NATIONAL INSTITUTE OF TECHNOLOGY, TIRUCHIRAPPALLI

CSPC62

COMPILER DESIGN

TOPIC : Compiler for Base Typescript

LAB REPORT - 3

<u>Sub Topic: Semantic Analyser</u>

DONE BY

S.No	Name	Roll No.
1.	Dhrubit Hajong	106121037
2.	Mercia Melvin Perinchery	106121077
3.	Nishith Eedula	106121085

Index

Phase 3: Semantic Analyser

I. Creating the Symbol Table

A **Symbol Table** is a data structure created and maintained by the compiler in order to keep track of the semantics of variables i.e. it stores information about the scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc. In the semantic analysis phase, it is used to verify that expressions and assignments are semantically correct (perform type checking) and update them accordingly.

A symbol table contains the following information for each and every symbol.

1. **Identifier Name**
   Specifies the actual name of the symbol used in the program
2. **Type Information**
   Specifies the type of data that the identifier holds along with the scope of declaration of the identifier
3. **Line Information**
   Specifies the line in the program where the identifier was used which can be very helpful when trying to debug errors in the program

Symbol Table Definitions

This section consists of four parts.

a. Structure of Symbol Table

The outline for each entry in the symbol table is given by a user-defined datatype. This data type contains all the necessary information such as the identifier name, the data type of the identifier, the category of the identifier and the line number corresponding to the identifier in the program.

```c
struct dataType {
    char *id_name;
    char *data_type;
    char *type;
    int   line_no;
} symbol_table[100];
```

b. Insertion of Types

The compiler needs a method that can recognise the data type of an identifier that is newly added to the symbol table. This method copies the data type of the current symbol (given by the LEX pointer `yytext`) into the symbol table using the `strcpy()` function built into C.

```
void insert_type() {
      strcpy(type, yytext);
}
```

c. Population of Symbol Table Entries

The compiler needs another method that inserts the data into the symbol table based on the type of symbol. The types of symbols are as follows.

1. **Headers**
   Used for all modular import statements
2. **Keywords**
   Used for keywords that are reserved by the compiler
3. **Variables**
   Used for regular variables that are declared
4. **Constants**
   Used for static values that during assignment of variables
5. **Function**
   Used for user-defined methods

This method also maps each entry to an index in the data structure and maintains the size of the symbol table.

```
void add(char c) {
q = search(yytext);
    if(!q) {
        if(c == 'H') {
            symbol_table[count].id_name = strdup(yytext);
            symbol_table[count].data_type = strdup(type);
            symbol_table[count].line_no = countn;
            symbol_table[count].type=strdup("Header");
            count++;
        }
        else if(c == 'K') {
            symbol_table[count].id_name=strdup(yytext);
            symbol_table[count].data_type=strdup("N/A");
            symbol_table[count].line_no=countn;
            symbol_table[count].type=strdup("Keyword\t");
            count++;
        }
        else if(c == 'V') {
            symbol_table[count].id_name=strdup(yytext);
            symbol_table[count].data_type=strdup(type);
            symbol_table[count].line_no=countn;
            symbol_table[count].type = strdup("Variable");
            count++;
        }
        else if(c == 'C') {
            symbol_table[count].id_name = strdup(yytext);
            symbol_table[count].data_type = strdup("CONST");
            symbol_table[count].line_no = countn;
            symbol_table[count].type = strdup("Constant");
            count++;
        }
        else if(c == 'F') {
            symbol_table[count].id_name = strdup(yytext);
            symbol_table[count].data_type = strdup(type);
            symbol_table[count].line_no = countn;
            symbol_table[count].type = strdup("Function");
            count++;
        }
    }
}
```

d. Check for Duplicates

The compiler needs a method that ensures the program does not have multiple instances of the same symbol in the symbol table.

```
int search(char *type) {
      int i;
      for(i = count - 1; i >= 0; i--) {
            if(strcmp(symbol_table[i].id_name, type) == 0) {
                  return -1;
                  break;
            }
      }
      return 0;
}
```

II. Creating the Semantic Analyser

A **semantic analyser** is the component of the compiler that ensures all the declarations and statements in the program are semantically correct and consistent with the language definition.

