu: ~/Desktop/CompilerDesign-master/test1
(0:23, 85%)
Mar 26 6:11 PM
Main_Function
int main()
Constant Table
float_Const
9.0
id
a
b
ID
VALUE
SCOPE
DATA TYPE

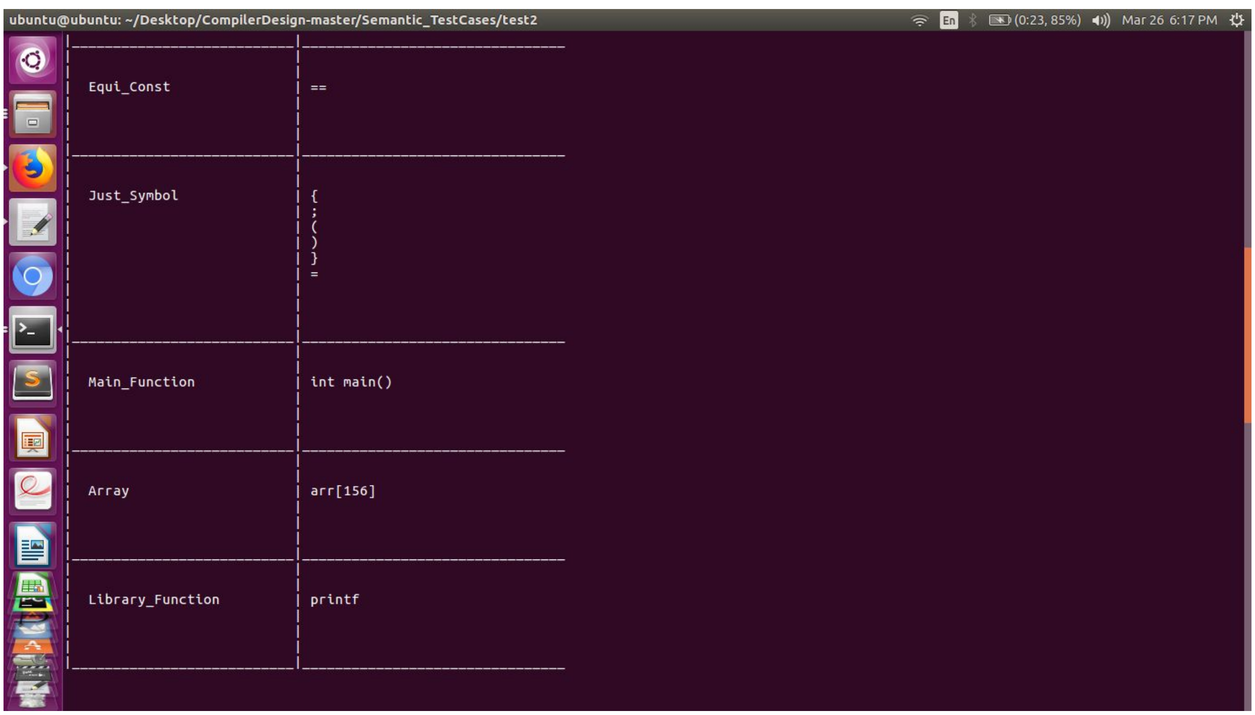

ubuntu@ubuntu: ~/Desktop/CompilerDesign-master/test1
(0:23, 85%)
Mar 26 6:12 PM
float_Const
9.0
id
a
b
ID
VALUE
SCOPE
DATA TYPE
a
1.010000
integer
b
1.010000
character
ubuntu@ubuntu:~/Desktop/CompilerDesign-master/test1$

Test 2:

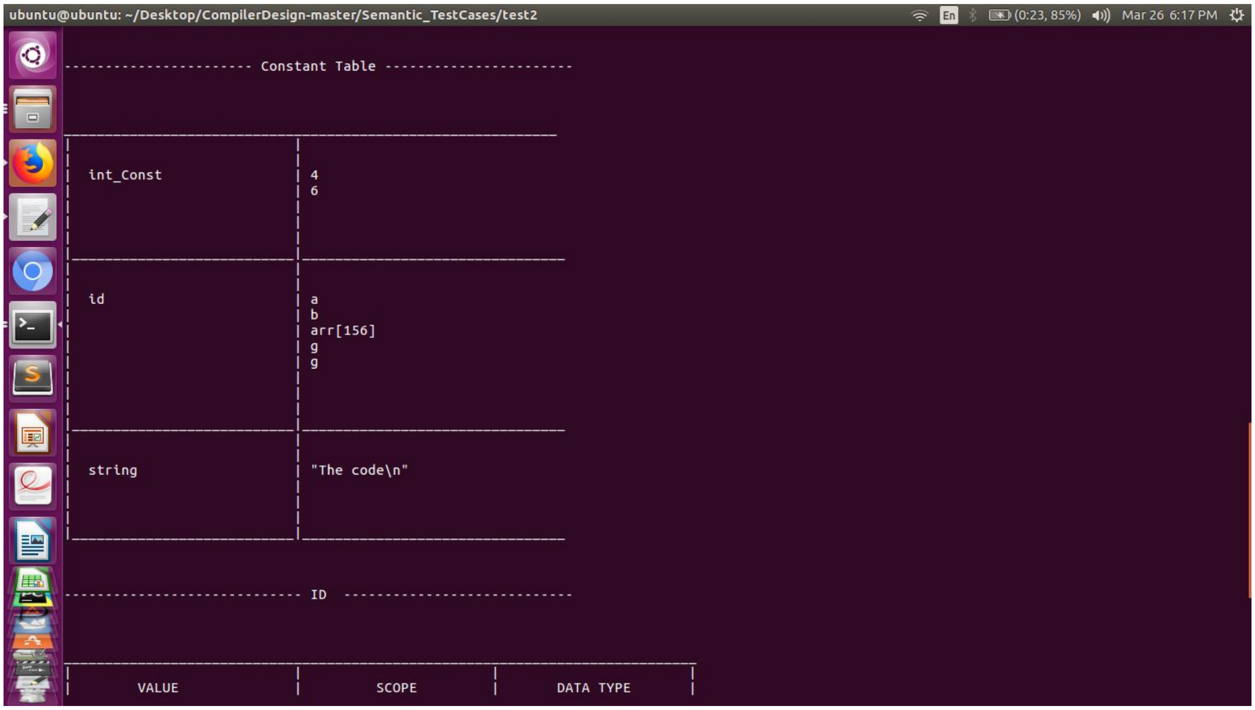
```
#include <stdio.h>
/*
This is a comment

This a semantically incorrect Code
*/
// undefined Variable
//Array structure

int main(){

int a;
float b;
char arr[156];
printf("The code\n");
if(a==4){

int g;

}
if(g==6){

 a=6;
}

}
```

