Compiler Design Lab
Mini Compiler in C

Report

on

Project 3

Semantic Checker

for C- language

by

Ritwick Mishra, 15CO240
Abhilash Sanap, 15CO241
Department of CSE, NITK

Table of Contents

Introduction

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:

```
char  a = 100;
```

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
- Array-bound checking

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 Attribute : get values only from the attribute values of their child nodes.
Inherited Attributes : can take values from parent and/or siblings.

Syntax Directed Translation

It refers to a method of compiler implementation where the source language translation is completely driven by the parser.

A common method of syntax-directed translation is translating a string into a sequence of actions by attaching one such action to each rule of a grammar. Thus, parsing a string of the grammar produces a sequence of rule applications. SDT provides a simple way to attach semantics to any such syntax.

S- attributed SDT

If an SDT uses only synthesized attributes, it is called as S-attributed SDT.

L- attributed SDT

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

Overview of the code

The lex file has no changes added to it, except for the addition of a buffer idname which stores the identifier name. All the semantic checks were added in the yacc file itself. The yacc file is named par.y . Like a typical yacc file, it has 4 sections: the header section, the definition section, the grammar section and lastly the C program to use the defined grammar.

Header section

All the relevant header files have been included including stdio.h, stdlib.h, string.h etc. The variables that have been used in the grammar section must be declared in this section. A few variables have been declared as extern as they refer to the same variable in the lex file.
idname[ ] is a character buffer where we store the name of the identifier when the lexer encounter one. This buffer is used in the parser file to add to the symbol table provided it is valid and legal. Otherwise it throws an error.
dim is a variable used in the parser file to store the value of the dimension of the array while checking the syntax of the declaration.
The typical functions that deal with the hash array implementation of the symbol table are used: hashCode(), insert_hash(), display_hash(). Each node of the symbol table has the following fields:

```
struct DataItem
    {
        char* text;             /* The token*/
        char* type;         /* Type of the token*/
        int scope;
        int lineno;         /* Line number in the input file*/
        int arrdim;
        char* retype;

        struct DataItem * next;     /*Pointer to next data item*/
    };
```

To add the functionality of scope checking in semantic phase, a few variables are introduced: scopesLength, scopeCounter. The functions used for this are: getCurrentScope(), startNewScope(), endCurrentScope(), isScopeValid().
Apart from this, hash_lookup_scope(), getidtype() are called throughout the grammar to implement checks. Variables like typ, op1 ,op2 are used to hold the strings that are later used to check the types of the variables used in arithmetic operations.

Declaration section

There are no further changes in the definition section after the parser phase implementation. This section defines the tokens, the associativity and precedence order of the tokens.

Note: %token UMINUS is pseudo token that helps in treating negative numbers.

Grammar rules section

This section contains most of the semantic checks implemented in the semantic phase. The start variable is *program*. This grammar rules check the syntax of the input program and throw errors if these are violated. Also these rules include actions which help in populating the symbol table and also implement semantic checks.
The scope is basically regulated by this:

```
compstmt: block_start stmtlist block_end
;
block_start: LBRACE
```

```
{
      startNewScope();
      //printf("%d\n",getCurrentScope());
}
block_end: RBRACE
{
      endCurrentScope();

}
```

The various kinds of C statements supported by our mini compiler are defined by:

```
stmtlist: stmtlist stmt
| stmt
;
stmt: declaration
| funcall
| whilestmt
| ifstmt
| returnstmt
| printstmt
| scanstmt
| expr
| SEMICOLON
;
```

The code is self-evident. In the next section of the report we will show the actual code which performs the semantic checks.

Listing of the code for the semantic checks performed

```
int getCurrentScope ()
{
      return scopes[scopesLength - 1];
}
```

This returns the current scope in which the program control is there. The scopes[ ] array is an integer array. scopesLength is a variable to keep track of the length of this array.

```
void startNewScope ()
{
   scopes[scopesLength++] = scopeCounter++;
}
```

This stores the scopeCounter which is a variable to identify a particular scope of a block of code, at the end of the scopes array, before incrementing it. This signifies the start of a scope.

```
void endCurrentScope ()
{
    scopesLength--;
}
```

This effectively deletes the scope when it exits a particular block of code.

```
int isScopeValid (int scope)
{
   int i=0;
   for (i = scopesLength-1; i>=0; i--)
        if (scopes[i] == scope)
            return 1;
   return 0;
}
```

This function checks if a particular scope is ‘ *valid*’ or not. This means given a scope, it checks the scopes array if it is active or deleted. That is why it traverses through the array in the active secton if the number correspponding to the scope is present or not.

```
int hash_lookup_scope(char * key)
{
int hashIndex = hashCode(key);
if(hashArray[hashIndex]!=NULL){
      struct DataItem * temp = hashArray[hashIndex];
      while(temp!=NULL){
            if(strcmp(temp->text,key)==0){
                  int sc=temp->scope;
                  if(isScopeValid(sc)==1)
                        return 1;
            }
            temp=temp->next;
      }
}
```
This function, given a variable name, scans through the symbol table for the variable. Once it arrives at the right node, it checks the scope stored in that node to be valid or not using the previous function.
```
void getidtype(char * key){

      if(hash_lookup_scope(key)==1){
            int hashIndex=hashCode(key);
            struct DataItem * temp = hashArray[hashIndex];
                  while(temp!=NULL){
                     if(strcmp(temp->text,key)==0){
                        int sc=temp->scope;
                          if(isScopeValid(sc)==1){

                              typ=(char
*)malloc(sizeof(char)*strlen(temp->type));
                              strcpy(typ,temp->type);

                              }
                     }
                        temp=temp->next;
                  }
            }
}
```
This function fetches the type of the variable, given the name of the variable. This is performed in a similar manner as above.
```

compstmt: block_start stmtlist block_end
;
block_start: LBRACE
{
      startNewScope();
      //printf("%d\n",getCurrentScope());
}
```

```
block_end: RBRACE
{
	endCurrentScope();

```

These rules start a new scope whenever it encounters a '{'. and ends current scope when it encounters a '}'.

```
funcall: fc expr COMMA expr RPAREN SEMICOLON
| fc expr RPAREN SEMICOLON
| fc RPAREN SEMICOLON
;
fc: ID LPAREN
{
	int hashIndex = hashCode(idname);
	int flag=0;
	if(hashArray[hashIndex]!=NULL){
	struct DataItem * temp = hashArray[hashIndex];

	while(temp!=NULL){
		if(strcmp(temp->text,idname)==0 &&
strcmp(temp->type,"function")==0){
			flag=1;
			break;
		}
		temp=temp->next;
	}
	}
	if(flag==0){
		printf("Undefined function: '%s' in line
%d\n",idname,yylineno);
	}

}
```

This piece of code allows for functions to have atmost 2 parameters. Also this checks for if the function call is valid or not, i.e., if the function has a defintion in the code elsewhere. Otherwise it throws an error 'Undefined function'.

```
expr: identifier {flg=0;}
| NUM {flg=1;}
| CHLIT    {flg=2;}
| identifier ASSIGN {getidtype(idname) ;  if(strcmp(typ,"int")!=0)
{printf("LHS of assignment should be int in line %d",yylineno);
return -1;} } expr
```

This flag is set to 0 if identifier, 1 if integer literal and 2 if character literal.
In an expression involving the ASSIGN (‘=’) token, the identifier on the LHS should be of type integer. This is implemented here.

```
expr ADD{
     if(flg==0){
     getidtype(idname); op1=(char
*)malloc(strlen(typ)*sizeof(char));  strcpy(op1,typ);}
     if(flg==2){
          printf("Character can't be operand\n in line
%d",yylineno);
          return -1;
     }
} expr    {
if(flg==0){
     getidtype(idname); op2=(char
*)malloc(strlen(typ)*sizeof(char));  strcpy(op2,typ);
     if(strcmp(op1,op2)!=0){
          printf("Operand are of different types in line
%d\n",yylineno);
          return -1;
     }
     }
     if(flg==2){
          printf("Character can't be operand\n in line
%d",yylineno);
          return -1;
     }
}
```

