CS347 Project Report
Group 4

30th March 2017

Members

Chirag Soni	Laxman Prabhakar	Alan Aipe
1401CS13	1401CS22	1401CS50

Index

Source code

c-parser.lex

```
D			[0-9]
L			[a-zA-Z_]
H			[a-fA-F0-9]
E			([Ee][+-]?{D}+)
P                       ([Pp][+-]?{D}+)
FS			(f|F|l|L)
IS                      ((u|U)|(u|U)?(l|L|ll|LL)|(l|L|ll|LL)(u|U))

%{
#include <stdio.h>
#include "y.tab.h"

void count(void);
void comment(void);
int check_type(void);
%}

%%
"/*"			{ comment(); }
"//"[^\n]*              { /* consume //-comment */ }


"auto"			{ count(); return(AUTO); }
"_Bool"			{ count(); return(BOOL); }
"break"			{ count(); return(BREAK); }
"case"			{ count(); return(CASE); }
"char"			{ count(); return(CHAR); }
"_Complex"		{ count(); return(COMPLEX); }
"const"			{ count(); return(CONST); }
"continue"		{ count(); return(CONTINUE); }
"default"		{ count(); return(DEFAULT); }
"do"			{ count(); return(DO); }
"double"		{ count(); return(DOUBLE); }
"else"			{ count(); return(ELSE); }
"enum"			{ count(); return(ENUM); }
"extern"		{ count(); return(EXTERN); }
"float"			{ count(); return(FLOAT); }
"for"			{ count(); return(FOR); }
"goto"			{ count(); return(GOTO); }
"if"			{ count(); return(IF); }
```

```
"_Imaginary"       { count(); return(IMAGINARY); }
"inline"           { count(); return(INLINE); }
"int"              { count(); return(INT); }
"long"             { count(); return(LONG); }
"register"         { count(); return(REGISTER); }
"restrict"         { count(); return(RESTRICT); }
"return"           { count(); return(RETURN); }
"short"            { count(); return(SHORT); }
"signed"           { count(); return(SIGNED); }
"sizeof"           { count(); return(SIZEOF); }
"static"           { count(); return(STATIC); }
"struct"           { count(); return(STRUCT); }
"switch"           { count(); return(SWITCH); }
"typedef"          { count(); return(TYPEDEF); }
"union"            { count(); return(UNION); }
"unsigned"         { count(); return(UNSIGNED); }
"void"             { count(); return(VOID); }
"volatile"         { count(); return(VOLATILE); }
"while"            { count(); return(WHILE); }

{L}({L}|{D})*      { count(); return(check_type()); }

0[xX]{H}+{IS}?          { count(); return(CONSTANT); }
0[0-7]*{IS}?       { count(); return(CONSTANT); }
[1-9]{D}*{IS}?          { count(); return(CONSTANT); }
L?'(\\.|[^\\'\n])+'     { count(); return(CONSTANT); }

{D}+{E}{FS}?       { count(); return(CONSTANT); }
{D}*"."{D}+{E}?{FS}?     { count(); return(CONSTANT); }
{D}+"."{D}*{E}?{FS}?     { count(); return(CONSTANT); }
0[xX]{H}+{P}{FS}?    { count(); return(CONSTANT); }
0[xX]{H}*"."{H}+{P}?{FS}?   { count(); return(CONSTANT); }
0[xX]{H}+"."{H}*{P}?{FS}?   { count(); return(CONSTANT); }


L?\"(\\.|[^\\"\n])*\"    { count(); return(STRING_LITERAL); }

"..."              { count(); return(ELLIPSIS); }
">>="              { count(); return(RIGHT_ASSIGN); }
"<<="              { count(); return(LEFT_ASSIGN); }
"+="               { count(); return(ADD_ASSIGN); }
"-="               { count(); return(SUB_ASSIGN); }
"*="               { count(); return(MUL_ASSIGN); }
"/="               { count(); return(DIV_ASSIGN); }
"%="               { count(); return(MOD_ASSIGN); }
"&="               { count(); return(AND_ASSIGN); }
"^="               { count(); return(XOR_ASSIGN); }
```

```
"|="                { count(); return(OR_ASSIGN); }
">>"                { count(); return(RIGHT_OP); }
"<<"                { count(); return(LEFT_OP); }
"++"                { count(); return(INC_OP); }
"--"                { count(); return(DEC_OP); }
"->"                { count(); return(PTR_OP); }
"&&"                { count(); return(AND_OP); }
"||"                { count(); return(OR_OP); }
"<="                { count(); return(LE_OP); }
">="                { count(); return(GE_OP); }
"=="                { count(); return(EQ_OP); }
"!="                { count(); return(NE_OP); }
";"                 { count(); return(';'); }
("{"|"<%")          { count(); return('{'); }
("}"|"%>")          { count(); return('}'); }
","                 { count(); return(','); }
":"                 { count(); return(':'); }
"="                 { count(); return('='); }
"("                 { count(); return('('); }
")"                 { count(); return(')'); }
("["|"<:")          { count(); return('['); }
("]"|":>")          { count(); return(']'); }
"."                 { count(); return('.'); }
"&"                 { count(); return('&'); }
"!"                 { count(); return('!'); }
"~"                 { count(); return('~'); }
"-"                 { count(); return('-'); }
"+"                 { count(); return('+'); }
"*"                 { count(); return('*'); }
"/"                 { count(); return('/'); }
"%"                 { count(); return('%'); }
"<"                 { count(); return('<'); }
">"                 { count(); return('>'); }
"^"                 { count(); return('^'); }
"|"                 { count(); return('|'); }
"?"                 { count(); return('?'); }

[ \t\v\f]           { count(); }
"\n" {count();}
.           { /* Add code to complain about unmatched characters */ }

%%

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