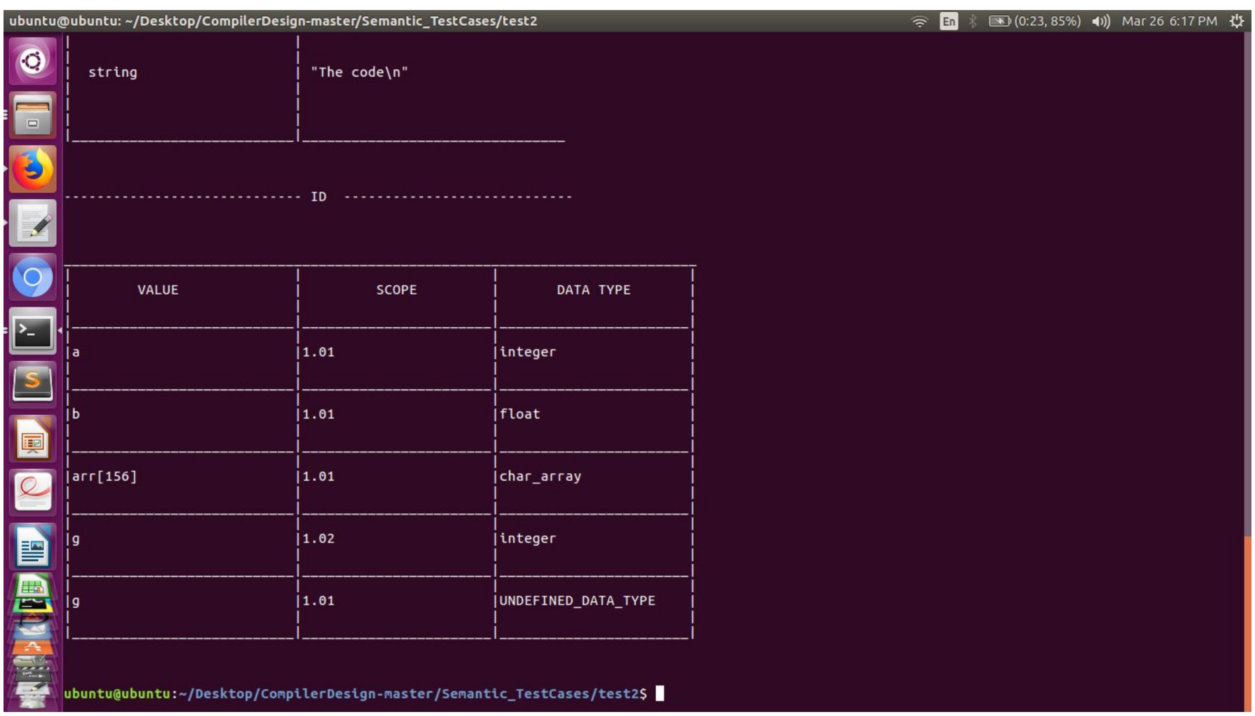


ubuntu@ubuntu: ~/Desktop/CompilerDesign-master/Semantic_TestCases/test2
(0:23, 85%)
Mar 26 6:16 PM
ubuntu@ubuntu:~/Desktop/CompilerDesign-master/Semantic_TestCases/test2$ ./a.out <inp
Semantic Error found at line No - 27!
g is not defined
Parsing Successful
---------------------- Symbol Table ----------------------
Header_File
#include <stdio.h>
Type_Constant
int
float
char
If
if



ubuntu@ubuntu: ~/Desktop/CompilerDesign-master/Semantic_TestCases/test2
(0:23, 85%)
Mar 26 6:17 PM
Equi_Const
==
Just_Symbol
{
;
(
)
}
=
Main_Function
int main()
Array
arr[156]
Library_Function
printf

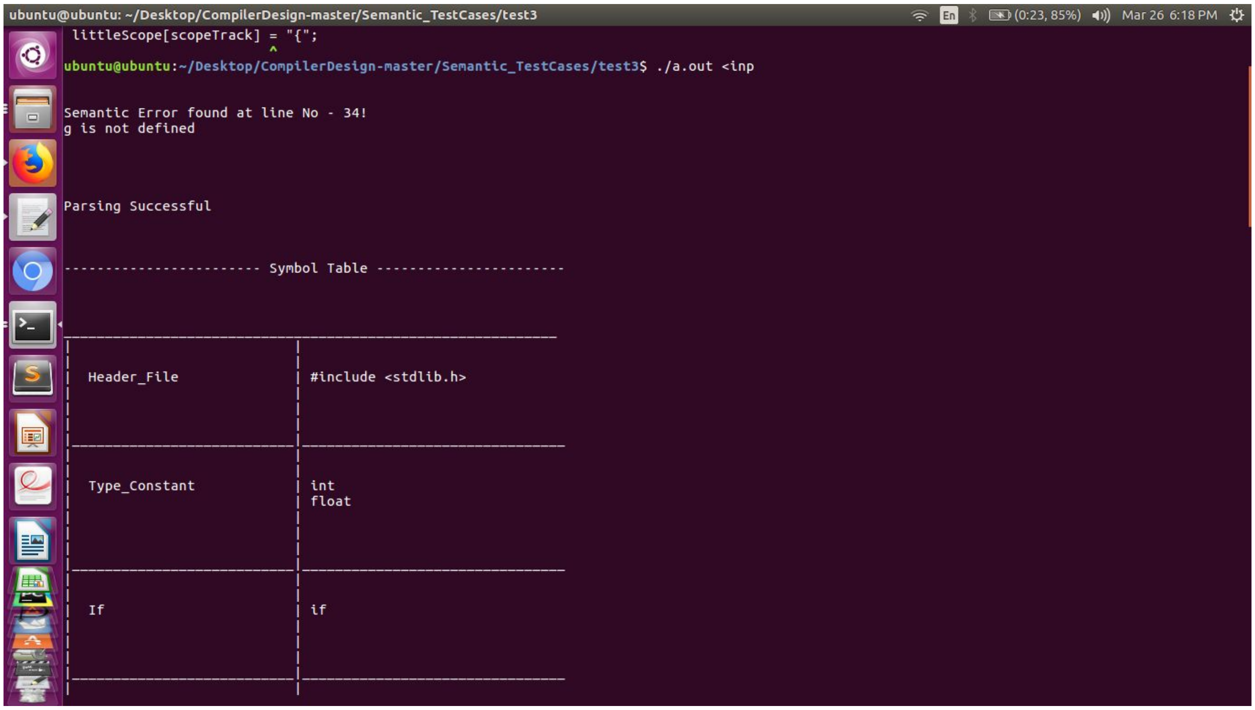

ubuntu@ubuntu: ~/Desktop/CompilerDesign-master/Semantic_TestCases/test2
En
(0:23, 85%)
Mar 26 6:17 PM
---------------------- Constant Table ----------------------
int_Const
4
6
id
a
b
arr[156]
g
g
string
"The code\n"
--------------------------- ID ---------------------------
VALUE
SCOPE
DATA TYPE