This piece of code is easily understandable. In ADD operator, the two operands must be of type int. Hence this has been implemented in a simple manner.
This code is replicated for each of the arithmetic operations like SUB, MUL etc.

```
| SUB expr            %prec UMINUS
{
      if(flg!=1){
            printf("Invalid operand in line %d",yylineno);
            return -1;
      }
}
;
```

This is the rule to treat a ‘-’ symbol in front of a token. The token has to be an integer literal to make any sense.

```
identifier: ID
{
            if(hash_lookup_scope(idname)==0){
                  printf("Undeclared Variable: '%s' in line
%d\n",idname,yylineno);
                  return -1;
            }

}
;
This checks the variable if it has been declared in the scope or not.
arrid: ID LSQ
{

      if(hash_lookup_scope(idname)==0){
            printf("Undeclared Variable: '%s' in line
%d\n",idname,yylineno);
            return -1;
      }
      int hashIndex = hashCode(idname);
      int f=0;
      struct DataItem * temp = hashArray[hashIndex];

      while(temp!=NULL){
            if(strcmp(temp->text,idname)==0){
                  int sc=temp->scope;
```

```
                 if(isScopeValid(sc)==1){
                      if(strcmp(temp->type,"array")!=0){
                           f=1;
                      }
                      else gdim=temp->arrdim;
                 }

            }
            temp=temp->next;
      }
      if(f==1){
            printf("Single variable id or function has subscript\n");
            return -1;
      }

return 0;

}
;
```

This checks for:

- If the array variable is undeclared.
- If a single variable identifier or function has subscript.

Both of these are not allowed.

```
declarray:  INT ID LSQ NUM RSQ SEMICOLON
{
      dim = $4;
      if(dim<1){
            printf("Array size less than 1\n");
            return -1;
      }

insert_hash(idname,"array",getCurrentScope(),yylineno,dim,"NA");
}
;
```

This checks if array size is less than 1 in the declaration of the array. Otherwise it is legal and is inserted in the symbol table.

```
 if(hashArray[hashIndex]!=NULL){
            struct DataItem * temp = hashArray[hashIndex];
            while(temp!=NULL){
            if(strcmp(temp->text,text)==0){
                 int sc=temp->scope;
                 if(sc==scope)
                      {printf("Multiple declarations: '%s' in line
%d\nVariable already declared in line
%d\n\n",text,lineno,temp->lineno);
                      return;
                      }
            }
            temp=temp->next;
      }

   }
```

This code is found in the insert_hash() function. This checks if the variable name has already been declared in the current scope. If so it stops the insertion and throws an error for multiple declaration.

Description of the checks performed

- **Scope Checking:**
  Scope is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. It is a common mistake by programmers to access a variable out of its scope. It is one of the basic functionality provided by any compiler to check the validity of scope of a variable and to show errors whenever any variable is accessed out of its scope.
  As described earlier, we have defined few functions that help us in overall scope checking. We discuss the outline of the same in this section. Whenever a variable is defined in a program, we simply insert it in the symbol table with an appropriate scope. Whenever a variable is used in the program, we check if it is accessed in its allowed scope. If not we put out an error message “Undeclared Variable: ”.
- **Type Checking:**

This refers to checking if a particular operator has operands whose types are allowed for it. For example, a modulo operator enforces its operands to be of integer type. In our implementation, we check if operands given to arithmetic operators like addition, subtraction, multiplication, division and comparison operators like greater or equal, lesser or equal, greater than, lesser than, is equal to, is not equal to have **identical** type. If not, we show an error message "Operands are of different types".
We also disallow any of the operand to be of type of "char" for certain operators. We achieve this by checking the type of both the operands of an operator in the code. We search for entry of the variable corresponding to the operands in the symbol table and fetch its type from the type field.

- **Checking for function definition:**
  Whenever a function is called, we check for its definition. If the definition is not provided in the code, we put an error message " Function not defined..". This is done by searching the symbol table for the identifier corresponding to the function name. If an entry matching the identifier exists and the type of the matched entry is "function", then we let the code execute. In any other case, we put the error message to the console.
- **Non-array identifiers with subscript:**
  Only arrays are allowed to have a subscript that denotes the offset from its base. In C language, there is no meaning to subscript after a normal variable or a function. But by mistake, the code may have it. Hence, on encountering such illegal usage, we print an error message
- **Array size less than 1 :**
  Arrays are allowed to have sizes greater or equal 1. So any illegal declaration is met with an error message compelling the programmer to correct the silly error committed from their side.

Description of Symbol table

The Symbol Table is implemented by using the concept of "Chain Hashing". For each entry to be included in symbol table, we calculate a hash index by using a function called HashCode(). It takes the identifier name as input and returns an index which is unique. If the same identifier appears again in the code, we get the same hash Index from the HashCode() function, i.e., a collision. We handle this by chaining the new entry to the current entry in the symbol table. While chaining, we insert the recent entry at the head of the linked list that is formed by the symbol table entries whose hash index

comes out to be the same. The choice of inserting a new symbol entry with an occupied hash index location at the head of it is made to reduce the time taken for checking the validity of the scope of a variable.
Following is a snapshot of code that shows all the attributes of a symbol to be stored in the symbol table. We'll discuss them in brief.

```
struct DataItem
    {
        char* text;             /* The token*/
        char* type;         /* Type of the token*/
        int scope;
        int lineno;         /* Line number in the input file*/
        int arrdim;
        char* retype;

        struct DataItem * next;     /*Pointer to next data item*/
    };
```

- text : Actual identifier as returned by the lexer
- type : Identifier can be of different type as int, char, unsigned int, etc. It could also be an array or a function.
- Scope: It is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. In our implementation we give whole number values to scope of a variable.The global scope has a default value of zero. Each new scope is given an incremented value. Scope checking is made easy due to simple implementation of the same
- Lineno: It is the line number on which the token appears in the code. Keeping a track of line number eases locating an error. If any error is present in the code, the compiler gives out the error with the line number.
- arrdim : Only for arrays, we store the dimensions of the same. Non-array entries have "NA" in this particular field
- Retype : Only for functions, the return type of a function. We can use this field to check if a particular function is returning a valid datatype.
- DataItem * next : It is a pointer to next DataItem node. It is used when collision happens, i.e., two entries that return same hashIndex appear or simply same variable appears at multiple places in the code.

We print the symbol table at end of each parsing irrespective of the success of the parsing.

