

1 INTRODUCTION:

1.1 PARSER:

A parser is one of the components in an interpreter or compiler, which checks for correct syntax and builds a data structure (often some kind of parse tree, abstract syntax tree or other hierarchical structure) implicit in the input tokens. The parser often uses a separate lexical analyzer to create tokens from the sequence of input characters. Parsers may be programmed by hand or may be (semi-)automatically generated (in some programming languages) by a tool (such as Yacc).

1.2 ROLE OF A PARSER:

The parser is expected to report any syntactic errors in the code in an intelligible fashion and to recover from commonly occurring errors and continue the execution of the remainder of the program. The parser obtains the set of tokens identified by the lexical analyzer and verifies that the string of tokens can be generated by the grammar for the source language.

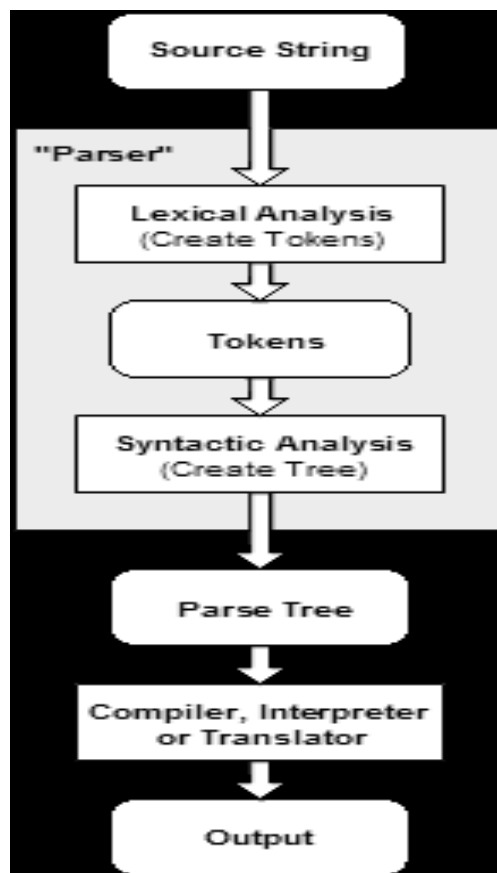


Fig 1.2

1.3 TYPES OF PARSER:

There are three general types of parsers: universal, top-down and bottom-up.

Universal parsers can parse any type of grammar. There are, however, very difficult to implement and too inefficient to use in the production of a compiler.

Top-down builds parse trees from the root to the leaves, as the name suggests and bottom-up parsers construct parse trees from the leaves and goes to the root.

1.3.1 TOP-DOWN PARSING:

A recursive descent parser is a top-down parser built from a set of mutually-recursive procedures (or a non-recursive equivalent) where each such procedure usually implements one of the production rules of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes.

A predictive parser is a recursive descent parser that does not require backtracking. Predictive parsing is possible only for the class of LL (k) grammars, which are the context-free grammars for which there exists some positive integer k that allows a recursive descent parser to decide which production to use by examining only the next k tokens of input. (The LL (k) grammars therefore exclude all ambiguous grammars, as well as all grammars that contain left recursion. Any context-free grammar can be transformed into an equivalent grammar that has no left recursion, but removal of left recursion does not always yield an LL (k) grammar.) A predictive parser runs in linear time.

Recursive descent with backup is a technique that determines which production to use by trying each production in turn. Recursive descent with backup is not limited to LL (k) grammars, but is not guaranteed to terminate unless the grammar is LL (k). Even when they terminate, parsers that use recursive descent with backup may require exponential time.

Although predictive parsers are widely used, programmers often prefer to create LR or LALR parsers via parser generators without transforming the grammar into LL (k) form.

1.3.2 BOTTOM-UP PARSING:

The Parsing method in which the Parse tree is constructed from the input language string beginning from the leaves and going up to the root node.

Bottom-Up parsing is also called shift-reduce parsing due to its implementation.

The YACC supports shift-reduce parsing.

Example: Suppose there is a grammar G having a production E

$E \rightarrow E * E$

and an input string $x * y$.

The left hand sides of any production are called Handles. Thus the handle for this example is E. The shift action is simply pushing an input symbol on a stack. When the R.H.S of a production is matched the stack elements are popped and replaced by the corresponding Handle. This is the reduce action.

Thus in the above example, the parser shifts the input token 'x' onto the stack. Then again it shifts the token '*' on the top of the stack. Still the production is not satisfied so it shifts the next token 'y' too. Now the production E is matched so it pops all the three tokens from the stack and replaces it with the handle 'E'. Any action that is specified with the rule is carried out.

If the input string reaches the end of file /line and no error has occurred then the parser executes the 'Accept' action signifying successful completion of parsing. Otherwise it executes an 'Error' action.