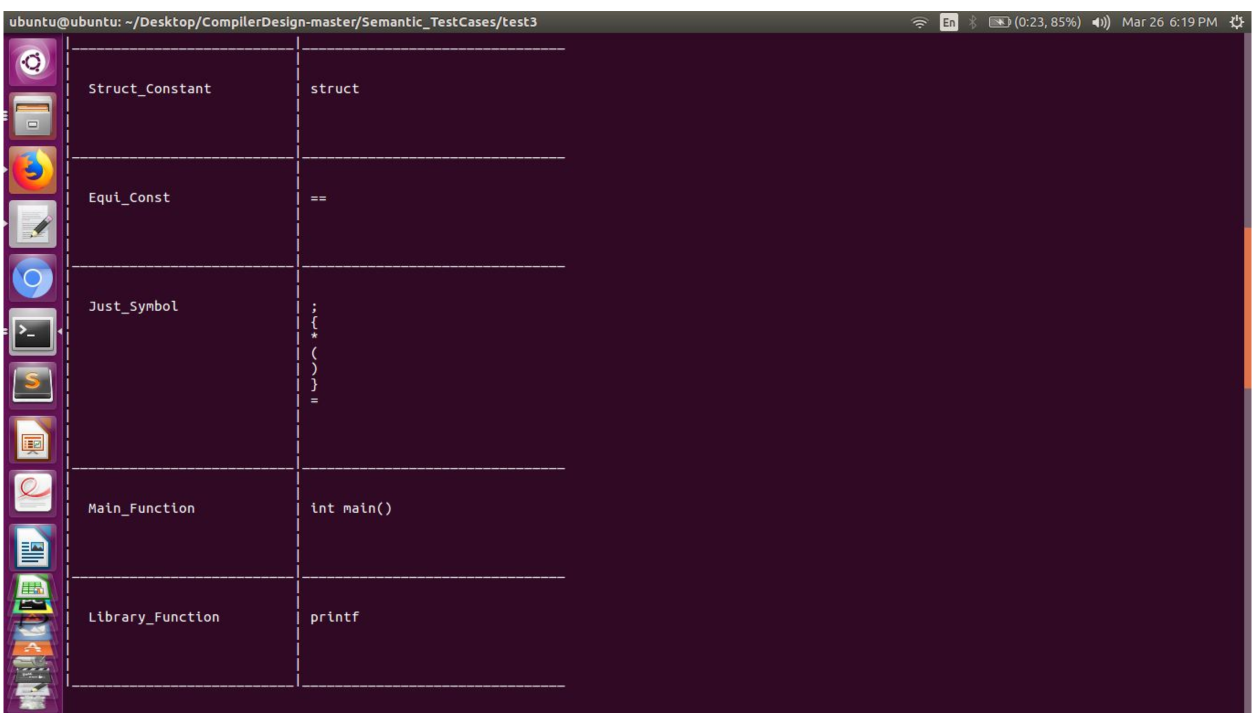

ubuntu@ubuntu: ~/Desktop/CompilerDesign-master/Semantic_TestCases/test2
En
(0:23, 85%)
Mar 26 6:17 PM
string
"The code\n"
--------------------------- ID ---------------------------
VALUE
SCOPE
DATA TYPE
a
1.01
integer
b
1.01
float
arr[156]
1.01
char_array
g
1.02
integer
g
1.01
UNDEFINED_DATA_TYPE
ubuntu@ubuntu:~/Desktop/CompilerDesign-master/Semantic_TestCases/test2$

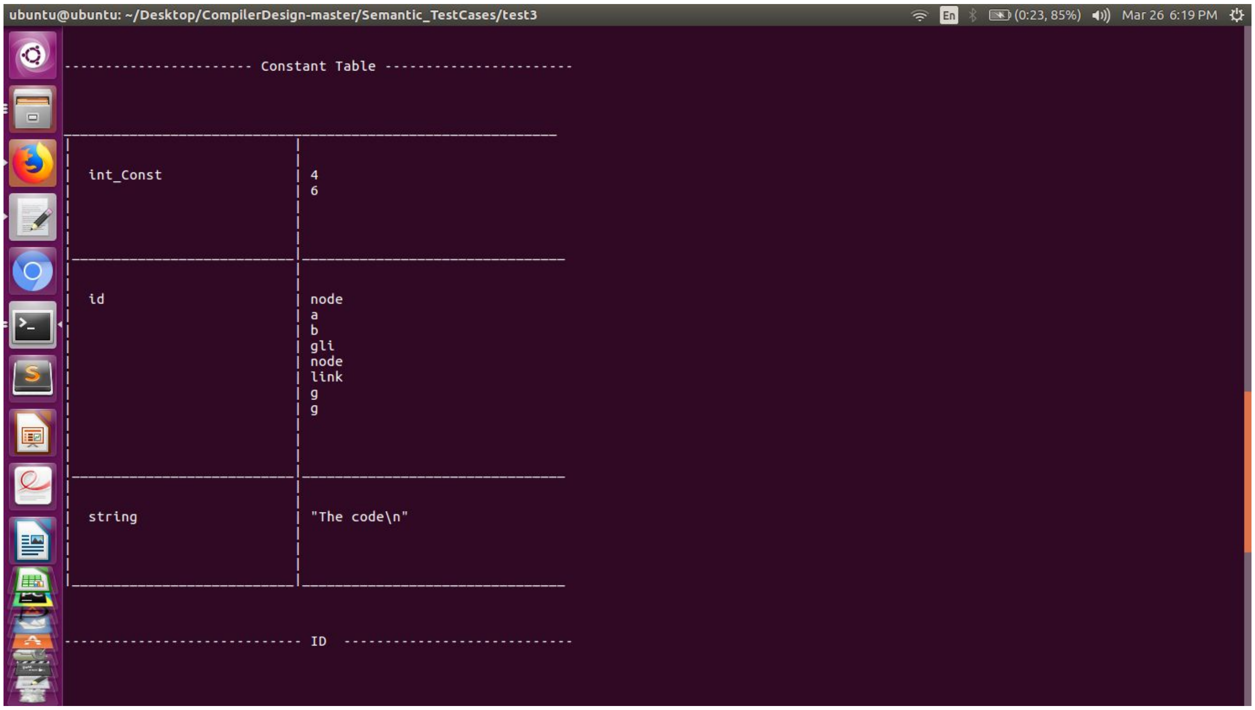
Test 3:

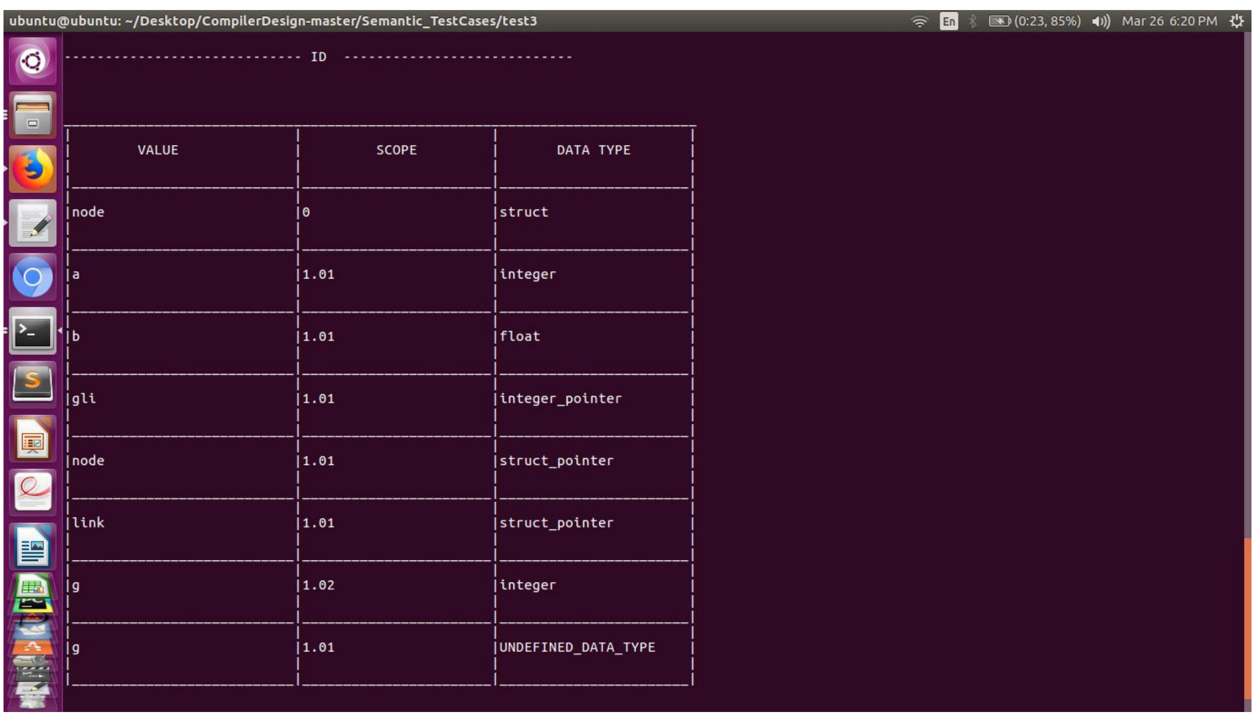
```
#include <stdlib.h>

/*
This is a comment
*/

//Struct pointers
// g is not accepted, because it's outside Scope
//pointers

struct node;
int main(){

int a;
float b;

int * gli;
struct node* link;

printf("The code\n");
if(a==4){
int g;

}
g=6;
}
```