CODE

lex.l

```
%{
    #include <string.h>
    #include <stdlib.h>
    char idname[1000];
%}
alpha [a-zA-Z_]
digit [-]?[0-9]
charlit [a-zA-Z0-9]
%%
[ \t]           ;                             /*Removes spaces and tabs */
[ \n]   { yylineno = yylineno + 1;}
int     { return INT;}
char     {return CHAR;}
void    {return VOID;}
while    {  return WHILE;}
if      {  return IF;}
else    { return ELSE;}
printf  { return PRINTF;}
scanf     { return SCANF;}
break    { return BREAK;}
^"#include ".+ ;
return    { return RETURN;}
{digit}+   {yylval=atoi(yytext); return NUM;}
\'{charlit}\'                 {     return CHLIT;}
{alpha}({alpha}|{digit})* { strcpy(idname,yytext); return ID;}



"{"    {     return LBRACE;}
"}"     {      return RBRACE;}
"["    {       return LSQ;}
"]"     {      return RSQ;}
";"     {      return SEMICOLON;}
"("     {      return LPAREN;}
")"     {      return RPAREN;}
"&&"    {      return DAND;}
"||"    {      return DOR;}
"&"     {      return AND;}
"|"     {      return OR;}
```

```
"!"      {      return NOT;}
","      {      return COMMA;}
"."      {      return DOT;}
"'"      {      return QM;}

    /*Operators*/
"+"      { return ADD;}
"-"      { return SUB;}
"*"      { return MUL;}
"%"      { return MOD;}
"/"      {      return DIV;}
"=="     {      return ISEQ;}
"!="     {      return NEQ;}
"="      {      return ASSIGN;}
">="     {      return GEQ;}
"<="     {      return LEQ;}
">"      {      return GT;}
"<"      {      return LT;}

\"(\\.|[^\\"])*\"    {return STRINGLIT;}

    /*Comments*/
\/\/.*    ;
\/\*(.*\n)*.*\*\/    ;
.               {printf("\nIllegal token: %s at line no :
%d\nExiting...\n\n",yytext,yylineno);
                return -1;
                }

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

par.y

```
%{
       #include <stdio.h>
       #include <stdlib.h>
       #include <string.h>
       char * typ;
       extern FILE *fp;
       extern char idname[1000];
       extern int yylineno;
```

```
        int yylex(void);
        int dim=0;
        void yyerror(char*);
        #define SIZE 1000
        int hashCode(char* key);
        void insert_hash(char *,char *,int,int,int, char *);
        void display_hash();
        struct DataItem
        {
        char* text;           /* The token*/
        char* type;           /* Type of the token*/
        int scope;
        int lineno;           /* Line number in the input file*/
        int arrdim;
        char* retype;

        struct DataItem * next;      /*Pointer to next data item*/
        };
        struct DataItem* hashArray[SIZE];
        struct DataItem* item;
        int scopes[1000];
int scopesLength = 1;
int scopeCounter = 1;
int getCurrentScope();
void startNewScope();
void endCurrentScope();
int isScopeValid(int);
int hash_lookup_scope(char * key);
void getidtype(char *);
int gdim=0;
char * typ, * op1,* op2;
int flg=0;
%}
%token ID NUM CHLIT STRINGLIT INT VOID CHAR UMINUS
%token IF ELSE WHILE BREAK SCANF PRINTF RETURN
%token LBRACE RBRACE LPAREN RPAREN SEMICOLON COMMA DOT LSQ RSQ QM
%token AND NOT OR
%token ADD SUB MUL DIV MOD ASSIGN
%token ISEQ GT LT GEQ LEQ DAND DOR NEQ
%left ADD SUB
%left MUL DIV
%left GT LT GEQ LEQ ISEQ NEQ
%left AND OR
%left DAND DOR
%right UMINUS
%right ASSIGN
%left ELSE
%start program
```

```
%%
program: declist funclist
| funclist
;
declist: declist declstmt
| declstmt
;
declstmt: INT intid assign COMMA intid  assign SEMICOLON
|INT intid  assign SEMICOLON
|CHAR cid assign COMMA cid  assign SEMICOLON
|CHAR cid  assign SEMICOLON
| declarray
;
intid: ID
{

    insert_hash(idname,"int",getCurrentScope(),yylineno,0,"NA");
}
;
cid: ID
{
    insert_hash(idname,"char",getCurrentScope(),yylineno,0,"NA");
}
;
assign: ASSIGN expr
|
;
funclist: funclist funcdecl
| funcdecl
;
funcdecl: INT ifid INT intid COMMA INT intid RPAREN compstmt
| INT ifid INT intid RPAREN compstmt
| INT ifid RPAREN compstmt
| VOID vfid INT intid COMMA INT intid RPAREN compstmt
| VOID vfid LPAREN INT intid RPAREN compstmt
| VOID vfid LPAREN RPAREN compstmt
;
ifid: ID LPAREN
{
    insert_hash(idname,"function",getCurrentScope(),yylineno,0,"int");
}
;
vfid: ID LPAREN
{
    insert_hash(idname,"function",getCurrentScope(),yylineno,0,"void");
}
;
compstmt: block_start stmtlist block_end
```

```
;
block_start: LBRACE
{
    startNewScope();
    //printf("%d\n",getCurrentScope());
}
block_end: RBRACE
{
    endCurrentScope();

}
;
stmtlist: stmtlist stmt
| stmt
;
stmt: declaration
| funcall
| whilestmt
| ifstmt
| returnstmt
| printstmt
| scanstmt
| expr
| SEMICOLON
;
declaration: INT intid assign COMMA intid  assign SEMICOLON
|INT intid  assign SEMICOLON
|CHAR cid assign COMMA cid  assign SEMICOLON
|CHAR cid  assign SEMICOLON
| declarray
;
funcall: fc expr COMMA expr RPAREN SEMICOLON
| fc expr RPAREN SEMICOLON
| fc RPAREN SEMICOLON
;
fc: ID LPAREN
{
    int hashIndex = hashCode(idname);
    int flag=0;
    if(hashArray[hashIndex]!=NULL){
    struct DataItem * temp = hashArray[hashIndex];
    while(temp!=NULL){
        if(strcmp(temp->text,idname)==0 && strcmp(temp->type,"function")==0){
               flag=1;
               break;
        }
        temp=temp->next;
    }
    }
    if(flag==0){
        printf("Undefined function: '%s' in line %d\n",idname,yylineno);
    }

}
;
whilestmt: WHILE LPAREN expr RPAREN compstmt
| WHILE LPAREN expr RPAREN stmt
;
ifstmt: IF LPAREN expr RPAREN compstmt
| IF LPAREN expr RPAREN compstmt ELSE compstmt
| IF LPAREN expr RPAREN stmt
;
returnstmt: RETURN expr SEMICOLON
| RETURN SEMICOLON
;
printstmt: PRINTF LPAREN STRINGLIT RPAREN SEMICOLON
| PRINTF LPAREN STRINGLIT COMMA ID RPAREN SEMICOLON
| PRINTF LPAREN STRINGLIT COMMA ID COMMA ID RPAREN SEMICOLON
;
scanstmt: SCANF LPAREN STRINGLIT RPAREN SEMICOLON
| SCANF LPAREN STRINGLIT COMMA AND ID RPAREN SEMICOLON
| SCANF LPAREN STRINGLIT COMMA AND ID COMMA AND ID RPAREN SEMICOLON
;
expr: identifier    {flg=0;}
| NUM {flg=1;}
| CHLIT    {flg=2;}
| identifier ASSIGN {getidtype(idname) ;  if(strcmp(typ,"int")!=0) {printf("LHS of
assignment should be int in line %d",yylineno); return -1;} } expr
| arrid expr RSQ
| arrid expr RSQ ASSIGN expr
| arrid expr RSQ ASSIGN funcall
| expr ADD{
    if(flg==0){
    getidtype(idname); op1=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op1,typ);}
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
} expr    {
if(flg==0){
    getidtype(idname); op2=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op2,typ);
    if(strcmp(op1,op2)!=0){
        printf("Operand are of different types in line %d\n",yylineno);
        return -1;
    }
    }
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
}
| expr SUB {
    if(flg==0){
    getidtype(idname); op1=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op1,typ);}
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
} expr {
if(flg==0){
    getidtype(idname); op2=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op2,typ);
    if(strcmp(op1,op2)!=0){
        printf("Operand are of different types in line %d\n",yylineno);
        return -1;
    }
    }
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
}
| expr MUL {
    if(flg==0){
    getidtype(idname); op1=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op1,typ);}
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
} expr {
if(flg==0){
    getidtype(idname); op2=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op2,typ);
    if(strcmp(op1,op2)!=0){
        printf("Operand are of different types in line %d\n",yylineno);
        return -1;
    }
    }
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
}
| expr DIV {
    if(flg==0){
    getidtype(idname); op1=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op1,typ);}
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
} expr {
if(flg==0){
    getidtype(idname); op2=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op2,typ);
    if(strcmp(op1,op2)!=0){
        printf("Operand are of different types in line %d\n",yylineno);
        return -1;
    }
    }
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
}
| expr GEQ {
    if(flg==0){
    getidtype(idname); op1=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op1,typ);}
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
} expr {
if(flg==0){
    getidtype(idname); op2=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op2,typ);
    if(strcmp(op1,op2)!=0){
        printf("Operand are of different types in line %d\n",yylineno);
        return -1;
    }
    }
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
}
| expr LEQ {
    if(flg==0){
    getidtype(idname); op1=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op1,typ);}
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
} expr {
if(flg==0){
    getidtype(idname); op2=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op2,typ);
    if(strcmp(op1,op2)!=0){
        printf("Operand are of different types in line %d\n",yylineno);
        return -1;
    }
    }
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
}
| expr GT {
    if(flg==0){
    getidtype(idname); op1=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op1,typ);}
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
} expr {
if(flg==0){
    getidtype(idname); op2=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op2,typ);
    if(strcmp(op1,op2)!=0){
        printf("Operand are of different types in line %d\n",yylineno);
        return -1;
    }
    }
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
}
| expr LT {
    if(flg==0){
    getidtype(idname); op1=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op1,typ);}
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
} expr {
if(flg==0){
    getidtype(idname); op2=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op2,typ);
    if(strcmp(op1,op2)!=0){
        printf("Operand are of different types in line %d\n",yylineno);
        return -1;
    }
    }
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
```