The use of stack in shift-reduce parsing can be justified by the fact that the handle will always eventually appear on the top of the stack and never inside.

1.4 CONFLICTS DURING SHIFT-REDUCE PARSING:

There are context-free grammars for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents and the next input symbol, cannot decide whether to shift or to reduce (Shift-reduce conflict) or cannot decide which of several reductions to make (Reduce-Reduce conflict).

2 DESIGN GOALS:

To implement a parser at that performed the following operations:

- Identify the correctness of declaring the main () function in C.
- To validate the declarations of identifiers of different data types such as int, char, float and double. The parser must identify dynamic initialization (assigning appropriate values to variables at the time of their initialization).
- Assignment operations must be validated i.e. statements such as,
 a==10; (wrong operator)
 And
 b=10 (no semi-colon)
must generate errors.
- Input and output statements which are printf and scanf respectively must be checked for syntactical correctness. The correct syntax for them are as follows:
 printf("<message to be displayed> ",<identifier names>);
 And
 scanf("<order of values to be input> ",<<&>identifier names>);
Therefore any statements of the following forms must generate errors:
 printf("this is a statement);
 Or
 printf("message""message1");
 Or
 scanf("%d,&n);
 Or
 scanf("%d",n);
- Arithmetic expressions must also be checked for. The syntax must be:
 <id> = <expression>;
The expression uses the addition, subtraction, multiplication and division operators and curved parenthesis.
- Conditional statements such as if, if-else and nested if-else statements must also be identified and tested. The basic syntax for if-else statements is shown below:
 if(<condition>)
 {
 <if-body>
 }
 else{ <else-body> }

- Looping constructs such as for loops, while and do-while loops are also identified and checked for correctness. Nesting of one or more looping constructs and conditional statements is permitted. Their syntax is mentioned below:

For loop:

```
for(<initialization>;<looping condition>;<increment/decrement>)  
{  
    <for-body>  
}
```

While loop:

```
while(<looping condition>)  
{  
    <while-body>  
}
```

Do-while loop:

```
do  
{  
    <do-while body>  
}  
while(<looping condition>);
```

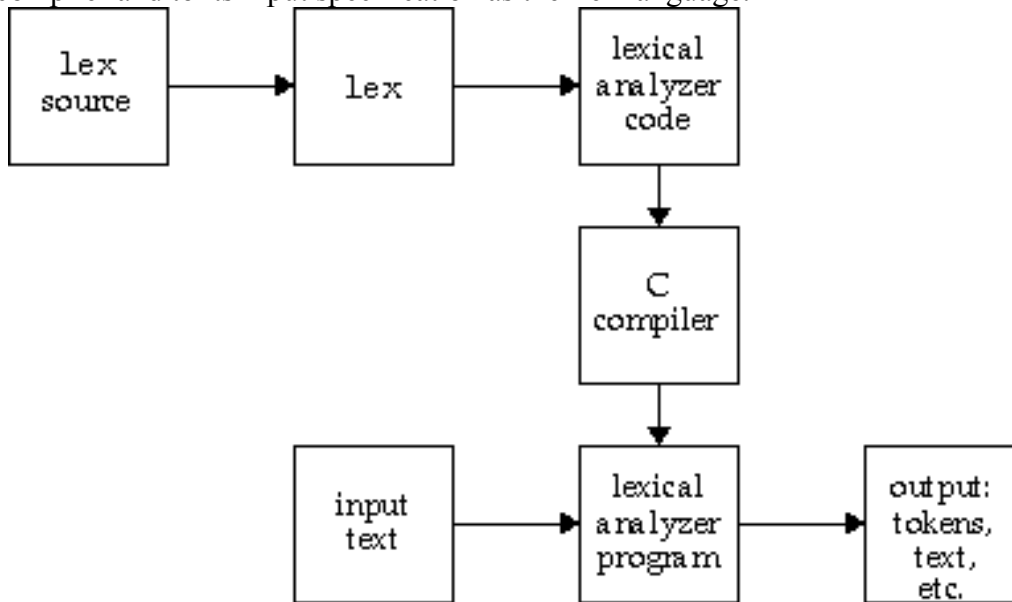
- The parenthesis {...} must also be checked for and errors must be generated if found unbalanced.

3 DESIGN TOOLS:

The Lex utility generates a 'C' code which is nothing but a `yylex ()` function which can be used as an interface to YACC. Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions.

The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages." Lex is widely used tool to specify lexical analyzers for a variety of languages. We refer to the tool as the Lex compiler and to its input specification as the Lex language.