```
void comment(void)
{
    // Takes care of matching comment characters
    char c, prev = 0;

    while ((c = input()) != 0)   /* (EOF maps to 0) */
    {
      if (c == '/' && prev == '*')
           return;
      prev = c;
    }
    printf("unterminated comment");
}


int column = 0;
int line=0;
int previous_column=0;
int update_prev=1;
void count(void)
{


    if(update_prev==1){
      previous_column=column;
    }

    if(strcmp(yytext,"\n")==0||strcmp(yytext,"\t")==0){
      update_prev=0;
    }
    else{
      update_prev=1;
    }

    int i;
    //printf("Entered count\n");
    for (i = 0; yytext[i] != '\0' ; i++){
      if (yytext[i] == '\n'){
           column = 0;
           line++;
      }
      else if (yytext[i] == '\t')
           column +=4;
      else
           column++;
}
//printf("Previous count   %d  column %d  yytext
%s\n",previous_column,column,yytext);
}


int check_type(void)
{
    //code to check for Identifier type if needed
    return IDENTIFIER;
}
```

c-parser.yacc

```
%{
   //Declarations
    #include <stdio.h>
    extern char yytext[];
    extern int previous_column,column,line;
    int printed=0;
    void yyerror(char const *s);
    int yylex();
%}
%error-verbose
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%token TYPEDEF EXTERN STATIC AUTO REGISTER INLINE RESTRICT
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST
VOLATILE VOID
%token BOOL COMPLEX IMAGINARY
%token STRUCT UNION ENUM ELLIPSIS

%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK
RETURN

%start translation_unit
%%

primary_expression
    : IDENTIFIER
    | CONSTANT
```

```
	| STRING_LITERAL
	| '(' expression ')'
	;

postfix_expression
	: primary_expression
	| postfix_expression '[' expression ']'
	| postfix_expression '(' ')'
	| postfix_expression '(' argument_expression_list ')'
	| postfix_expression '.' IDENTIFIER
	| postfix_expression PTR_OP IDENTIFIER
	| postfix_expression INC_OP
	| postfix_expression DEC_OP
	| '(' type_name ')' '{' initializer_list '}'
	| '(' type_name ')' '{' initializer_list ',' '}'
	;

argument_expression_list
	: assignment_expression
	| argument_expression_list ',' assignment_expression
	;

unary_expression
	: postfix_expression
	| INC_OP unary_expression
	| DEC_OP unary_expression
	| unary_operator cast_expression
	| SIZEOF unary_expression
	| SIZEOF '(' type_name ')'
	;

unary_operator
	: '&'
	| '*'
	| '+'
	| '-'
	| '~'
	| '!'
	;