The syntax analyser checks if the structure of each part of the grammar matches that of the defined language, however it is not able to detect inconsistencies in the logic and meaning of the declared statements. Thus, we use a semantic analyser that retrieves information from both the syntax tree and the symbol table to verify the consistency of the program.

Types of Semantic Errors

This section describes the four types of semantic errors that can be detected by the Base Typescript Compiler.

a. No Variable Declaration

The semantic analyser needs to check if the variable being used has been declared before being manipulated, in other words the identifier needs to be present in the symbol table before any operations are performed on it. The following function is called for every instance an identifier is used apart from in its declaration.

```
int check_declaration(char *c) {
    q = search(c);
    if(!q) {
        sprintf(errors[sem_errors], "Line %d: Variable \"%s\" not declared
before usage!\n", countn + 1, c);
            sem_errors++;
            return 0;
    }
      return 1;
}
```

b. Multiple Variable Declarations

The semantic analyser needs to check if the variable being declared has not already been declared before so as to prevent multiple instances of the same identifier in the symbol table at different locations. Furthermore, the analyser should verify the scope of variables to ensure that they are not redeclared in the same scope, such as a loop. The following function is called before inserting any record into the symbol table.

```
if(c == 'I' && q) {
    sprintf(errors[sem_errors], "Line %d: Multiple declarations of \"%s\" not
allowed!\n", countn+1, yytext);
    sem_errors++;
}
```

c. Reservation of Keywords

The semantic analyser needs to ensure that keywords are not declared as variable names by maintaining a list of all the reserved keywords and comparing before adding new symbols to the table. The following function is called before inserting any record into the symbol table.

```
if(c == 'I') {
    for(int i = 0; i < 10; i++){
        if(!strcmp(reserved[i], strdup(yytext))){
            sprintf(errors[sem_errors], "Line %d: Variable name \"%s\" is a
reserved keyword!\n", countn + 1, yytext);
            sem_errors++;
            return;
        }
    }
}
```

d. Checking of Types

The semantic analyser needs to check if the variable being declared has not been declared before so as to prevent multiple instances of the same identifier in the symbol table at different locations. Furthermore, the analyser should verify the scope of variables to ensure that they are not redeclared in the same scope, such as a loop. The following function is called before inserting any record into the symbol table.

```
int check_types(char *type1, char *type2) {
      if(!strcmp(type2, "null"))
            return -1;
      if(!strcmp(type1, type2))
            return 0;
      if(!strcmp(type1, "int") && !strcmp(type2, "float"))
            return 1;
      if(!strcmp(type1, "float") && !strcmp(type2, "int"))
            return 2;
      if(!strcmp(type1, "int") && !strcmp(type2, "char"))
            return 3;
      if(!strcmp(type1, "char") && !strcmp(type2, "int"))
            return 4;
      if(!strcmp(type1, "float") && !strcmp(type2, "char"))
            return 5;
      if(!strcmp(type1, "char") && !strcmp(type2, "float"))
            return 6;
}

char *get_type(char *var){
      for(int i = 0; i < count; i++) {
            if(!strcmp(symbol_table[i].id_name, var)) {
                  return symbol_table[i].data_type;
            }
      }
}
```

Type checking needs to be performed in multiple instances throughout the code to maintain the consistency of the program with the language definition.

i) Type Checking in Declarations and Assignments

```
statement: declaration IDENTIFIER { add('I');} ':' datatype init {
    $5.nd = mknode(NULL, NULL, $5.name); //making for datatype
      $2.nd = mknode(NULL, NULL, $2.name); //making for identifier
      $1.nd = mknode($5.nd, $2.nd, $1.name); //making for the declaration
      int t = check_types($5.name, $6.type); //here we're checking types
      // printf("%s\n", $5.name);
      // printf("%s\n", $6.type);
      // printf("%d\n", t);
      if(t>0) {
            sprintf(errors[sem_errors],  "Line  %d:  Type  mismatch  in  expression!\n",
countn+1);sem_errors++;
      }
      else {
            $$.nd = mknode($2.nd, $6.nd, "Initialisation");
      }
 }
| IDENTIFIER {  } '=' expression {
      if(check_declaration($1.name)){
            $1.nd = mknode(NULL, NULL, $1.name);
            char *id_type = get_type($1.name);
            if(strcmp(id_type, $4.type)) {
                  sprintf(errors[sem_errors],   "Line   %d:   Type   mismatch   in
expression!\n", countn+1); sem_errors++;
            }
            else {
                  $$.nd = mknode($1.nd, $4.nd, "=");
            }
      }
}
| IDENTIFIER { check_declaration($1.name); } relop expression { $1.nd = mknode(NULL,
NULL, $1.name); $$.nd = mknode($1.nd, $4.nd, $3.name ); }
;
```