Output:


ubuntu@ubuntu: ~/Desktop/CompilerDesign-master/Semantic_TestCases/test3
Mar 26 6:18 PM
littleScope[scopeTrack] = "{";
ubuntu@ubuntu:~/Desktop/CompilerDesign-master/Semantic_TestCases/test3$ ./a.out <inp
Semantic Error found at line No - 34!
g is not defined
Parsing Successful
---------------------- Symbol Table ----------------------
Header_File
#include <stdlib.h>
Type_Constant
int
float
If
if


ubuntu@ubuntu: ~/Desktop/CompilerDesign-master/Semantic_TestCases/test3
Mar 26 6:19 PM
Struct_Constant
struct
Equi_Const
==
Just_Symbol
;
{
*
(
)
}
=
Main_Function
int main()
Library_Function
printf

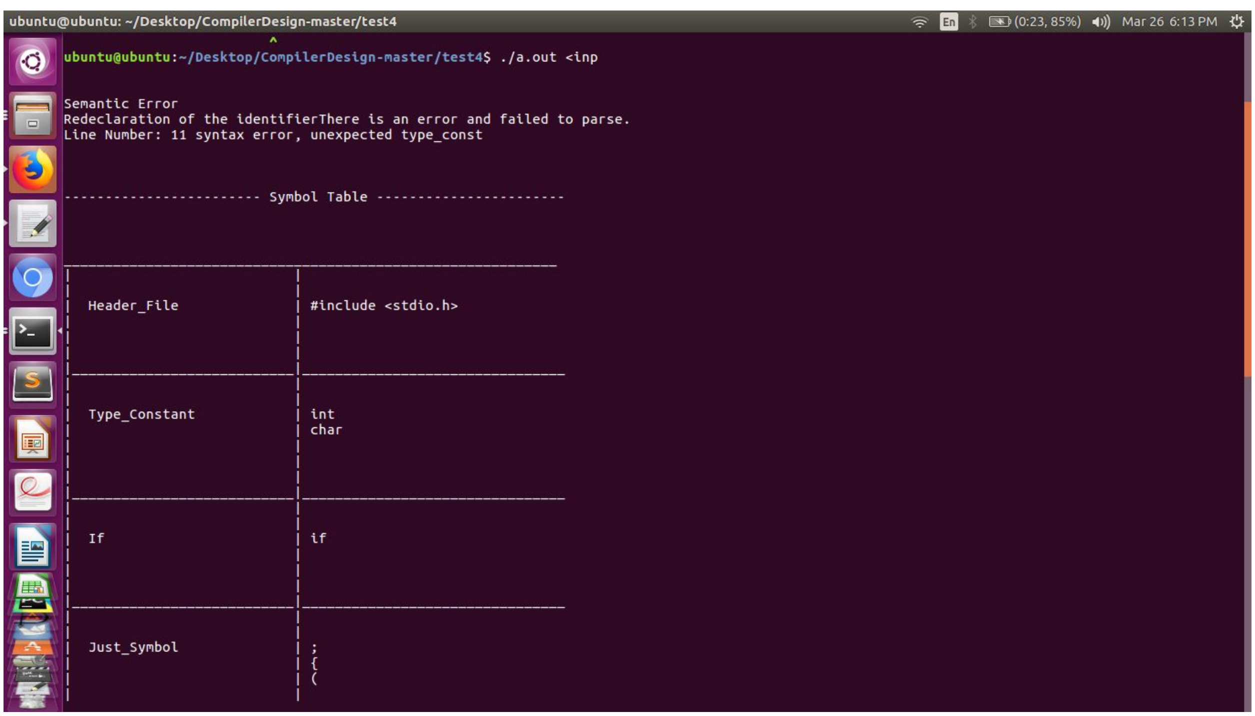

ubuntu@ubuntu: ~/Desktop/CompilerDesign-master/Semantic_TestCases/test3
Mar 26 6:19 PM
---------------------- Constant Table ----------------------
int_Const
4
6
id
node
a
b
gli
node
link
g
g
string
"The code\n"
--------------------------- ID ---------------------------