3.1 STRUCTURE OF A LEX PROGRAM:

The general structure of a lex program is as follows:

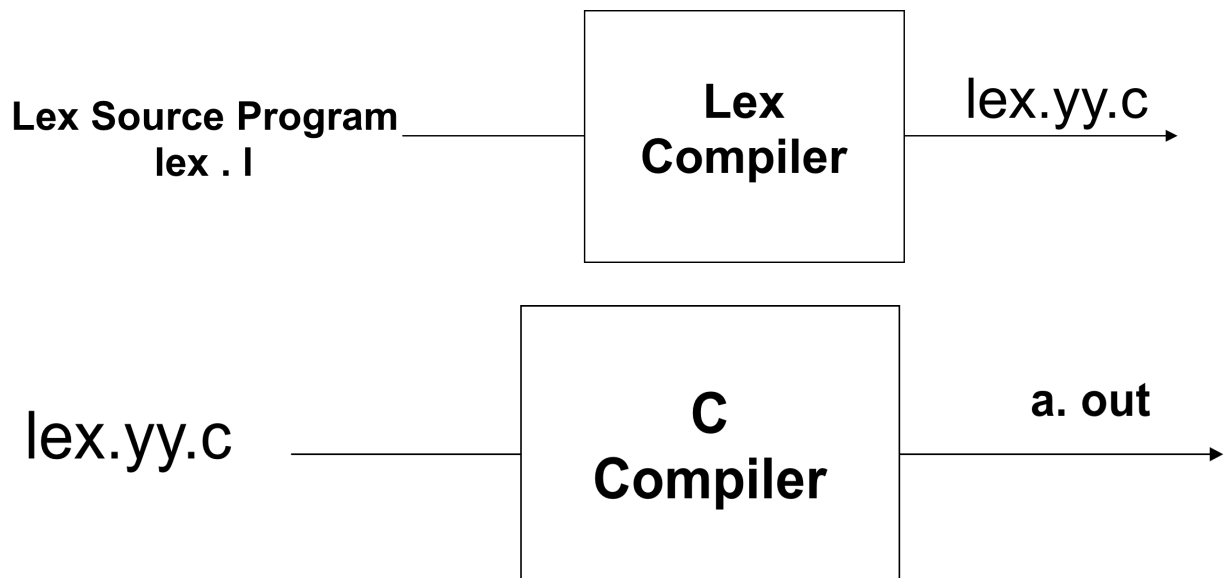
```

<definitions>
%%
<rules>
%%
<user subroutines>

```

Definitions consist of any external 'C' definitions used in the lex actions or subroutines. E.g. all preprocessor directives like `#include`, `#define` macros etc. These are simply copied to the `lex.yy.c` file. The other types of definitions are Lex definitions which are essentially the lex substitution strings, lex start states and lex table size declarations. The Rules is the basic part which specifies

the regular expressions and their corresponding actions. The User Subroutines are the function definitions of the functions that are used in the Lex actions.



3.2 COMPILATION OF A LEX PROGRAM:

1. First a specification of a lexical analyzer is prepared by creating a program lex.l in the lex language.
2. lex.l is run through the lex compiler to produce a C program lex.yy.c.
3. Finally, lex.yy.c is run through the C compiler to produce an object program a.out.

3.3 FORMAT OF THE RULES SECTION:

x	the character "x"
"x"	an "x", even if x is an operator.
\x	an "x", even if x is an operator.
[xy]	the character x or y.
[x-z]	the characters x, y, ... or z.
[^x]	any character but x.
.	any character but newline.
^x	an x at the beginning of a line.
<y>x	an x when Lex is in start condition y.
x\$	an x at the end of a line.
x?	an optional x.
x*	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x y	an x or a y.

(x) an x.
x/y an x but only if followed by y.
{xx} the translation of xx from the definitions section.
x{m,n} m through n occurrences of x

3.4 YACC (YET ANOTHER COMPILER COMPILER):

Yacc provides a general tool for describing the input to a computer program.

The user can specify his input in terms of individual input characters or in terms of higher level constructs such as names and numbers.

User prepares a specification of the input process; this includes:

- rules describing the input structure,
- code to be invoked when these rules are recognized,
- And a low-level routine to do the basic input.

Yacc then generates a function to control the input process.

This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream.

A structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal symbol.

To avoid confusion, terminal symbols will usually be referred to as tokens.

Yacc is the Utility which generates the function 'yyparse' which is indeed the Parser. Yacc describes a context free, LALR (1) grammar and supports both bottom-up and top-down parsing. The general format for the YACC file is very similar to that of the Lex file.

1. Declarations
2. Grammar Rules
3. Subroutines

In Declarations apart from the legal 'C' declarations there are few Yacc specific declarations which begin with a %sign.