cast_expression
	: unary_expression
	| '(' type_name ')' cast_expression
	;

multiplicative_expression
	: cast_expression
```

```
    | multiplicative_expression '*' cast_expression
    | multiplicative_expression '/' cast_expression
    | multiplicative_expression '%' cast_expression
    ;

additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;

shift_expression
    : additive_expression
    | shift_expression LEFT_OP additive_expression
    | shift_expression RIGHT_OP additive_expression
    ;

relational_expression
    : shift_expression
    | relational_expression '<' shift_expression
    | relational_expression '>' shift_expression
    | relational_expression LE_OP shift_expression
    | relational_expression GE_OP shift_expression
    ;

equality_expression
    : relational_expression
    | equality_expression EQ_OP relational_expression
    | equality_expression NE_OP relational_expression
    ;

and_expression
    : equality_expression
    | and_expression '&' equality_expression
    ;

exclusive_or_expression
    : and_expression
    | exclusive_or_expression '^' and_expression
    ;

inclusive_or_expression
    : exclusive_or_expression
    | inclusive_or_expression '|' exclusive_or_expression
    ;

logical_and_expression
```

```
    : inclusive_or_expression
    | logical_and_expression AND_OP inclusive_or_expression
    ;

logical_or_expression
    : logical_and_expression
    | logical_or_expression OR_OP logical_and_expression
    ;

conditional_expression
    : logical_or_expression
    | logical_or_expression '?' expression ':' conditional_expression
    ;

assignment_expression
    : conditional_expression
    | unary_expression assignment_operator assignment_expression
    ;
if_expression
    : logical_or_expression
    ;

assignment_operator
    : '='
    | MUL_ASSIGN
    | DIV_ASSIGN
    | MOD_ASSIGN
    | ADD_ASSIGN
    | SUB_ASSIGN
    | LEFT_ASSIGN
    | RIGHT_ASSIGN
    | AND_ASSIGN
    | XOR_ASSIGN
    | OR_ASSIGN
    ;

expression
    : assignment_expression
    | expression ',' assignment_expression
    ;

constant_expression
    : conditional_expression
    ;

declaration
    : declaration_specifiers ';'
    | declaration_specifiers init_declarator_list ';'
    ;

declaration_specifiers
    : storage_class_specifier
    | storage_class_specifier declaration_specifiers
    | type_specifier
    | type_specifier declaration_specifiers
    | type_qualifier
    | type_qualifier declaration_specifiers
    | function_specifier
    | function_specifier declaration_specifiers
    ;

init_declarator_list
    : init_declarator
    | init_declarator_list ',' init_declarator
    ;

init_declarator
    : declarator
    | declarator '=' initializer
    ;

storage_class_specifier
    : TYPEDEF
    | EXTERN
    | STATIC
    | AUTO
    | REGISTER
    ;

type_specifier
    : VOID
    | CHAR
    | SHORT
    | INT
    | LONG
    | FLOAT
    | DOUBLE
    | SIGNED
    | UNSIGNED
    | BOOL
    | COMPLEX
    | IMAGINARY
    | struct_or_union_specifier
    | enum_specifier
    | TYPE_NAME
    ;

struct_or_union_specifier
    : struct_or_union IDENTIFIER '{' struct_declaration_list '}'
    | struct_or_union '{' struct_declaration_list '}'
    | struct_or_union IDENTIFIER
    ;

struct_or_union
    : STRUCT
    | UNION
    ;

struct_declaration_list
    : struct_declaration
    | struct_declaration_list struct_declaration
    ;

struct_declaration
    : specifier_qualifier_list struct_declarator_list ';'
    ;

specifier_qualifier_list
    : type_specifier specifier_qualifier_list
    | type_specifier
    | type_qualifier specifier_qualifier_list
    | type_qualifier
    ;

struct_declarator_list
    : struct_declarator
    | struct_declarator_list ',' struct_declarator
    ;

struct_declarator
    : declarator
    | ':' constant_expression
    | declarator ':' constant_expression
    ;

enum_specifier
    : ENUM '{' enumerator_list '}'
    | ENUM IDENTIFIER '{' enumerator_list '}'
    | ENUM '{' enumerator_list ',' '}'
    | ENUM IDENTIFIER '{' enumerator_list ',' '}'
    | ENUM IDENTIFIER
    ;