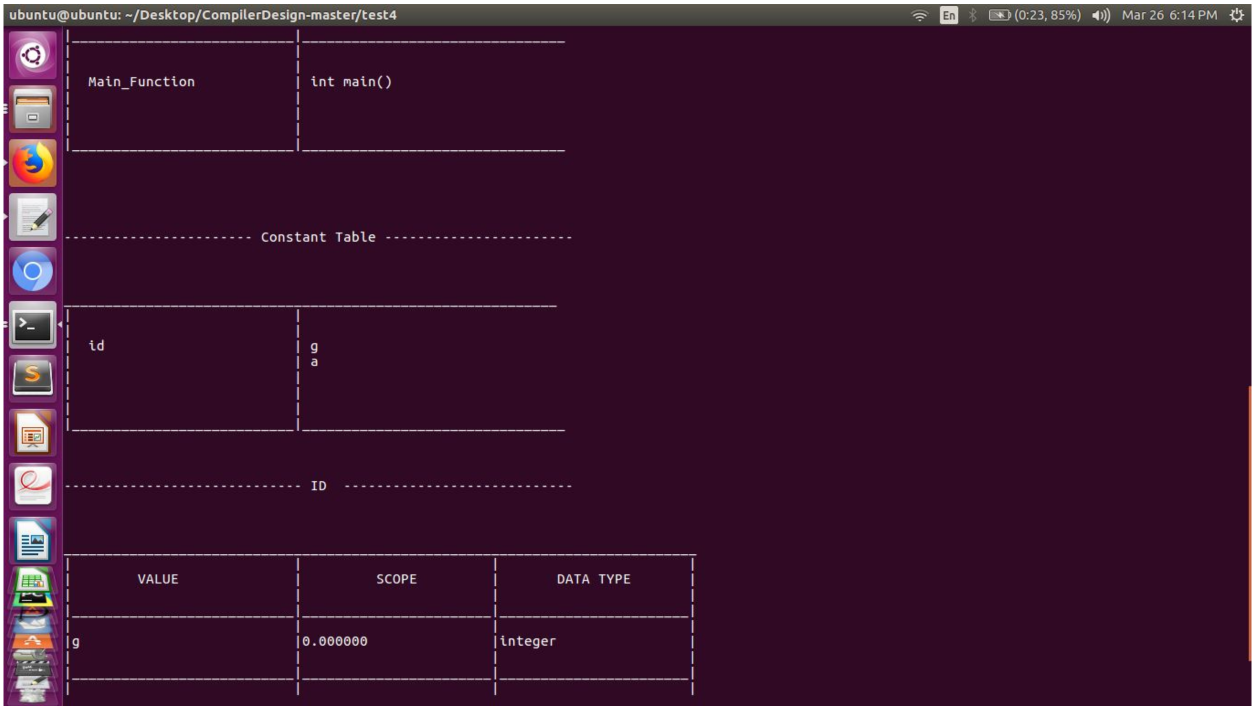

ubuntu@ubuntu: ~/Desktop/CompilerDesign-master/Semantic_TestCases/test3
Mar 26 6:20 PM
---------------------------- ID ---------------------------
VALUE
SCOPE
DATA TYPE
node
0
struct
a
1.01
integer
b
1.01
float
gli
1.01
integer_pointer
node
1.01
struct_pointer
link
1.01
struct_pointer
g
1.02
integer
g
1.01
UNDEFINED_DATA_TYPE

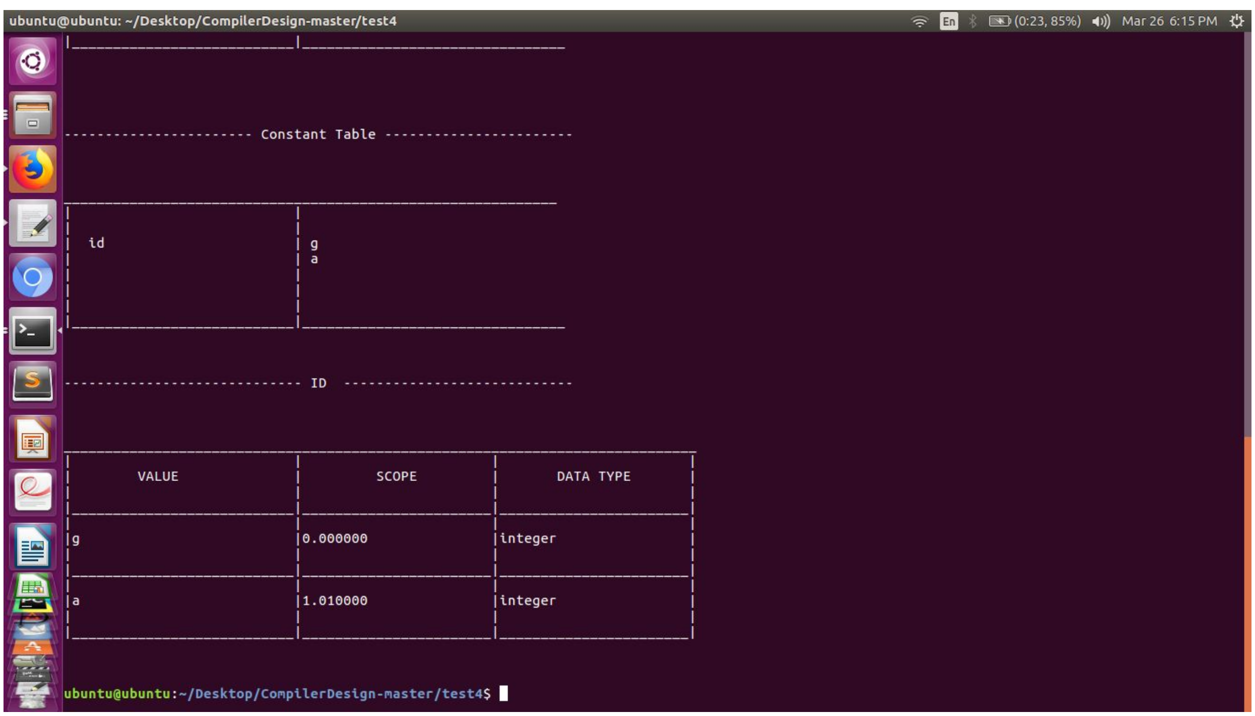
Test 4:

```
#include <stdio.h>


int g;
int main(){
int a;
char a;


if(int a){
}
}
```

Output:



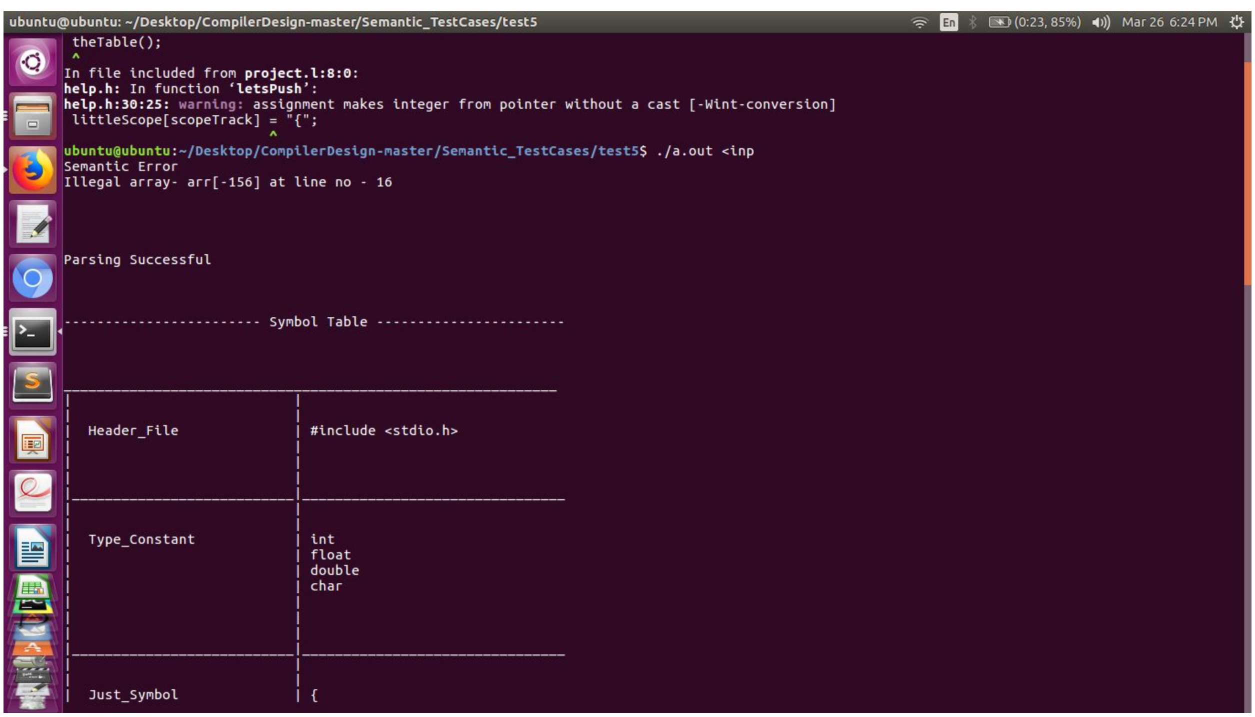

ubuntu@ubuntu: ~/Desktop/CompilerDesign-master/test4
Mar 26 6:14 PM
Main_Function
int main()
Constant Table
id
g
a
ID
VALUE
SCOPE
DATA TYPE
g
0.000000
integer