1. %union: It defines the Stack type for the Parser.
 It is a union of various data/structures/objects.
2. %token: These are the terminals returned by the yylex
 function to the yacc. A token can also have type
 associated with it for good type checking and
 syntax directed translation. A type of a token
 can be specified as %token <stack member>
 tokenName.

3. %type: The type of a non-terminal symbol in the Grammar rule can be specified with this. The format is %type <stack member> non-terminal.
4. %noassoc: Specifies that there is no associativity of a terminal symbol.
5. %left: Specifies the left associativity of a Terminal Symbol
6. %right: Specifies the right associativity of a Terminal Symbol.
7. %start: Specifies the L.H.S non-terminal symbol of a production rule which should be taken as the starting point of the grammar rules.
8. %prec: Changes the precedence level associated with a particular rule to that of the following token name or literal.
The grammar rules are specified as follows:
Context-free grammar production-
$$p \rightarrow AbC$$

Yacc Rule-
$$p : A b C \{ /* 'C' actions */ \}$$

The general style for coding the rules is to have all Terminals in upper-case and all non-terminals in lower-case.

To facilitate a proper syntax directed translation the Yacc has something called pseudo-variables which forms a bridge between the values of terminal/non-terminals and the actions. These pseudo variables are \$\$,\$1,\$2,\$3..... The \$\$ is the L.H.S value of the rule whereas \$1 is the first R.H.S value of the rule and so is \$2 etc. The default type for pseudo variables is integer unless they are specified by %type , %token <type> etc.

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called yylex. The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the ``# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. The end marker must have token number 0 or negative.

Yacc turns the specification file into a C program, which parses the input according to the specification given.

- The parser produced by Yacc consists of a finite state machine with a stack.
- The parser is also capable of reading and remembering the next input token (called the lookahead token).
- The current state is always the one on the top of the stack.

The machine has only four actions available to it, called shift, reduce, accept, and error.

A move of the parser is done as follows:

- Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls **yylex** to obtain the next token.
- Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

3.5 AMBIGUITY AND CONFLICTS:

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule

$$expr : expr \text{ '-' } expr$$

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

$$expr - expr - expr$$

the rule allows this input to be structured as either

$$(expr - expr) - expr \quad (\text{left association})$$

or as

$$expr - (expr - expr) \quad (\text{right association}).$$

It is instructive to consider the problem that confronts the parser given an input such as:

$$expr - expr - expr$$

When the parser has read the second *expr*, the input that it has seen:

$$expr - expr$$

Matches the right side of the grammar rule above. The parser could reduce the input by applying this rule; after applying the rule, the input is reduced to *expr* (the left side of the rule).

The parser would then read the final part of the input:

$$- expr$$

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

$$expr - expr$$

it could defer the immediate application of the rule, and continue reading the input until it had seen

$expr - expr - expr$

It could then apply the rule to the rightmost three symbols, reducing them to $expr$ and leaving

$expr - expr$

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

$expr - expr$

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a shift / reduce conflict.

It may also happen that the parser has a choice of two legal reductions; this is called a reduce / reduce conflict.

Yacc invokes two disambiguating rules by default:

- In a shift/reduce conflict, the default is to do the shift.
- In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts.

Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

3.6 EXECUTION OF A YACC PROGRAM:

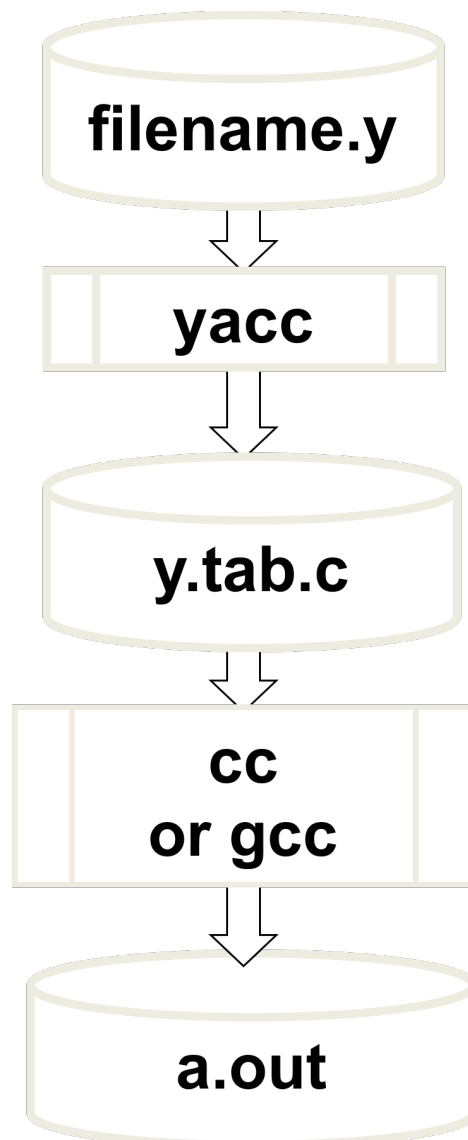


Fig 3.6.1

The Yacc programs can be executed in two ways.