ii) Type Checking in Expressions

```
expression : expression addops term {
      if(strcmp($1.type, $3.type)){
            sprintf(errors[sem_errors],  "Line  %d:  Type  mismatch  in  expression!\n",
countn+1);sem_errors++;
      }
      else {
            $$.nd = mknode($1.nd, $3.nd, $2.name);
      }
}
| term { strcpy($$.name, $1.name); sprintf($$.type, "%s", $1.type); $$.nd = $1.nd; }
;
```

The **above** is repeated for the **term and factor** production rules.

iii) Type Assignment

Initialization

```
init: '=' value { $$.nd = $2.nd; sprintf($$.type, "%s", $2.type); strcpy($$.name,
$2.name); }
| '=' expression { $$.nd = $2.nd; sprintf($$.type, "%s", $2.type); strcpy($$.name,
$2.name); }
| { sprintf($$.type, "%s", "null"); $$.nd = mknode(NULL, NULL, "NULL");
strcpy($$.name, "NULL"); }
;
```

Assignments

```
number: INTEGER { strcpy($$.name, $1.name); sprintf($$.type, "%s", "number");
add('C'); $$.nd = mknode(NULL, NULL, $1.name); }
| FLOAT { strcpy($$.name, $1.name); sprintf($$.type, "%s", "number"); add('C'); $$.nd
= mknode(NULL, NULL, $1.name); }
;

value: number {$$.nd = mknode(NULL, NULL, $1.name);}
| IDENTIFIER { strcpy($$.name, $1.name); char *id_type = get_type($1.name);
sprintf($$.type, "%s", id_type); check_declaration($1.name); $$.nd = mknode(NULL,
NULL, $1.name); }
| STRINGVALUE { strcpy($$.name, $1.name); sprintf($$.type, "string"); add('C'); $$.nd
= mknode(NULL, NULL, $1.name); }
| TRUE { add('K');} {$$.nd = mknode(NULL, NULL, $1.name);}
| FALSE { add('K');} {$$.nd = mknode(NULL, NULL, $1.name);}
| SCAN { add('K');} '('')' { $$.nd = mknode(NULL, NULL, "scan"); }
;
```