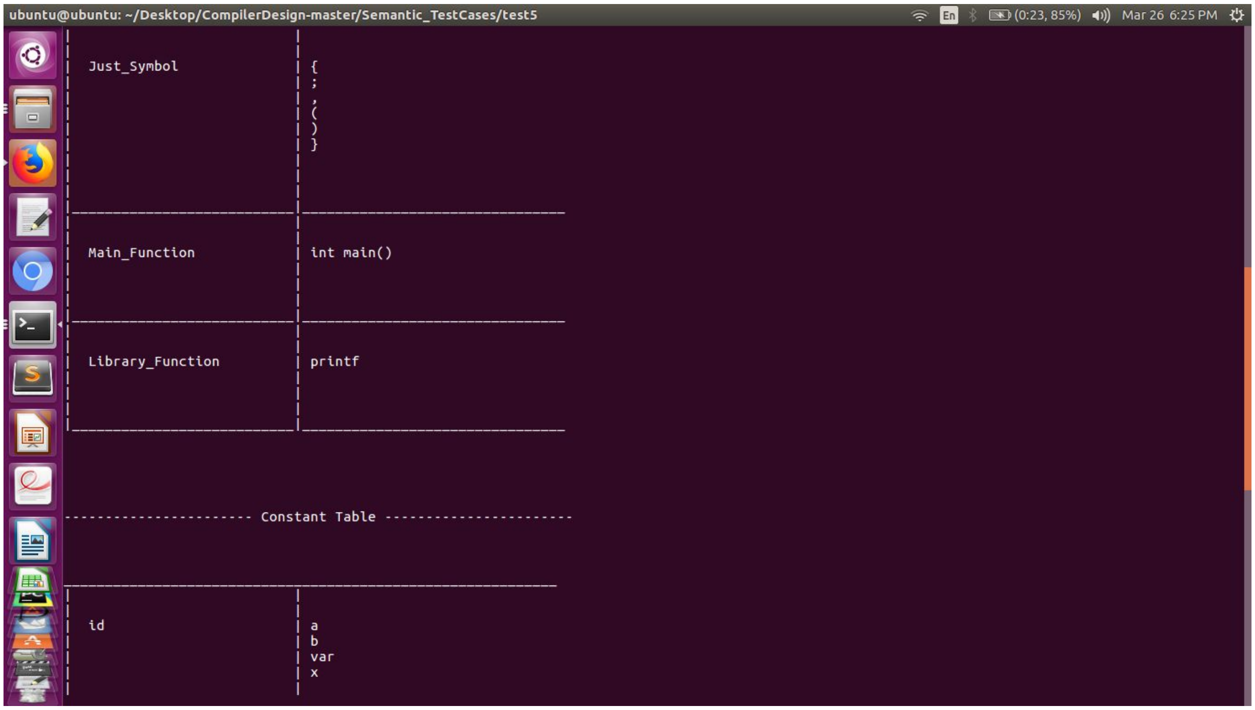

ubuntu@ubuntu: ~/Desktop/CompilerDesign-master/test4
Mar 26 6:15 PM
Constant Table
id
g
a
ID
VALUE
SCOPE
DATA TYPE
g
0.000000
integer
a
1.010000
integer
ubuntu@ubuntu:~/Desktop/CompilerDesign-master/test4$

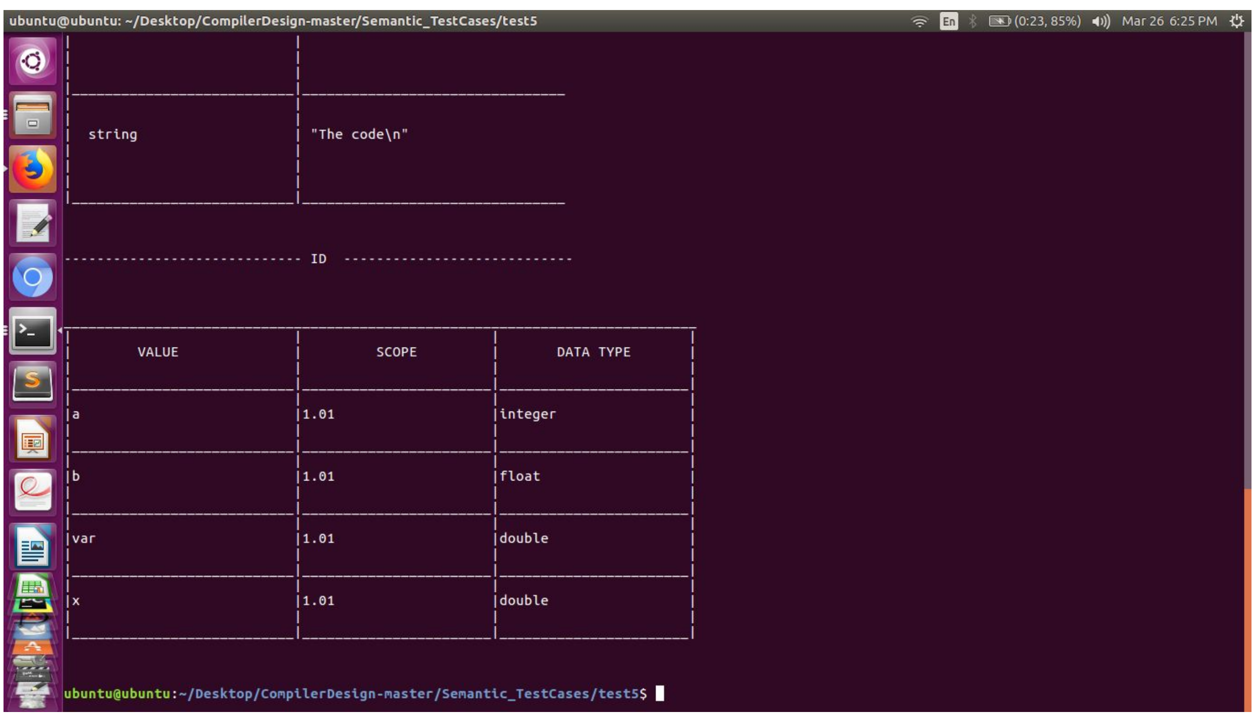
Test 5:

```
#include <stdio.h>
/*
This is a comment
This a semantically correct Code
*/

int main(){
int a;
float b;
double var, x;
char arr[-156];
printf("The code\n");

}
```

Output:



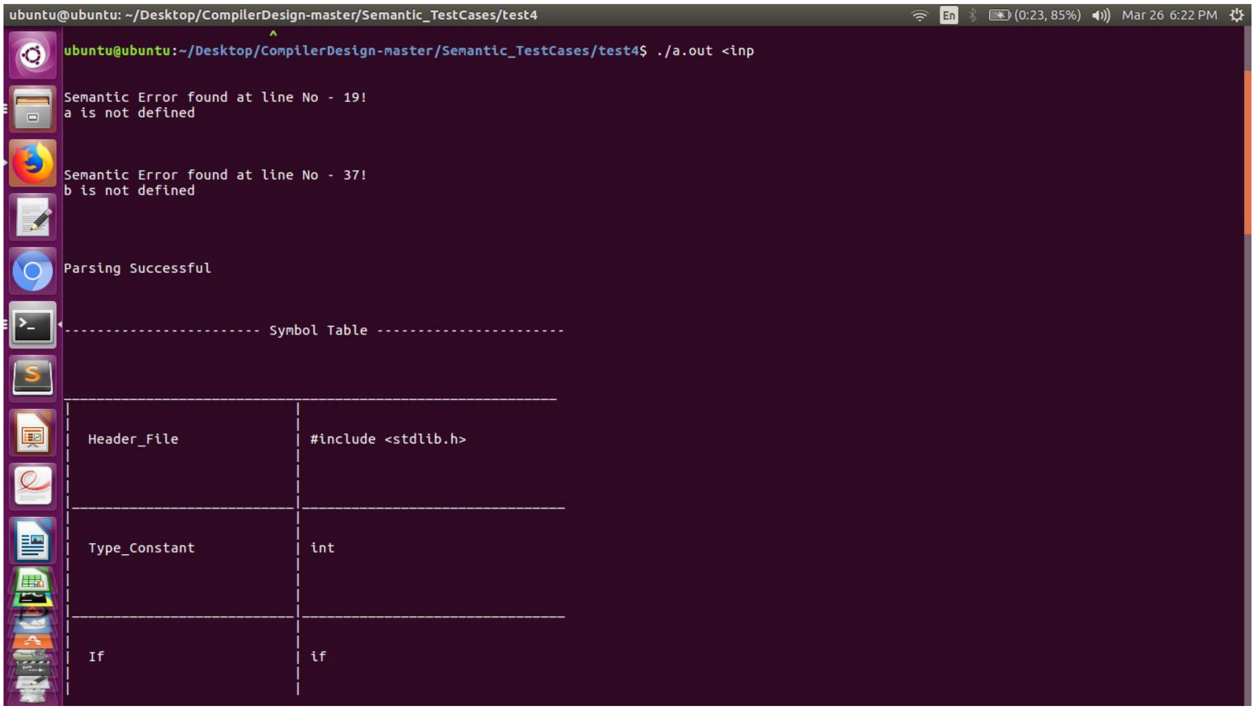

ubuntu@ubuntu: ~/Desktop/CompilerDesign-master/Semantic_TestCases/test5
Mar 26 6:25 PM
Just_Symbol
{
;
,
(
)
}
Main_Function
int main()
Library_Function
printf
Constant Table
id
a
b
var
x

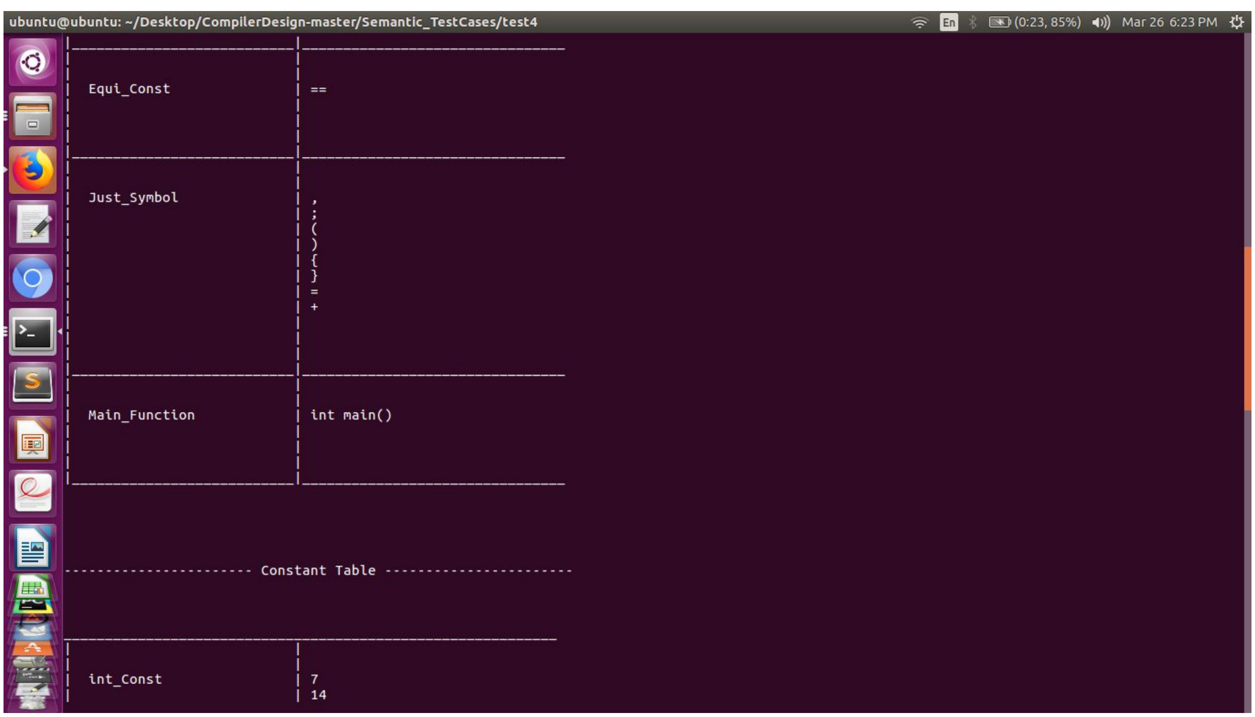

ubuntu@ubuntu: ~/Desktop/CompilerDesign-master/Semantic_TestCases/test5
Mar 26 6:25 PM
string
"The code\n"
ID
VALUE
SCOPE
DATA TYPE
a
1.01
integer
b
1.01
float
var
1.01
double
x
1.01
double
ubuntu@ubuntu:~/Desktop/CompilerDesign-master/Semantic_TestCases/test5$