enumerator_list
    : enumerator
    | enumerator_list ',' enumerator
    ;

enumerator
    : IDENTIFIER
    | IDENTIFIER '=' constant_expression
    ;

type_qualifier
    : CONST
    | RESTRICT
    | VOLATILE
    ;

function_specifier
    : INLINE
    ;

declarator
    : pointer direct_declarator
    | direct_declarator
    ;


direct_declarator
    : IDENTIFIER
    | '(' declarator ')'
    | direct_declarator '[' type_qualifier_list assignment_expression
']'
    | direct_declarator '[' type_qualifier_list ']'
    | direct_declarator '[' assignment_expression ']'
    | direct_declarator '[' STATIC type_qualifier_list
assignment_expression ']'
    | direct_declarator '[' type_qualifier_list STATIC
assignment_expression ']'
    | direct_declarator '[' type_qualifier_list '*' ']'
    | direct_declarator '[' '*' ']'
    | direct_declarator '[' ']'
    | direct_declarator '(' parameter_type_list ')'
    | direct_declarator '(' identifier_list ')'
    | direct_declarator '(' ')'
    ;
```

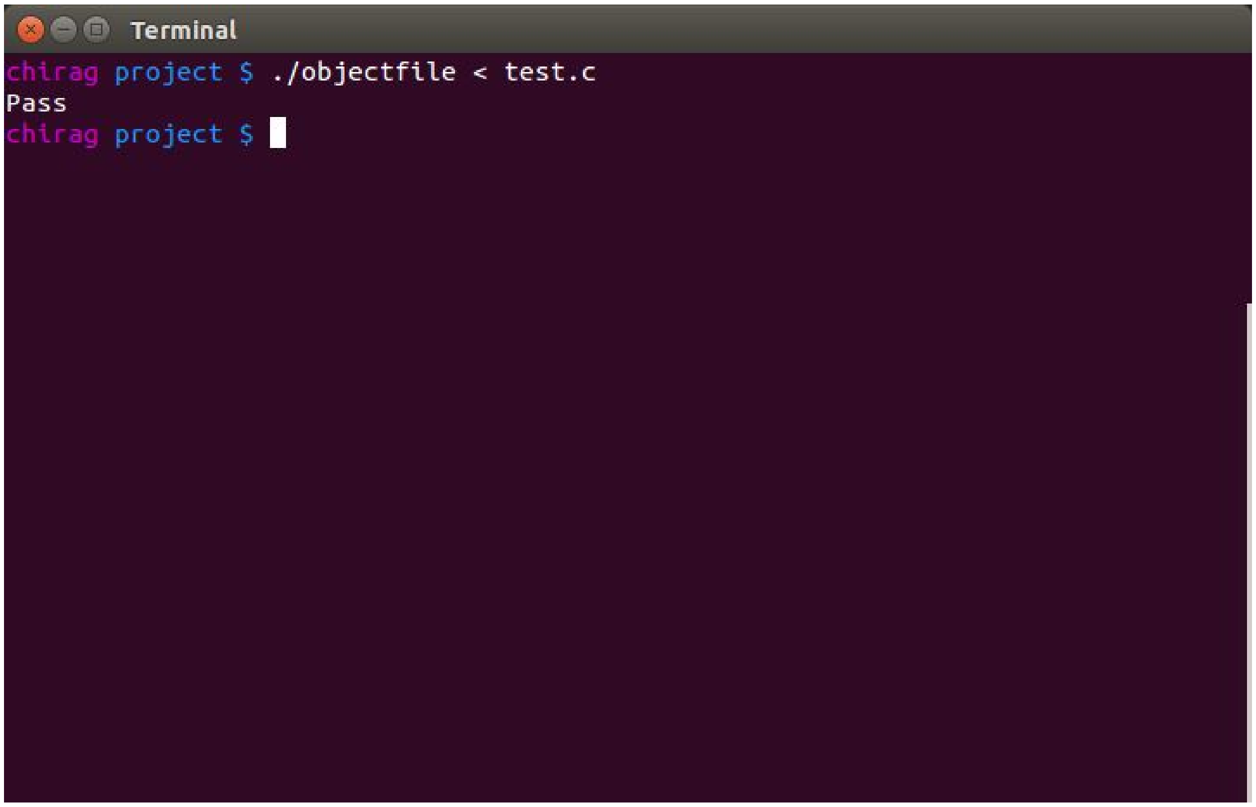
```
pointer
    : '*'
    | '*' type_qualifier_list
    | '*' pointer
    | '*' type_qualifier_list pointer
    ;