```
}
| expr ISEQ {
    if(flg==0){
    getidtype(idname); op1=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op1,typ);}
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
} expr {
if(flg==0){
    getidtype(idname); op2=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op2,typ);
    if(strcmp(op1,op2)!=0){
        printf("Operand are of different types in line %d\n",yylineno);
        return -1;
    }
    }
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
}
| expr NEQ {
    if(flg==0){
    getidtype(idname); op1=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op1,typ);}
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
} expr {
if(flg==0){
    getidtype(idname); op2=(char *)malloc(strlen(typ)*sizeof(char));  strcpy(op2,typ);
    if(strcmp(op1,op2)!=0){
        printf("Operand are of different types in line %d\n",yylineno);
        return -1;
    }
    }
    if(flg==2){
        printf("Character can't be operand\n in line %d",yylineno);
        return -1;
    }
}
| LPAREN expr RPAREN
| SUB expr      %prec UMINUS
{
    if(flg!=1){
        printf("Invalid operand in line %d",yylineno);
        return -1;
    }
```

```
}
;
identifier: ID
{
        if(hash_lookup_scope(idname)==0){
               printf("Undeclared Variable: '%s' in line %d\n",idname,yylineno);
               return -1;
        }

}
;
arrid: ID LSQ
{
    if(hash_lookup_scope(idname)==0){
        printf("Undeclared Variable: '%s' in line %d\n",idname,yylineno);
        return -1;
    }
    int hashIndex = hashCode(idname);
    int f=0;
    struct DataItem * temp = hashArray[hashIndex];
    while(temp!=NULL){
        if(strcmp(temp->text,idname)==0){
               int sc=temp->scope;
               if(isScopeValid(sc)==1){
                      if(strcmp(temp->type,"array")!=0){
                             f=1;
                      }
                      else gdim=temp->arrdim;
               }

        }
        temp=temp->next;
    }
    if(f==1){
        printf("Single variable id or function has subscript\n");
        return -1;
    }

return 0;
}
;
declarray:  INT ID LSQ NUM RSQ SEMICOLON
{
    dim = $4;
    if(dim<1){
        printf("Array size less than 1\n");
        return -1;
    }
    insert_hash(idname,"array",getCurrentScope(),yylineno,dim,"NA");
}
;
%%
#include"lex.yy.c"
#include<ctype.h>
int count=0;
int getCurrentScope ()
{
       return scopes[scopesLength - 1];
}
void startNewScope ()
{
   scopes[scopesLength++] = scopeCounter++;
}
void endCurrentScope ()
{
       scopesLength--;
}
int isScopeValid (int scope)
{
   int i=0;
   for (i = scopesLength-1; i>=0; i--)
       if (scopes[i] == scope)
               return 1;
   return 0;
}
int hash_lookup_scope(char * key)
{
int hashIndex = hashCode(key);
if(hashArray[hashIndex]!=NULL){
    struct DataItem * temp = hashArray[hashIndex];
    while(temp!=NULL){
        if(strcmp(temp->text,key)==0){
                int sc=temp->scope;
                if(isScopeValid(sc)==1)
                        return 1;
        }
        temp=temp->next;
    }
    }
return 0;
}
void getidtype(char * key){

        if(hash_lookup_scope(key)==1){
                int hashIndex=hashCode(key);
                struct DataItem * temp = hashArray[hashIndex];
                while(temp!=NULL){
                        if(strcmp(temp->text,key)==0){
                                int sc=temp->scope;
                                if(isScopeValid(sc)==1){

                                        typ=(char *)malloc(sizeof(char)*strlen(temp->type));
                                        strcpy(typ,temp->type);

                                }
                        }
                        temp=temp->next;
                }
        }

}
int hashCode(char* key)              /* Returns unique index for a token */
{
   unsigned int i,hash=7;
   for(i=0;i<strlen(key);++i)
   {
       hash=hash*31+key[i];
   }
   return hash % SIZE;
}
void insert_hash(char* text,char* type,int scope, int lineno,int dim, char* retype)
{
       int len,len2,len3;
       len= strlen(text);
       len2= strlen(type);
       len3 = strlen(retype);