III. YACC Code for Implementing Semantic Analysis and the Symbol Table

```
%{
    #include <stdio.h>
    #include <string.h>
    #include <stdlib.h>
    #include <ctype.h>
    #include "lex.yy.c"

    #define BOLDRED "\033[1m\033[31m"

    int yyerror(const char *s);
    int yylex(void);
    int yywrap();

    struct node *head;
    struct node {
        struct node *left;
        struct node *right;
        char *token;
    };

    struct node* mknode(struct node *left, struct node *right, char *token);
    void printBT(struct node*);

    struct dataType {
        char *id_name;
        char *data_type;
        char *type;
        int   line_no;
    } symbol_table[100];

    char type[100];
    int count = 0;
    int q;
    int sem_errors = 0;
    int label = 0;
    char buff[100];
    char errors[10][100];
    char reserved[10][10] = {"int", "float", "char", "string","void", "if", "else",
"for", "main", "return"};

    void insert_type();
    void add(char c);
    int search(char *);


    void printBTHelper(char* prefix, struct node* ptr, int isLeft);
    void printBT(struct node* ptr);
    void printSymbolTable();
       void printSemanticErrors();
    int check_declaration(char *);
    void check_return_type(char *);
    int check_types(char *, char *);
    char *get_type(char *);
      void insert_type();

    extern int countn;
%}


%union {
      struct var_name {
            char name[100];
            struct node* nd;
      } treeNode;

    struct var_name2 {
        char name[100];
        struct node* nd;
        char type[5];
    } treeNode2;
}

%token <treeNode>  IMPORT FROM AS CONSOLELOG SCAN IF WHILE ELSE RETURN ELIF LET VAR
CONST ADD SUB MULT DIV LOG GE LE GT LT EQ NE TRUE FALSE AND OR NUMBERTYPE STRINGTYPE
BOOLEANTYPE FUNCTION INTEGER FLOAT IDENTIFIER STRINGVALUE
%token <treeNode2> POW
%type  <treeNode>  main importList imports moduleList modules function parameter
parameterList datatype body block console_outputs else condition statement declaration
mulops addops relop return and_or
%type  <treeNode2> init expression value number term factor base exponent
%define parse.error verbose

%%

main: importList body { $$.nd = mknode($1.nd, $2.nd, "Program"); head = $$.nd; }
;

importList: imports importList { $$.nd = mknode($1.nd, $2.nd, "ImportList"); }
| imports { $$.nd = mknode(NULL, $1.nd, "ImportList"); }
| { $$.nd = mknode(NULL, NULL, "ImportList"); }
;

imports: IMPORT '{' moduleList '}' FROM STRINGVALUE ';' { $$.nd = $3.nd; }
| IMPORT '{' modules '}' AS IDENTIFIER FROM STRINGVALUE ';' { $$.nd = $3.nd; }
| IMPORT moduleList FROM STRINGVALUE ';' { $$.nd = $2.nd; }
| IMPORT modules AS IDENTIFIER FROM STRINGVALUE ';' { $$.nd = $2.nd; }
;

moduleList: moduleList ',' moduleList { $$.nd = mknode($1.nd, $3.nd, "ModuleList"); }
| modules { $$.nd = mknode(NULL, NULL, $1.name); }
;
```

```
modules: IDENTIFIER
| MULT
;

body: block body {$$.nd = mknode($1.nd, $2.nd, "Scope"); }
| { $$.nd = mknode(NULL, NULL, "EndOfScope"); }
;

block: function { $$.nd = $1.nd; }
| WHILE { add('K');} '(' condition ')''{' body '}' { $$.nd = mknode($4.nd, $7.nd,
$1.name); }
| IF { add('K');}  '(' condition ')''{' body '}' else { struct node *iff =
mknode($4.nd, $7.nd, $1.name); $$.nd = mknode(iff, $9.nd, "conditionalBranch"); }
| statement ';' { $$.nd = $1.nd; }
| CONSOLELOG { add('K');} '(' console_outputs ')' ';' { struct node *data =
mknode(NULL, NULL, $4.name); $$.nd = mknode(NULL, data, "ConsoleLog"); }
;

console_outputs: STRINGVALUE { add('C');}
| IDENTIFIER  { add('I');}
;


function: FUNCTION { } IDENTIFIER { } '(' parameterList ')' '{' body return '}' {