type_qualifier_list
    : type_qualifier
    | type_qualifier_list type_qualifier
    ;


parameter_type_list
    : parameter_list
    | parameter_list ',' ELLIPSIS
    ;

parameter_list
    : parameter_declaration
    | parameter_list ',' parameter_declaration
    ;

parameter_declaration
    : declaration_specifiers declarator
    | declaration_specifiers abstract_declarator
    | declaration_specifiers
    ;

identifier_list
    : IDENTIFIER
    | identifier_list ',' IDENTIFIER
    ;

type_name
    : specifier_qualifier_list
    | specifier_qualifier_list abstract_declarator
    ;

abstract_declarator
    : pointer
    | direct_abstract_declarator
    | pointer direct_abstract_declarator
    ;

direct_abstract_declarator
    : '(' abstract_declarator ')'
    | '[' ']'
    | '[' assignment_expression ']'
    | direct_abstract_declarator '[' ']'
    | direct_abstract_declarator '[' assignment_expression ']'
    | '[' '*' ']'
    | direct_abstract_declarator '[' '*' ']'
    | '(' ')'
    | '(' parameter_type_list ')'
    | direct_abstract_declarator '(' ')'
    | direct_abstract_declarator '(' parameter_type_list ')'
    ;

initializer
    : assignment_expression
    | '{' initializer_list '}'
    | '{' initializer_list ',' '}'
    ;

initializer_list
    : initializer
    | designation initializer
    | initializer_list ',' initializer
    | initializer_list ',' designation initializer
    ;

designation
    : designator_list '='
    ;

designator_list
    : designator
    | designator_list designator
    ;

designator
    : '[' constant_expression ']'
    | '.' IDENTIFIER
    ;

statement
    : labeled_statement
    | compound_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    ;
```

```
labeled_statement
    : IDENTIFIER ':' statement
    | CASE constant_expression ':' statement
    | DEFAULT ':' statement
    ;

compound_statement
    : '{' '}'
    | '{' block_item_list '}'
    ;

block_item_list
    : block_item
    | block_item_list block_item
    ;

block_item
    : declaration
    | statement
    ;

expression_statement
    : ';'
    | expression ';'
    | error '\n'

    // |{printf("Error in line %d column %d : ",
line,previous_column);
    //     printf("\"; not found \"\n");
    //     printed=1;}
    // | expression {printf("Error in line %d column %d : ",
line,previous_column);
    //     printf("\"; not found \"\n");
    //     printed=1;}
    ;

selection_statement
    : IF '(' if_expression ')' statement
    | IF '(' if_expression ')' statement ELSE statement
    | SWITCH '(' if_expression ')' statement
    | error ';'
    ;

iteration_statement
    : WHILE '(' expression ')' statement
    | DO statement WHILE '(' expression ')' ';'
    | FOR '(' expression_statement expression_statement ')' statement
    | FOR '(' expression_statement expression_statement expression
')' statement
    | FOR '(' declaration expression_statement ')' statement
    | FOR '(' declaration expression_statement expression ')'
statement
    | error ';'
    ;

jump_statement
    : GOTO IDENTIFIER ';'
    | CONTINUE ';'
    | BREAK ';'
    | RETURN ';'
    | RETURN expression ';'

    ;

translation_unit
    : external_declaration
{if(printed==0)printf("Pass\n");printed=1;}
    | translation_unit external_declaration
{if(printed==0)printf("Pass\n");printed=1;}
    ;

external_declaration
    : function_definition
    | declaration
    ;

function_definition
    : declaration_specifiers declarator declaration_list
compound_statement
    | declaration_specifiers declarator compound_statement
    ;

declaration_list
    : declaration
    | declaration_list declaration
    ;

%%

void yyerror(char const *s)
{
    //fflush(stdout);
    printf("Error in line %d column %d : ", line,previous_column);
    printf("\"%s\"\n",s);
    printed=1;