   int hashIndex = hashCode(text);
   if(hashArray[hashIndex]!=NULL){
                struct DataItem * temp = hashArray[hashIndex];
                while(temp!=NULL){
        if(strcmp(temp->text,text)==0){
                int sc=temp->scope;
                if(sc==scope)
                        {printf("Multiple declarations: '%s' in line %d\nVariable already
declared in line %d\n\n",text,lineno,temp->lineno);
                        return;
                        }
        }
        temp=temp->next;
    }
   }
                struct DataItem * new;
       new = (struct DataItem *)malloc(sizeof(struct DataItem));
```

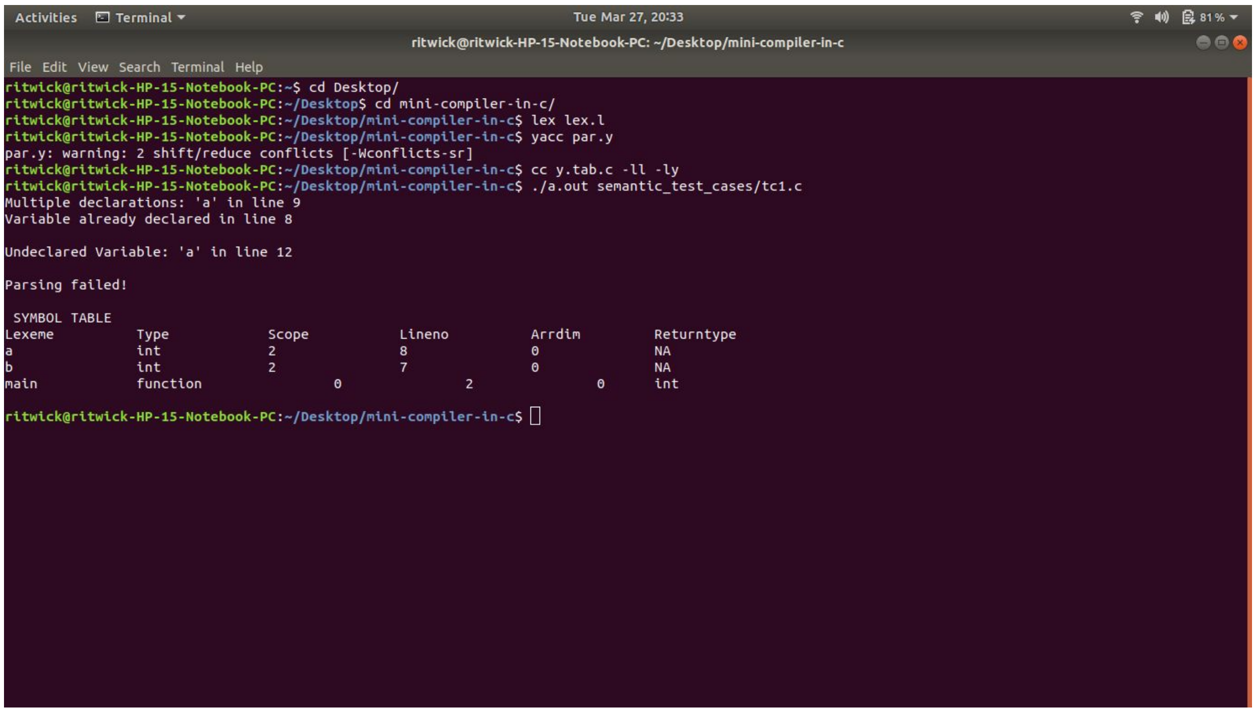
```
        new-> type = (char *)malloc(len2*sizeof(char));
        strcpy(new->type,type);
        new->lineno= lineno;
        new-> text = (char *)malloc(len*sizeof(char));
        strcpy(new->text,text);
        new->retype = (char*)malloc(len3*sizeof(char));
        strcpy(new->retype,retype);
        new->arrdim=dim;
        new->scope=scope;


        new->next= NULL;
    if(hashArray[hashIndex] != NULL)
    {   //chaining
        struct DataItem * head = hashArray[hashIndex];
        new->next=head;
        hashArray[hashIndex]=new;

    }
    else{

    hashArray[hashIndex] = new;
}
}
void display_hash()
{
    int i = 0;
  printf("Lexeme\t\tType\t\tScope\t\tLineno\t\tArrdim\t\tReturntype\n");
    for(i = 0; i<SIZE; i++)
    {
        if(hashArray[i] != NULL){
        struct DataItem * head = hashArray[i];

printf("%s\t\t%s\t\t%d\t\t%d\t\t%d\t\t%s\n",head->text,head->type,head->scope,head->lineno,h
ead->arrdim,head->retype);
        while(head->next!=NULL){
               head=head->next;

printf("%s\t\t%s\t\t%d\t\t%d\t\t%d\t\t%s\n",head->text,head->type,head->scope,head->lineno,h
ead->arrdim,head->retype);
        }
        }
    }

    printf("\n");
}
int main(int argc, char *argv[])
{
        yyin = fopen(argv[1], "r");

    if(!yyparse())
        printf("\nParsing complete!\n");
        else
        printf("\nParsing failed!\n");
        printf("\n SYMBOL TABLE\n");
        display_hash();
        fclose(yyin);
        return 0;
}

void yyerror(char *s) {
        printf("Line %d : %s before '%s'\n", yylineno, s, yytext);
}
```

Test cases and output

tc1.c

```
#include <stdio.h>
int main()
{
      /* code */

      if(1){
            int b;
            int a;
            char a;
      }
      if(2){
      a=5;
      }

}
```

This is to check if variable is undeclared or not.

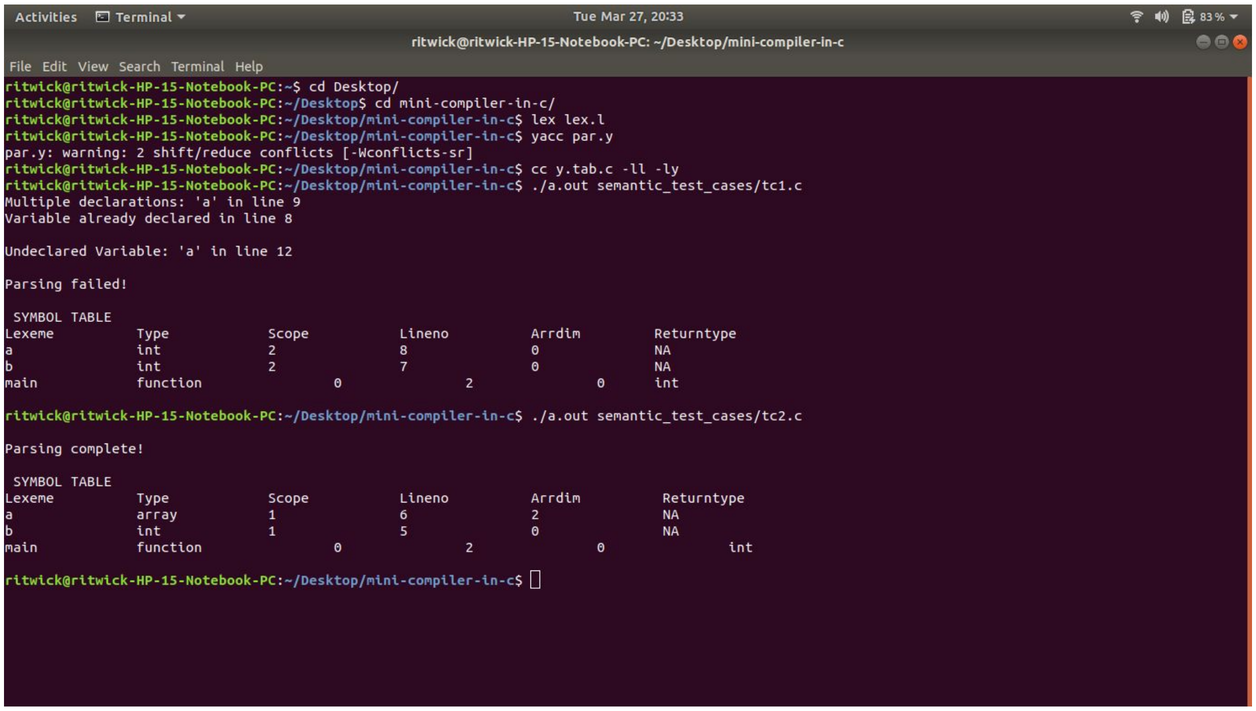

Activities
Terminal
Tue Mar 27, 20:33
81 %
ritwick@ritwick-HP-15-Notebook-PC: ~/Desktop/mini-compiler-in-c
File Edit View Search Terminal Help

```
ritwick@ritwick-HP-15-Notebook-PC:~$ cd Desktop/
ritwick@ritwick-HP-15-Notebook-PC:~/Desktop$ cd mini-compiler-in-c/
ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ lex lex.l
ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ yacc par.y
par.y: warning: 2 shift/reduce conflicts [-Wconflicts-sr]
ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ cc y.tab.c -ll -ly
ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ ./a.out semantic_test_cases/tc1.c
Multiple declarations: 'a' in line 9
Variable already declared in line 8

Undeclared Variable: 'a' in line 12

Parsing failed!