$9.nd = mknode($9.nd, $10.nd, "FunctionBody"); struct node *main = mknode($9.nd,
$6.nd, $3.name); $$.nd = mknode($1.nd, main, "Function"); }
;

parameterList: parameter ',' parameterList { $$.nd = mknode($1.nd, $3.nd,
"ParameterList"); }
| parameter { $$.nd = $1.nd; }
| { $$.nd = $$.nd = mknode(NULL, NULL, "Parameter"); }
;

parameter: IDENTIFIER { add('I');} ':' datatype {$4.nd = mknode(NULL, NULL, $4.name);
$1.nd = mknode(NULL, NULL, $1.name); $$.nd = mknode($4.nd, $1.nd, "Parameter");}
;


datatype: NUMBERTYPE { insert_type(); }
| STRINGTYPE { insert_type(); }
| BOOLEANTYPE { insert_type(); }
;



else: ELSE { add('K');} '{' body '}' {  struct node *cond = mknode(NULL, NULL,
"EndOfConditional"); $$.nd = mknode($4.nd, cond, $1.name); }
| { $$.nd = NULL; }
;

condition: condition and_or condition { $$.nd = mknode($1.nd, $3.nd, $2.name); }
```

```
| value relop value { $$.nd = mknode($1.nd, $3.nd, $2.name);}
| value { $$.nd = $1.nd;}
| TRUE { add('K');} {$$.nd = NULL; }
| FALSE { add('K');} {$$.nd = NULL; }
;

statement: declaration IDENTIFIER { add('I');} ':' datatype init {
    $5.nd = mknode(NULL, NULL, $5.name);
      $2.nd = mknode(NULL, NULL, $2.name);
      $1.nd = mknode($5.nd, $2.nd, $1.name);
      int t = check_types($5.name, $6.type);
    if(t>0) {
            sprintf(errors[sem_errors], "Line %d: Type mismatch in expression!\n",
countn + 1);sem_errors++;
      }
      else {
            $$.nd = mknode($2.nd, $6.nd, "Initialisation");

      }

}
| IDENTIFIER {  } '=' expression {
      if(check_declaration($1.name)) {
            $1.nd = mknode(NULL, NULL, $1.name);
            char *id_type = get_type($1.name);
            if(strcmp(id_type, $4.type)) {
                  sprintf(errors[sem_errors], "Line %d: Type mismatch in
expression!\n", countn + 1); sem_errors++;
            }
            else {
                  $$.nd = mknode($1.nd, $4.nd, "=");
            }
      }
}
| IDENTIFIER { check_declaration($1.name); } relop expression { $1.nd = mknode(NULL,
NULL, $1.name); $$.nd = mknode($1.nd, $4.nd, $3.name ); }
;

declaration: LET { add('K');}
| VAR { add('K');}
| CONST { add('K');}
;

init: '=' value { $$.nd = $2.nd; sprintf($$.type, "%s", $2.type); strcpy($$.name,
$2.name); }
| '=' expression { $$.nd = $2.nd; sprintf($$.type, "%s", $2.type); strcpy($$.name,
$2.name); }
| { sprintf($$.type, "%s", "null"); $$.nd = mknode(NULL, NULL, "NULL");
strcpy($$.name, "NULL"); }
;


expression : expression addops term {
      if(strcmp($1.type, $3.type)) {
            sprintf(errors[sem_errors], "Line %d: Type mismatch in expression!\n", 
countn + 1);sem_errors++;
      }
      else {
            $$.nd = mknode($1.nd, $3.nd, $2.name);
      }
}
| term { strcpy($$.name, $1.name); sprintf($$.type, "%s", $1.type); $$.nd = $1.nd; }
;

term : term mulops factor {
      if(strcmp($1.type, $3.type)){
            sprintf(errors[sem_errors], "Line %d: Type mismatch in expression!\n", 
countn + 1);sem_errors++;
      }
      else {
            $$.nd = mknode($1.nd, $3.nd, $2.name);
      }

      }
| factor { strcpy($$.name, $1.name); sprintf($$.type, "%s", $1.type); $$.nd = $1.nd; }
;

factor : base exponent base {
      if(strcmp($1.type, $3.type)){
            sprintf(errors[sem_errors], "Line %d: Type mismatch in expression!\n", 
countn + 1);sem_errors++;
      }
      else {
            $$.nd = mknode($1.nd, $3.nd, $2.name);
      }
      }
| LOG '(' value ',' value ')' {$$.nd = mknode($3.nd, $5.nd, $1.name); }
| base { strcpy($$.name, $1.name); sprintf($$.type, "%s", $1.type); $$.nd = $1.nd; }
;