}

int main (void) {
      yyparse ( );
      return 0;
}
```

Test scripts

test.c

```
int main(){
    srand(time(NULL));
    printf("10\n");
    int i,j;
    for(i = 0;i < 10;i++){
      printf("1000 ");
      for(j = 0;j < 1000;j++){
           printf("%d ",rand()%1500 + 1);
      }
    }
}
```

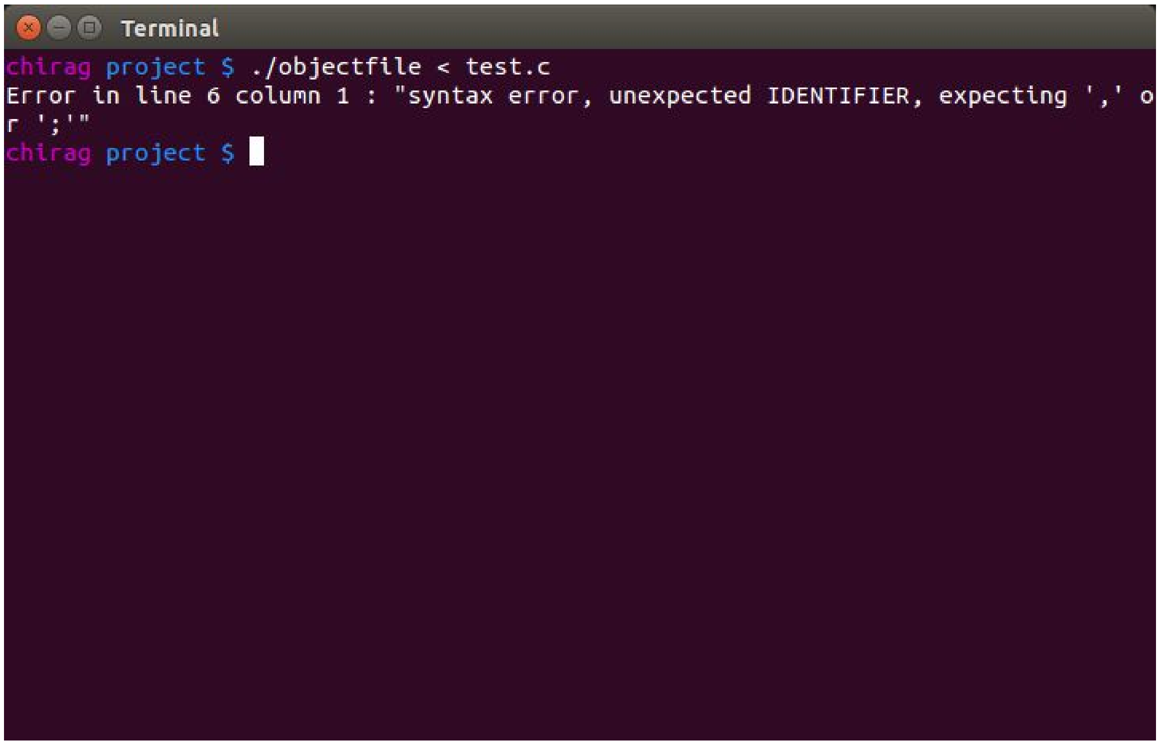


```
chirag project $ ./objectfile < test.c
Pass
chirag project $ 
```

Fig : Terminal output for above test case - test1.c

test2.c

```
int main(){
struct s{
int a;
char *name;
}tmp,*p;
5
tmp.name = "Hello";
tmp.a = 10;
p = &tmp;

printf("%s\n",tmp.name);
printf("%s\n",(*p).name);
printf("%c\n",*(p->name));
return 0;
}
```



```
chirag project $ ./objectfile < test.c
Error in line 6 column 1 : "syntax error, unexpected IDENTIFIER, expecting ',' o
r ';'"
chirag project $
```

Fig - Terminal output for above test case - test2.c

Contributions

- Chirag Soni :
  - Writing grammar for c-parser
  - Conflict resolving (10 out of 29 conflicts)
  - Report Writing

- Laxman Prabhakar :
  - Conflict resolving (19 out of 29 conflicts)
  - Commenting
  - Report Writing

- Alan Aipe :
  - Writing lex file
  - Enhancing error detection
  - Report Writing