 SYMBOL TABLE
Lexeme          Type            Scope           Lineno          Arrdim          Returntype
a               int             2               8               0               NA
b               int             2               7               0               NA
main            function                0               2               0       int

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ 
```

tc2.c

```c
#include <stdio.h>
int main()
{
	/* code */
	int b;
	int a[2];
	a[2]=6;
b=a+'a';
	//int b[-1];
	return 0;
}
```

This code is used for type checking.

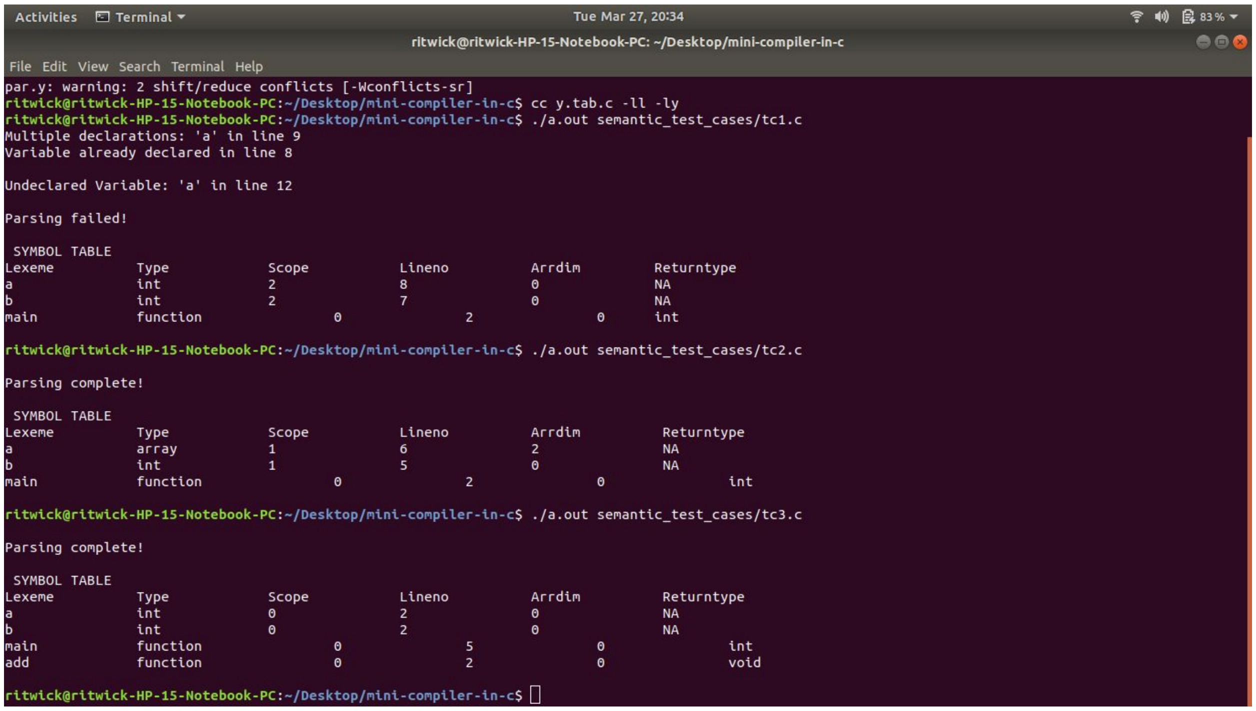


```
ritwick@ritwick-HP-15-Notebook-PC:~$ cd Desktop/
ritwick@ritwick-HP-15-Notebook-PC:~/Desktop$ cd mini-compiler-in-c/
ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ lex lex.l
ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ yacc par.y
par.y: warning: 2 shift/reduce conflicts [-Wconflicts-sr]
ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ cc y.tab.c -ll -ly
ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ ./a.out semantic_test_cases/tc1.c
Multiple declarations: 'a' in line 9
Variable already declared in line 8

Undeclared Variable: 'a' in line 12

Parsing failed!

 SYMBOL TABLE
Lexeme          Type            Scope           Lineno          Arrdim          Returntype
a               int             2               8               0               NA
b               int             2               7               0               NA
main            function                0               2               0       int

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ ./a.out semantic_test_cases/tc2.c

Parsing complete!

 SYMBOL TABLE
Lexeme          Type            Scope           Lineno          Arrdim           Returntype
a               array           1               6               2                NA
b               int             1               5               0                NA
main            function                0               2               0                int

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ 
```

tc3.c

```c
#include <stdio.h>
void add(int a,int b){
        return a+b;
}
int main()
{
        /* code */
        ;
        add(4);
}
```

This is for undefined function.

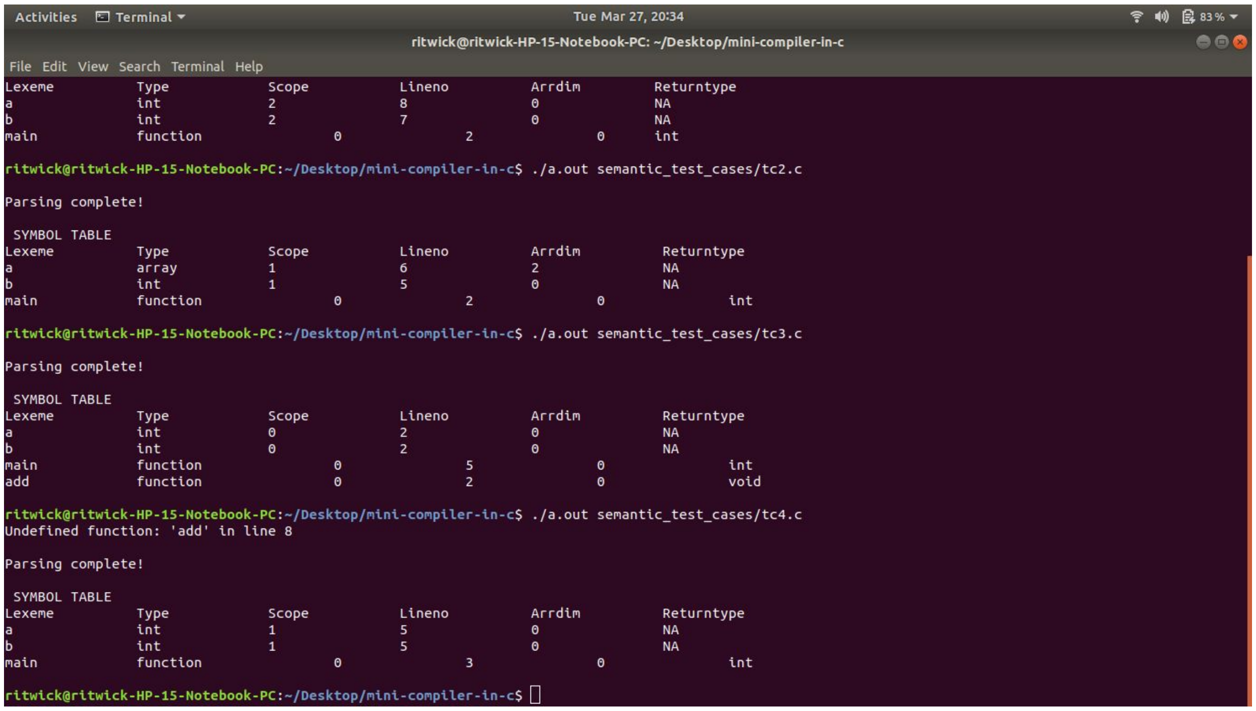

Activities
Terminal
Tue Mar 27, 20:34
83 %
ritwick@ritwick-HP-15-Notebook-PC: ~/Desktop/mini-compiler-in-c
File Edit View Search Terminal Help

```
par.y: warning: 2 shift/reduce conflicts [-Wconflicts-sr]
ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ cc y.tab.c -ll -ly
ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ ./a.out semantic_test_cases/tc1.c
Multiple declarations: 'a' in line 9
Variable already declared in line 8

Undeclared Variable: 'a' in line 12

Parsing failed!