base : value  { strcpy($$.name, $1.name); sprintf($$.type, "%s", $1.type); $$.nd = 
$1.nd; }
| '(' expression ')'  { strcpy($$.name, $2.name); sprintf($$.type, "%s", $2.type); 
$$.nd = $2.nd; }
;

and_or : AND { $$.nd = mknode(NULL, NULL, $1.name); }
| OR { $$.nd = mknode(NULL, NULL, $1.name); }
;

exponent: POW
;

mulops: MULT
| DIV
;
```

```
addops: ADD
| SUB
;

relop: LT
| GT
| LE
| GE
| EQ
| NE
;

number: INTEGER { strcpy($$.name, $1.name); sprintf($$.type, "%s", "number");
add('C'); $$.nd = mknode(NULL, NULL, $1.name); }
| FLOAT { strcpy($$.name, $1.name); sprintf($$.type, "%s", "number"); add('C'); $$.nd
= mknode(NULL, NULL, $1.name); }
;

value: number {$$.nd = mknode(NULL, NULL, $1.name);}
| IDENTIFIER { strcpy($$.name, $1.name); char *id_type = get_type($1.name);
sprintf($$.type, "%s", id_type); check_declaration($1.name); $$.nd = mknode(NULL,
NULL, $1.name); }
| STRINGVALUE { strcpy($$.name, $1.name); sprintf($$.type, "string"); add('C'); $$.nd
= mknode(NULL, NULL, $1.name); }
| TRUE { add('K');} {$$.nd = mknode(NULL, NULL, $1.name);}
| FALSE { add('K');} {$$.nd = mknode(NULL, NULL, $1.name);}
| SCAN { add('K');} '('')' { $$.nd = mknode(NULL, NULL, "scan"); }
;

return: RETURN { add('K');} value ';'  { check_return_type($3.name); $1.nd =
mknode(NULL, NULL, "return"); $$.nd = mknode($1.nd, $3.nd, "ReturnStatement"); }
| { $$.nd = NULL; }
;

%%


int main() {
    extern FILE *yyin, *yyout;

    int p = -1;
    p = yyparse();
    if(p)
        printf("Parsing Successful\n");

    printf("\033[4mParse Tree\033[24m");
    printf("\n\n");
       printBT(head);
    printf("\n\n");

    printSymbolTable();
    printSemanticErrors();
```

```c
    return p;
}

int yyerror(const char *msg) {
    extern int yylineno;
    printf("%sParsing Failed\nLine Number: %d,%s\n",BOLDRED,yylineno,msg);
    exit(0);
    return 0;
}

void printBTHelper(char* prefix, struct node* ptr, int isLeft) {
    if( ptr != NULL ) {
        printf("%s",prefix);
        if(isLeft) { printf("├──"); }
             else { printf("└──"); }
        printf("%s",ptr->token);
             printf("\n");
             char* addon = isLeft ? "│   " : "    ";
      int len2 = strlen(addon);
      int len1 = strlen(prefix);
      char* result = (char*)malloc(len1 + len2 + 1);
      strcpy(result, prefix);
      strcpy(result + len1, addon);
             printBTHelper(result, ptr->left, 1);
             printBTHelper(result, ptr->right, 0);
      free(result);
    }
}

void printBT(struct node* ptr) {
      printf("\n");
    printBTHelper("", ptr, 0);
}

struct node* mknode(struct node *left, struct node *right, char *token) {
      struct node *newnode = (struct node *)malloc(sizeof(struct node));
      char *newstr = (char *)malloc(strlen(token)+1);
      strcpy(newstr, token);
      newnode->left = left;
      newnode->right = right;
      newnode->token = newstr;
      return(newnode);
}

void printSymbolTable(){
    printf("\n\n");
      printf("\nSYMBOL\tDATATYPE\tTYPE\tLINE NUMBER\n");
      printf("_______________________________________\n\n");

      int i=0;

      for(i=0; i<count; i++) {
```

```
			printf("%s\t%s\t%s\t%d\t\n", symbol_table[i].id_name,
symbol_table[i].data_type, symbol_table[i].type, symbol_table[i].line_no);
		}

		for(i=0;i<count;i++) {
			free(symbol_table[i].id_name);
			free(symbol_table[i].type);
		}
		printf("\n\n");
}


void printSemanticErrors(){
    printf("\n\n\n\n");
	if(sem_errors>0) {
		printf("Semantic analysis completed with %d errors\n", sem_errors);
		for(int i=0; i<sem_errors; i++){
			printf("\t - %s", errors[i]);
		}
	} else {