The yacc program itself will have c-code which passes the tokens.

The program has to convert the typed number to digit and pass the number to the yacc program. Then the yacc program can be executed by giving the command:

```
yacc -d <filename.y>
```

The output of this execution results in y.tab.c file. This file can be compiled to get the executable file. The compilation is as follows:

```
cc y.tab.c lex.yy.c -ll
```

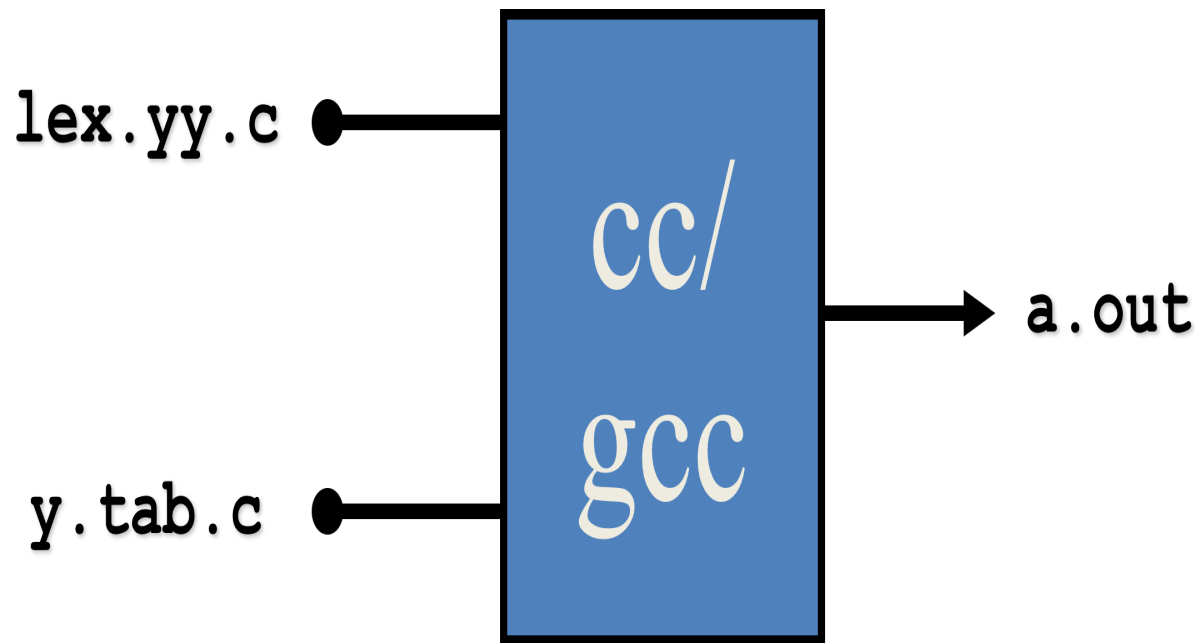


Fig 3.6.2

4 FUTURE ENHANCEMENTS:

- The parser must be capable of parsing the entire source code. The implemented parser exits on encountering the first error.
- Implementation of header files and other pre-processor directives.
- Validating the identifiers. Identifiers that are not declared must not be used.
- Displaying the line number of the errors that occur along with appropriate messages on how to correct them. Error handling and recovery strategies can be implemented
- Identifying the number of format specifiers and matching the data types with the variable type in printf and scanf statements.
- Validation of pointer variables, their declarations, usage and operations.
- Validating function declarations, parameter matching, function call and recursions.
- Declaration of global variables, constant variable and macro variables.

5 BIBLIOGRAPHY:

- Compilers: Principles, Techniques and Tools by Jeffrey D. Ullaman, Alfred V. Aho, Monica S. Lam and Ravi Sethi.
- System Software by Leland L. Beck and D. Manjula.
- Lex and yacc by John R. Levine, Tony Mason and Doug Brown.
- The Complete Reference to C by Hebert Schildt.
- www.google.com