 SYMBOL TABLE
Lexeme          Type            Scope           Lineno          Arrdim         Returntype
a               int             2               8               0              NA
b               int             2               7               0              NA
main            function                0               2               0      int

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ ./a.out semantic_test_cases/tc2.c

Parsing complete!

 SYMBOL TABLE
Lexeme          Type            Scope           Lineno          Arrdim          Returntype
a               array           1               6               2               NA
b               int             1               5               0               NA
main            function                0               2               0               int

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ ./a.out semantic_test_cases/tc3.c

Parsing complete!

 SYMBOL TABLE
Lexeme          Type            Scope           Lineno          Arrdim          Returntype
a               int             0               2               0               NA
b               int             0               2               0               NA
main            function                0               5               0               int
add             function                0               2               0               void

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ 
```

tc4.c

```c
#include <stdio.h>

int main(){
	int a,b;
	a=5;
	b=3;
	add(a,b);
}
```

This is a similar testcase.

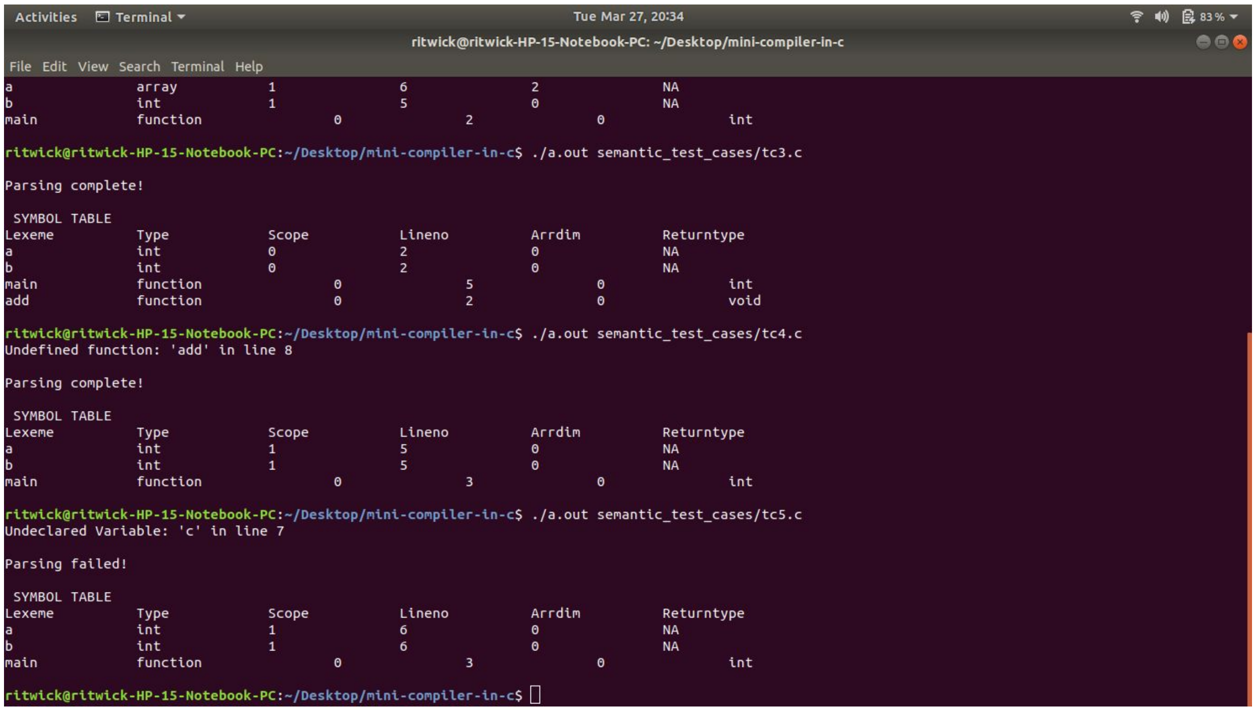

Activities
Terminal
Tue Mar 27, 20:34
83 %
ritwick@ritwick-HP-15-Notebook-PC: ~/Desktop/mini-compiler-in-c
File Edit View Search Terminal Help

```
Lexeme          Type            Scope           Lineno          Arrdim          Returntype
a               int             2               8               0               NA
b               int             2               7               0               NA
main            function                0               2               0       int

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ ./a.out semantic_test_cases/tc2.c

Parsing complete!

 SYMBOL  TABLE
Lexeme          Type            Scope           Lineno          Arrdim          Returntype
a               array           1               6               2               NA
b               int             1               5               0               NA
main            function                0               2               0               int

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ ./a.out semantic_test_cases/tc3.c

Parsing complete!

 SYMBOL  TABLE
Lexeme          Type            Scope           Lineno          Arrdim          Returntype
a               int             0               2               0               NA
b               int             0               2               0               NA
main            function                0               5               0               int
add             function                0               2               0               void

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ ./a.out semantic_test_cases/tc4.c
Undefined function: 'add' in line 8

Parsing complete!

 SYMBOL  TABLE
Lexeme          Type            Scope           Lineno          Arrdim          Returntype
a               int             1               5               0               NA
b               int             1               5               0               NA
main            function                0               3               0               int

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ 
```

tc5.c

```c
#include <stdio.h>

int main()
{
        /* code */
        int a,b;
        a=b+c;
}
```

This shows type checking functionalities.

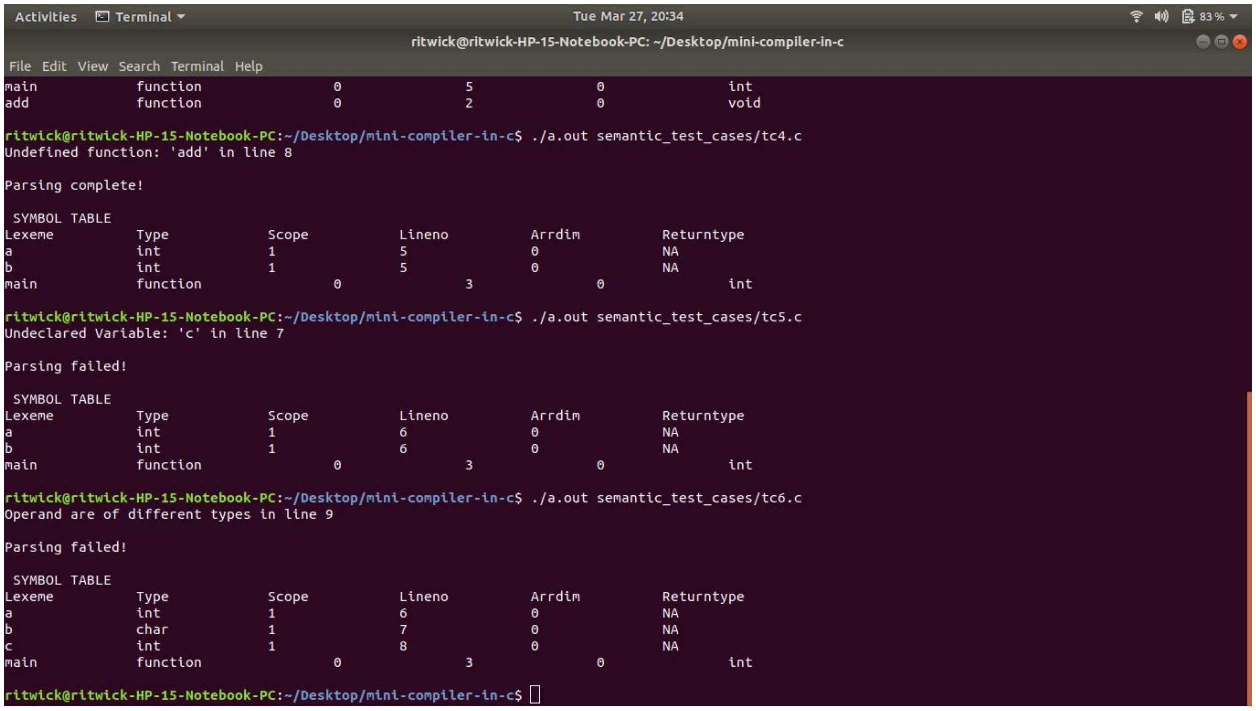

Activities
Terminal
Tue Mar 27, 20:34
83 %
ritwick@ritwick-HP-15-Notebook-PC: ~/Desktop/mini-compiler-in-c
File Edit View Search Terminal Help

```
a               array           1               6               2               NA
b               int             1               5               0               NA
main            function                0               2               0               int

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ ./a.out semantic_test_cases/tc3.c

Parsing complete!