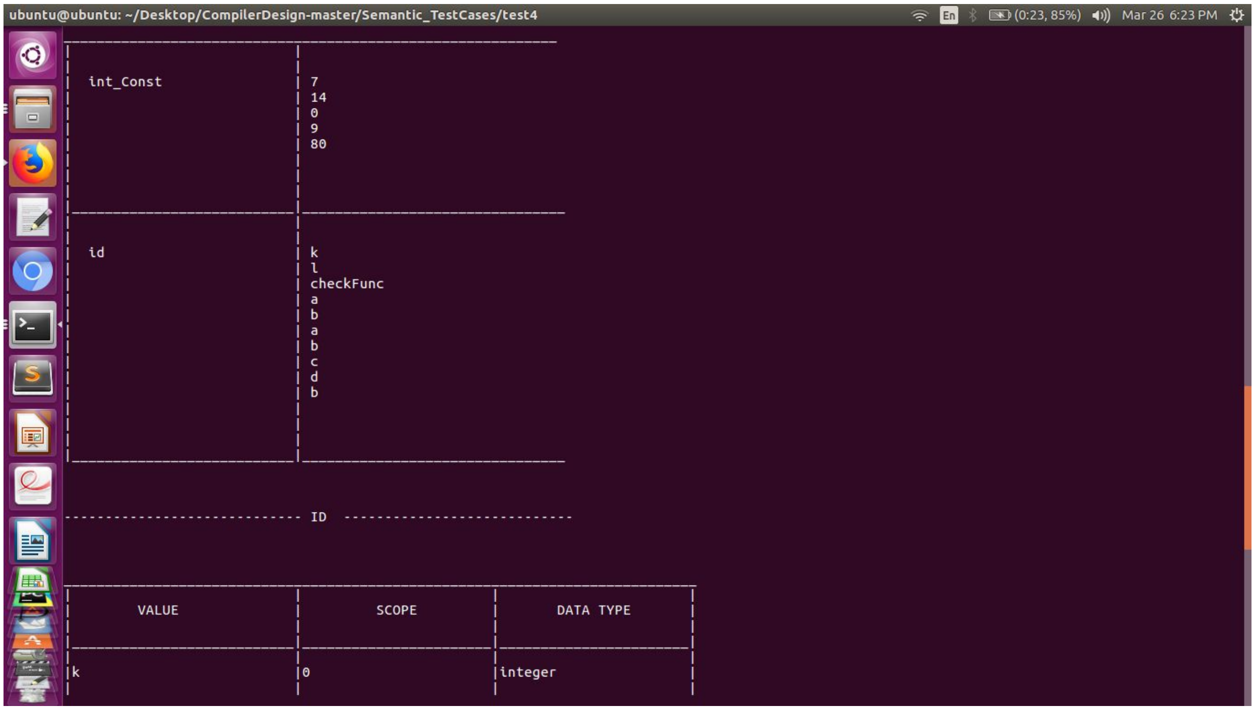
Test 6:

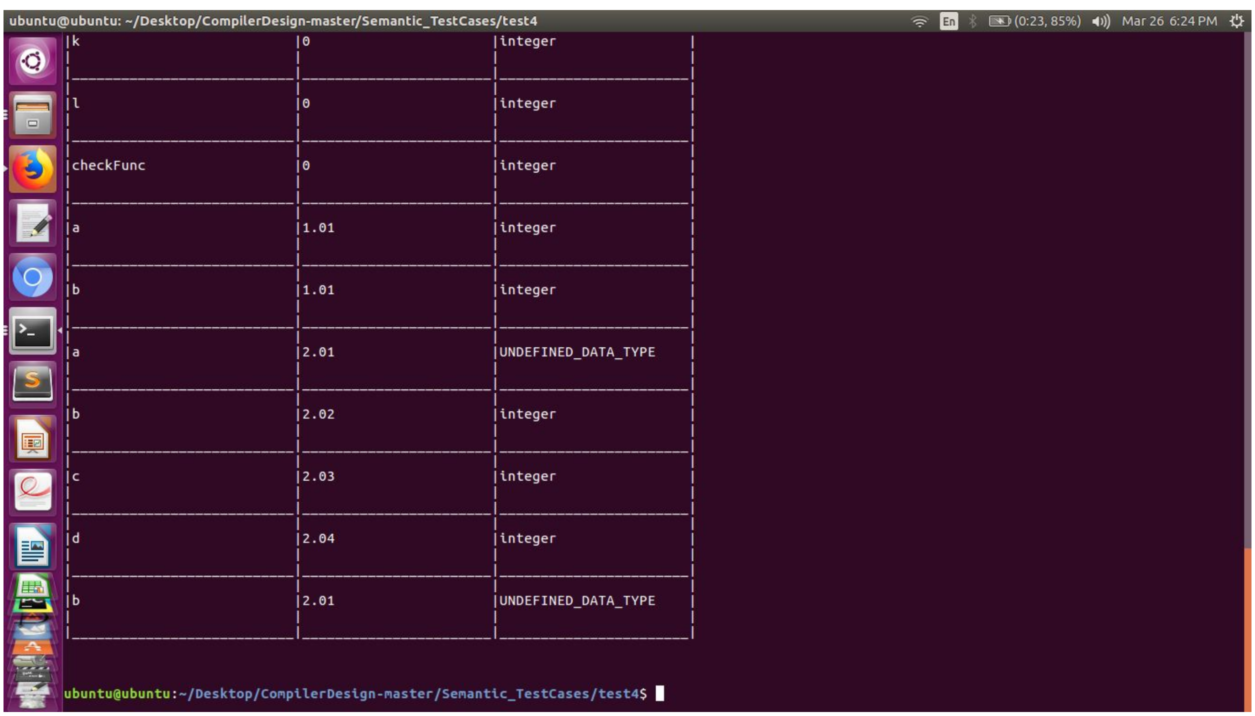
```
#include <stdlib.h>
//Global Varible Acceptance
// Other function defined varibale rejection
//hierarchy of variables
int k,l;
int checkFunc(){

int a, b;

}

int main(){

a = 7+7;

if(a==14){
int b;
if(b==0){
int c;
if(c==0){
int d;
}
}
}
b = 9;
l= 80;

}
```

Output:


ubuntu@ubuntu: ~/Desktop/CompilerDesign-master/Semantic_TestCases/test4
Mar 26 6:22 PM
ubuntu@ubuntu:~/Desktop/CompilerDesign-master/Semantic_TestCases/test4$ ./a.out <inp
Semantic Error found at line No - 19!
a is not defined
Semantic Error found at line No - 37!
b is not defined
Parsing Successful
----------------------- Symbol Table ----------------------
Header_File
#include <stdlib.h>
Type_Constant
int
If
if


ubuntu@ubuntu: ~/Desktop/CompilerDesign-master/Semantic_TestCases/test4
Mar 26 6:23 PM
Equi_Const
==
Just_Symbol
,
;
(
)
{
}
=
+
Main_Function
int main()
---------------------- Constant Table ----------------------
int_Const
7
14


ubuntu@ubuntu: ~/Desktop/CompilerDesign-master/Semantic_TestCases/test4
Mar 26 6:23 PM
int_Const
7
14
0
9
80
id
k
l
checkFunc
a
b
a
b
c
d
b
ID
VALUE
SCOPE
DATA TYPE
k
0
integer


ubuntu@ubuntu: ~/Desktop/CompilerDesign-master/Semantic_TestCases/test4
Mar 26 6:24 PM
k
0
integer
l
0
integer
checkFunc
0
integer
a
1.01
integer
b
1.01
integer
a
2.01
UNDEFINED_DATA_TYPE
b
2.02
integer
c
2.03
integer
d
2.04
integer
b
2.01
UNDEFINED_DATA_TYPE
ubuntu@ubuntu:~/Desktop/CompilerDesign-master/Semantic_TestCases/test4$

Conclusion:

Compilers use semantic analysis to enforce the static semantic rules of a language .It is hard to generalize the exact boundaries between semantic analysis and the generation of intermediate representations (or even just straight to final represenations); this demarcation is the logical boundary between the front-end of a compiler (lexical analysis and parsing) and the back-end of the compiler (intermediate representations and final code.)For instance, a completely separated compiler could have a well-defined lexical analysis and parsing stage generating a parse tree, which is passed wholesale to a semantic analyzer, which could then create a syntax tree and populate a symbol table, and then pass it all on to a code generator; Or a completely interleaved compiler could intermix all of these stages, literally generating final code as part of the parsing engine.The text focuses on an organization where the parser creates a syntax tree (and no full parse tree), and semantic analysis is done over a separate traversal of the syntax tree.