6 APPENDIX-A (code)

parser.l:

```
%{
#include "y.tab.h"
}%

%%

"main"      {printf("%s",yytext);return MAIN;}
"+"         {printf("%s",yytext);return PLUS;}
"="         {printf("%s",yytext);return EQU;}
"_"         {printf("%s",yytext);return MIN;}
"*"         {printf("%s",yytext);return STAR;}
"/"         {printf("%s",yytext);return SLASH;}
"%/"        {printf("%s",yytext);return MOD;}
"!"         {printf("%s",yytext);return NOT;}
"<"         {printf("%s",yytext);return LT;}
">"         {printf("%s",yytext);return GT;}
"<="        {printf("%s",yytext);return LE;}
">="        {printf("%s",yytext);return GE;}
"=="        {printf("%s",yytext);return DEQU;}
"\""        {printf("%s",yytext);return SQT;}
"("         {printf("%s",yytext);return OB;}
"["         {printf("%s",yytext);return OSB;}
"]"         {printf("%s",yytext);return CSB;}
"#"         {printf("%s",yytext);return HASH;}
"include"   {printf("%s",yytext);return INCLUDE;}
"\"         {printf("%s",yytext);return DOT;}
")"         {printf("%s",yytext);return CB;}
"{"         {printf("%s",yytext);return OCB;}
"}"         {printf("%s",yytext);return CCB;}
":"         {printf("%s",yytext);return COL;}
"."         {printf("%s",yytext);return SEMI;}
"return"    {printf("%s",yytext);return RET;}
"int"       {printf("%s",yytext);return INT;}
"float"     {printf("%s",yytext);return FLOAT;}
"char"      {printf("%s",yytext);return CHAR;}
"&"         {printf("%s",yytext);return AMP;}
"%d"        {printf("%s",yytext);return MODD;}
"%i"        {printf("%s",yytext);return MODF;}
"%s"        {printf("%s",yytext);return MODS;}
","         {printf("%s",yytext);return COM;}
"\"         {printf("%s",yytext);return DQT;}
"if"        {printf("%s",yytext);return IF;}
```

```
"else"      {printf("%s",yytext);return ELSE;}
"for"       {printf("%s",yytext);return FOR;}
"while"     {printf("%s",yytext);return WHILE;}
"do"        {printf("%s",yytext);return DO;}
"switch"    {printf("%s",yytext);return SWTC;}
"break"     {printf("%s",yytext);return BRK;}
"case"      {printf("%s",yytext);return CASE;}
"default"   {printf("%s",yytext);return DEFA;}
"void"      {printf("%s",yytext);return VOID;}
"printf"    {printf("%s",yytext);return PF;}
"scanf"     {printf("%s",yytext);return SF;}
\n          {printf("\n");}
[ \t]*      {printf("\t");}
[A-Za-z]*   {printf("%s",yytext);return CHR;}
[0-9]*      {printf("%s",yytext);return DIG;}
.           {return yytext[0];}
%%
```