 SYMBOL TABLE
Lexeme          Type            Scope           Lineno          Arrdim          Returntype
a               int             0               2               0               NA
b               int             0               2               0               NA
main            function                0               5               0               int
add             function                0               2               0               void

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ ./a.out semantic_test_cases/tc4.c
Undefined function: 'add' in line 8

Parsing complete!

 SYMBOL TABLE
Lexeme          Type            Scope           Lineno          Arrdim          Returntype
a               int             1               5               0               NA
b               int             1               5               0               NA
main            function                0               3               0               int

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ ./a.out semantic_test_cases/tc5.c
Undeclared Variable: 'c' in line 7

Parsing failed!

 SYMBOL TABLE
Lexeme          Type            Scope           Lineno          Arrdim          Returntype
a               int             1               6               0               NA
b               int             1               6               0               NA
main            function                0               3               0               int

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ 
```

tc6.c

```c
#include <stdio.h>

int main()
{
	/* code */
	int a;
	char b;
	int c;
	c=a+b;
}
```

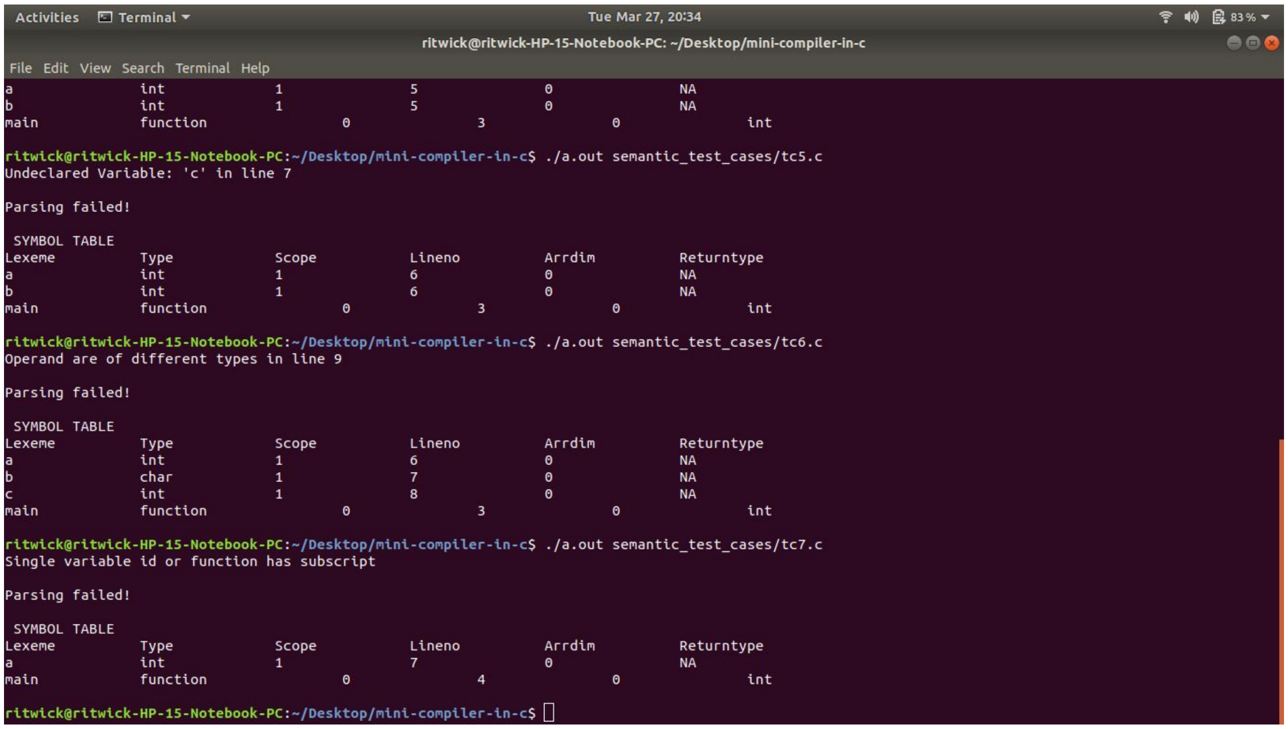
This shows some other type checking features.


Activities
Terminal
Tue Mar 27, 20:34
83 %
ritwick@ritwick-HP-15-Notebook-PC: ~/Desktop/mini-compiler-in-c
File Edit View Search Terminal Help

```
main            function                0               5               0               int
add             function                0               2               0               void

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ ./a.out semantic_test_cases/tc4.c
Undefined function: 'add' in line 8

Parsing complete!

 SYMBOL TABLE
Lexeme          Type            Scope           Lineno          Arrdim          Returntype
a               int             1               5               0               NA
b               int             1               5               0               NA
main            function                0               3               0               int

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ ./a.out semantic_test_cases/tc5.c
Undeclared Variable: 'c' in line 7

Parsing failed!

 SYMBOL TABLE
Lexeme          Type            Scope           Lineno          Arrdim          Returntype
a               int             1               6               0               NA
b               int             1               6               0               NA
main            function                0               3               0               int

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ ./a.out semantic_test_cases/tc6.c
Operand are of different types in line 9

Parsing failed!

 SYMBOL TABLE
Lexeme          Type            Scope           Lineno          Arrdim          Returntype
a               int             1               6               0               NA
b               char            1               7               0               NA
c               int             1               8               0               NA
main            function                0               3               0               int

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$
```

tc7.c

```
//single variable has subscript
#include <stdio.h>

int main()
{
        /* code */
        int a;
        if(a[2])
                {;}
}
```


```
a               int             1               5               0               NA
b               int             1               5               0               NA
main            function                0               3               0               int

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ ./a.out semantic_test_cases/tc5.c
Undeclared Variable: 'c' in line 7

Parsing failed!

 SYMBOL TABLE
Lexeme          Type            Scope           Lineno          Arrdim          Returntype
a               int             1               6               0               NA
b               int             1               6               0               NA
main            function                0               3               0               int

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ ./a.out semantic_test_cases/tc6.c
Operand are of different types in line 9

Parsing failed!

 SYMBOL TABLE
Lexeme          Type            Scope           Lineno          Arrdim          Returntype
a               int             1               6               0               NA
b               char            1               7               0               NA
c               int             1               8               0               NA
main            function                0               3               0               int

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ ./a.out semantic_test_cases/tc7.c
Single variable id or function has subscript

Parsing failed!

 SYMBOL TABLE
Lexeme          Type            Scope           Lineno          Arrdim          Returntype
a               int             1               7               0               NA
main            function                0               4               0               int

ritwick@ritwick-HP-15-Notebook-PC:~/Desktop/mini-compiler-in-c$ 
```

This is if a single variable has subscript. It throws the corresponding error message.