		printf("Semantic analysis completed with no errors");
	}
	printf("\n\n");
}


int search(char *type) {
	int i;
	for(i=count-1; i>=0; i--) {
		if(strcmp(symbol_table[i].id_name, type)==0) {
			return -1;
			break;
		}
	}
	return 0;
}

int check_declaration(char *c) {
    q = search(c);
    if(!q) {
        sprintf(errors[sem_errors], "Line %d: Variable \"%s\" not declared before
usage!\n", countn + 1, c);
		sem_errors++;
		return 0;
    }
	return 1;
}

void check_return_type(char *value) {
	char *main_datatype = get_type("main");
	char *return_datatype = get_type(value);
	if((!strcmp(main_datatype, "int") && !strcmp(return_datatype, "CONST")) ||
!strcmp(main_datatype, return_datatype)){
			return ;
		}
		else {
			sprintf(errors[sem_errors], "Line %d: Return type mismatch\n", countn +
1);
			sem_errors++;
		}
}

int check_types(char *type1, char *type2){
	if(!strcmp(type2, "null"))
		return -1;
	if(!strcmp(type1, type2))
		return 0;
	if(!strcmp(type1, "int") && !strcmp(type2, "float"))
		return 1;
	if(!strcmp(type1, "float") && !strcmp(type2, "int"))
		return 2;
	if(!strcmp(type1, "int") && !strcmp(type2, "char"))
		return 3;
	if(!strcmp(type1, "char") && !strcmp(type2, "int"))
		return 4;
	if(!strcmp(type1, "float") && !strcmp(type2, "char"))
		return 5;
	if(!strcmp(type1, "char") && !strcmp(type2, "float"))
		return 6;
}

char *get_type(char *var){
	for(int i=0; i<count; i++) {
		if(!strcmp(symbol_table[i].id_name, var)) {
			return symbol_table[i].data_type;
		}
	}
}


void add(char c) {
	if(c == 'I'){
		for(int i=0; i<10; i++){
			if(!strcmp(reserved[i], strdup(yytext))){
			sprintf(errors[sem_errors], "Line %d: Variable name \"%s\" is a
reserved keyword!\n", countn + 1, yytext);
				sem_errors++;
				return;
			}
		}
	}
    q=search(yytext);
	if(!q) {
		if(c == 'H') {
			symbol_table[count].id_name=strdup(yytext);
			symbol_table[count].data_type=strdup(type);
```

```
                    symbol_table[count].line_no=countn;
                    symbol_table[count].type=strdup("Header");
                    count++;
             }
             else if(c == 'K') {
                    symbol_table[count].id_name=strdup(yytext);
                    symbol_table[count].data_type=strdup("N/A");
                    symbol_table[count].line_no=countn;
                    symbol_table[count].type=strdup("Keyword\t");
                    count++;
             }
             else if(c == 'I') {
                    symbol_table[count].id_name=strdup(yytext);
                    symbol_table[count].data_type=strdup(type);
                    symbol_table[count].line_no=countn;
                    symbol_table[count].type=strdup("Variable");
                    count++;
             }
             else if(c == 'C') {
                    symbol_table[count].id_name=strdup(yytext);
                    symbol_table[count].data_type=strdup("CONST");
                    symbol_table[count].line_no=countn;
                    symbol_table[count].type=strdup("Constant");
                    count++;
             }
             else if(c == 'F') {
                    symbol_table[count].id_name=strdup(yytext);
                    symbol_table[count].data_type=strdup(type);
                    symbol_table[count].line_no=countn;
                    symbol_table[count].type=strdup("Function");
                    count++;
             }
    }
    else if(c == 'I' && q) {
        sprintf(errors[sem_errors], "Line %d: Multiple declarations of \"%s\" not 
allowed!\n", countn + 1, yytext);
             sem_errors++;
    }
}


void insert_type() {
      strcpy(type, yytext);
}
```

IV. Sample Input Program and Terminal Output

Input Program (TypeScript File)

```
// Ignored
/* Ignored */

import { importedValue1, importedValue2 } from "../Syntax Analyser";

function trialFunction (param1: number, param2: string) {
    let valueNumber:number = 3;
    const valueBoolean:boolean = true;
    var valueExpression:number = 5 * 6;

    if(valueNumber > valueExpression && valueBoolean == true) {
        console.log("ABC");
    }
    else{
        console.log(valueBoolean);
    }

    let loopValue:number = 10;
    while(loopValue) {
        loopValue = loopValue - 1;
    }

    return 3;
}
```

Terminal Output : Generated Parse Tree

```
Parse Tree

└──Program
    ├──ImportList
    │   └──ModuleList
    │       ├──importedValue1
    │       └──importedValue2
    └──Scope
        ├──Function
        │   └──trialFunction
        │       ├──FunctionBody
        │       │   ├──Scope
        │       │   │   ├──Initialisation
        │       │   │   │   ├──let
        │       │   │   │   │   ├──number
        │       │   │   │   │   └──valueNumber
        │       │   │   │   └──3
        │       │   │   └──Scope
        │       │   │       ├──Initialisation
        │       │   │       │   ├──const
        │       │   │       │   │   ├──boolean
│       │   │       │   │   └──valueBoolean
│       │   │       │   └──true
│       │   │       └──Scope
│       │   │           ├──Initialisation
│       │   │           │   ├──var
│       │   │           │   │   ├──number
│       │   │           │   │   └──valueExpression
│       │   │           │   └──*
│       │   │           │       ├──5
│       │   │           │       └──6
│       │   │           └──Scope
│       │   │               ├──conditionalBranch
│       │   │               │   ├──if
│       │   │               │   │   ├──&&
│       │   │               │   │   │   ├──>
│       │   │               │   │   │   │   ├──valueNumber
│       │   │               │   │   │   │   └──valueExpression
│       │   │               │   │   │   └──==
│       │   │               │   │   │       ├──valueBoolean
│       │   │               │   │   │       └──true
│       │   │               │   │   └──Scope
│       │   │               │   │       ├──ConsoleLog
│       │   │               │   │       │   └──"ABC"
│       │   │               │   │       └──EndOfScope
│       │   │               │   └──else
│       │   │               │       ├──Scope
│       │   │               │       │   ├──ConsoleLog
│       │   │               │       │   │   └──valueBoolean
│       │   │               │       │   └──EndOfScope
│       │   │               │       └──EndOfConditional
│       │   │               └──Scope
│       │   │                   ├──Initialisation
│       │   │                   │   ├──let
│       │   │                   │   │   ├──number
│       │   │                   │   │   └──loopValue
│       │   │                   │   └──10
│       │   │                   └──Scope
│       │   │                       ├──while
│       │   │                       │   ├──loopValue
│       │   │                       │   └──Scope
│       │   │                       │       ├──=
│       │   │                       │       │   ├──loopValue
│       │   │                       │       │   └──-
│       │   │                       │       │       ├──loopValue
│       │   │                       │       │       └──1
│       │   │                       │       └──EndOfScope
│       │   │                       └──EndOfScope
│       │   └──ReturnStatement
│       │       ├──return
│       │       └──3
│       └──ParameterList
│           ├──Parameter
│           │   ├──number
```

```
        │           │   └──param1
        │           └──Parameter
        │               ├──string
        │               └──param2
        └──EndOfScope
```

Terminal Output : Generated Symbol Table

```
SYMBOL          DATATYPE        TYPE        LINE NUMBER
_______________________________________________________
import                          Header      4
trialFunction   N/A             Function    6
param1          number          Variable    6
param2          string          Variable    6
valueNumber     number          Variable    7
3               CONST           Constant    7
valueBoolean    booolean        Variable    8
true            CONST           Constant    8
valueExpression number          Variable    9
5               CONST           Constant    9
6               CONST           Constant    9
if              N/A             Keyword     11
console.log     N/A             Keyword     12
loopValue       number          Variable    18
10              CONST           Constant    18
while           N/A             Keyword     19
1               CONST           Constant    20
return          N/A             Keyword     23
```

Terminal Output : Generated Semantic Errors

```
Semantic analysis completed with no errors
```

V. Semantic Error Handling with Terminal Output

Input Program (No Variable Declaration)

```
let valueString: string = "Dru";
valueNumber = 789;
```

Terminal Output : Generated Semantic Errors

```
Semantic analysis completed with 1 errors
Line 2: Variable "valueNumber" not declared before usage
```

Input Program (Multiple Variable Declarations)

```
let valueString: string = "Dru";
let valueString: string = "Merc";
let valueString: string = "Nish";
```

Terminal Output : Generated Semantic Errors

```
Semantic analysis completed with 2 errors
Line 2: Multiple declarations of "valueString" not allowed
Line 3: Multiple declarations of "valueString" not allowed
```

Input Program (Reservation of Keywords)

```
let console.log: string = "Dru";
```

Terminal Output : Generated Semantic Errors

```
Line Number: 1,syntax error, unexpected CONSOLELOG, expecting IDENTIFIER
```

Input Program (Checking of Types)

```
let valueString:string = "Dru";
let valueNumber:number = 789;
let valueFloat:number = valueString;

let exprValue:number = valueNumber + valueString;
```

Terminal Output : Generated Semantic Errors

```
Semantic analysis completed with 3 errors
Line 3: Type mismatch in expression
Line 5: Type mismatch in expression
Line 5: Type mismatch in expression
```