Parser.y:

```
%{
#include<stdio.h>
int flag=0;
extern FILE *yyin;
%}

%token OB CB OSB CSB OCB CCB SEMI INT VOID PF SF MAIN
      FLOAT CHAR PLUS MIN STAR SLASH
      MOD NOT LT GT LE GE DEQU SQT CHR DIG HASH INCLUDE DOT
      EQU AMP MODD MODF MODS COM
      IF ELSE FOR WHILE DO SWTC BRK CASE DEFA EXP DQT COL

%%
T: A {flag=1;printf("\n\n the entered c program is valid \n\n\n\n"); return 0;}
A: HASH INCLUDE LT CHR DOT CHR GT A | S ;
S:   type MAIN OB CB OCB stmt1 ret CCB ;
ret:  RET DIG SEMI | ;
type: VOID | INT | ;
decl: type1 ptr CHR ext stmt1 ;
ptr:  STAR | ;
ext:  OSB DIG CSB ext | COM ptr CHR ext | SEMI | assin ;
assin: EQU dig ext1 ;
ext1:  ext ;
dig:  DIG |DIG DOT DIG exp2 | ;
type1: INT | FLOAT | CHAR ;
```



```
stmt1: OCB stmt1 CCB | decl | io | exp | branch | loop ;
io:    print | read ;
exp:   exp1 SEMI anythin ;

exp1:  CHR EQU exp2 | exp2 ;
exp2:  exp3 sign exp2 | OB exp2 CB exp2 | exp4 | exp3 ;
exp4 : exp3 sign1 | sign1 exp3 ;
sign1: add | sub ;
exp3:  CHR | dig ;
sign:  PLUS | EQU | MIN | STAR | SLASH | LT | GT | LE | GE | DEQU | ob ;
ob:    OB arg1 CB ;
arg1:  CHR COM arg1 | exp3 ;
print: PF OB DQT string DQT paral CB SEMI anythin ;
string: CHR ;
paral: COM CHR paral | ;
read:  SF OB DQT format DQT params CB SEMI anythin ;
format: mod format | ;
mod:   MODD | MODF | MODS ;
params: COM AMP CHR params | ;
branch: iff | switchh ;
iff:   IF OB cond CB inif els ;
inif:  anythin ;
els:   ELSE anythin | ;
switchh: SWTC OB CHR CB OCB swstmts CCB ;
swstmts: CASE SQT CHR SQT COL stmt1 BRK SEMI swstmts | DEFA COL print | ;
loop:   forlp | whilelp ;

forlp:   FOR OB ass SEMI cond SEMI incr CB infor ;
infor:   OCB anythin CCB | anythin ;
anythin: stmt1 | ;
ass:     CHR EQU DIG ;
whilelp: WHILE OB cond CB inwhile ;
inwhile: OCB anythin CCB | anythin ;
cond:    CHR sign num ;
num:     CHR | DIG ;
incr:    CHR add | CHR sub ;
add:     PLUS PLUS ;
sub:     MIN MIN ;

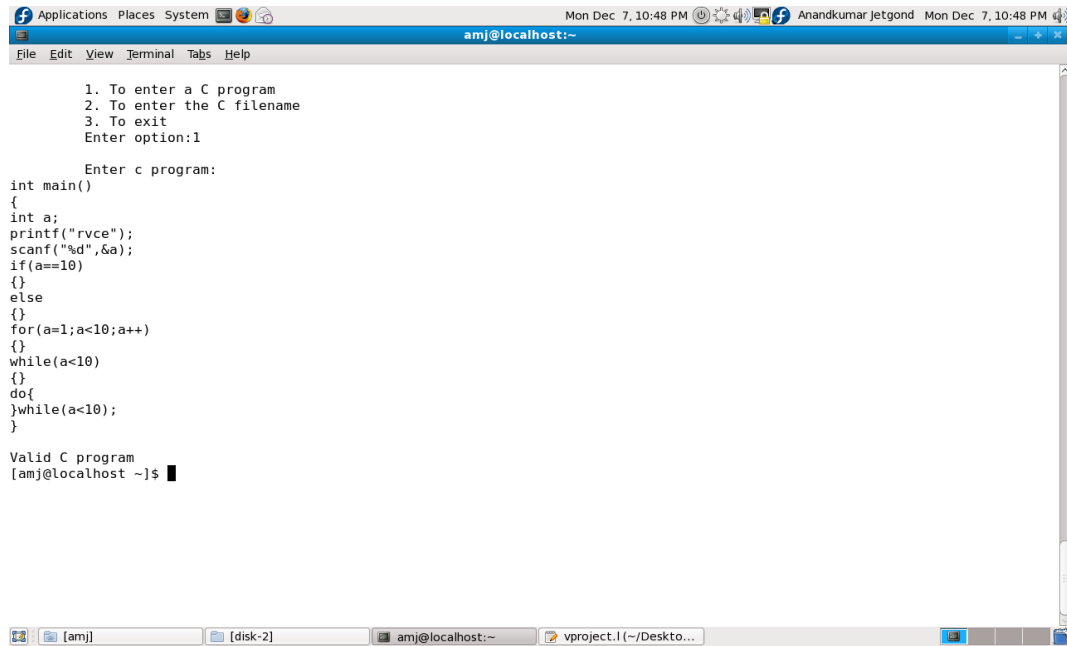
%%
int main()
{
FILE *fp;
int ch;
char fname[20];
```

```
printf("\n enter 1 : to read program from a file \n 2 : exit\n");
scanf("%d",&ch);
switch(ch){
    case 1 :
        printf("\n enter the file name\n ");
        scanf("%s",fname);
        fp = fopen(fname,"r");
        yyin = fp;
        yyparse();
        fclose(fp);
        break;
    case 2 : exit(0);
}
return 0;
}

int yyerror()
{
    if(flag==0)
        printf("\n invalid prog \n ");
    return 0;
}
```

7 APPENDIX-B (snapshots)

- Valid C program:



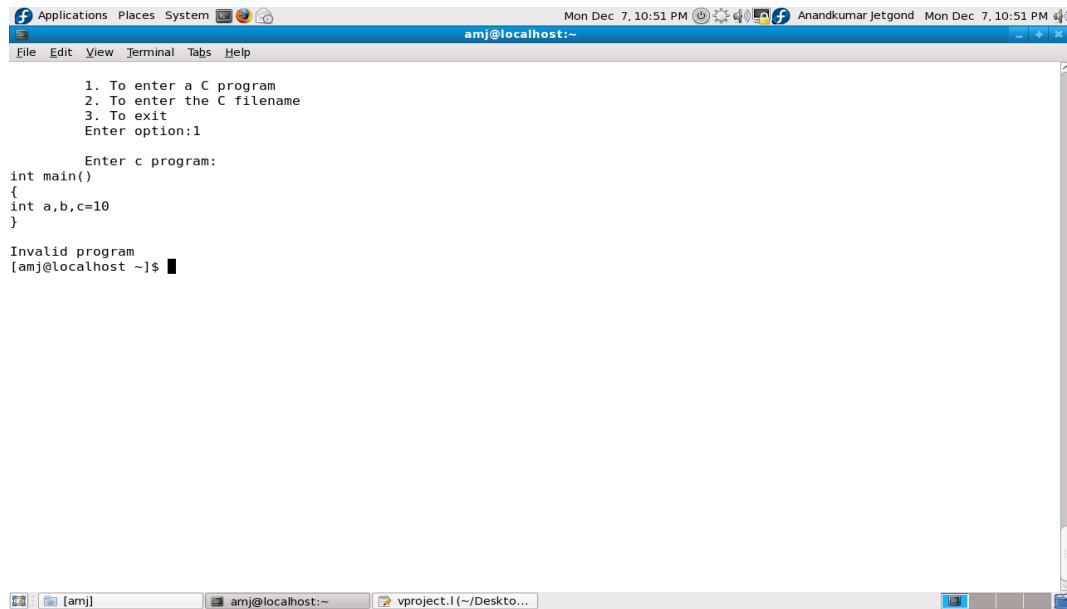
The screenshot shows a terminal window titled 'amj@localhost:~'. The window contains the following text:

```
1. To enter a C program
2. To enter the C filename
3. To exit
Enter option:1
Enter c program:
int main()
{
    int a;
    printf("rvce");
    scanf("%d",&a);
    if(a==10)
    {}
    else
    {}
    for(a=1;a<10;a++)
    {}
    while(a<10)
    {}
    do{
    }while(a<10);
}

Valid C program
[amj@localhost ~]$
```

The terminal window has a menu bar with 'File', 'Edit', 'View', 'Terminal', 'Tabs', and 'Help'. The status bar at the bottom shows the current directory as '~' and the prompt as '\$'.

- Invalid declaration:



The screenshot shows a terminal window titled 'amj@localhost:~'. The window contains the following text:

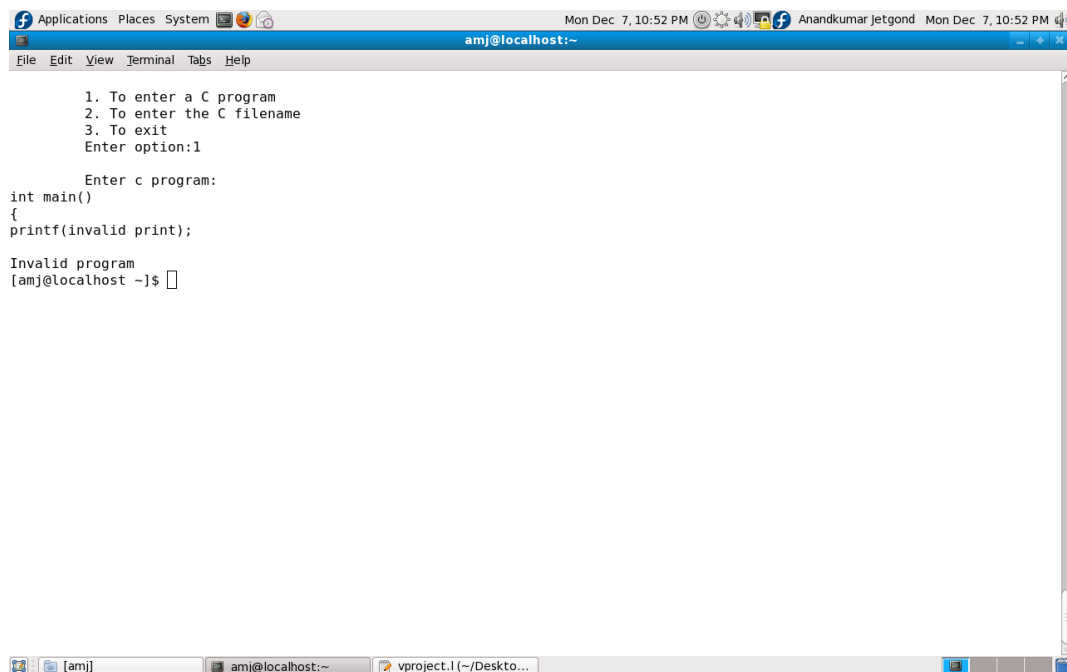
```
1. To enter a C program
2. To enter the C filename
3. To exit
Enter option:1

Enter c program:
int main()
{
int a,b,c=10
}

Invalid program
[amj@localhost ~]$
```

The terminal window has a menu bar with 'File', 'Edit', 'View', 'Terminal', 'Tabs', and 'Help'. The status bar at the bottom shows the current directory as '~' and the prompt as '[amj@localhost ~]\$'.

- Invalid printf statement (no quotes):



The screenshot shows a terminal window titled 'amj@localhost:~'. The window contains the following text:

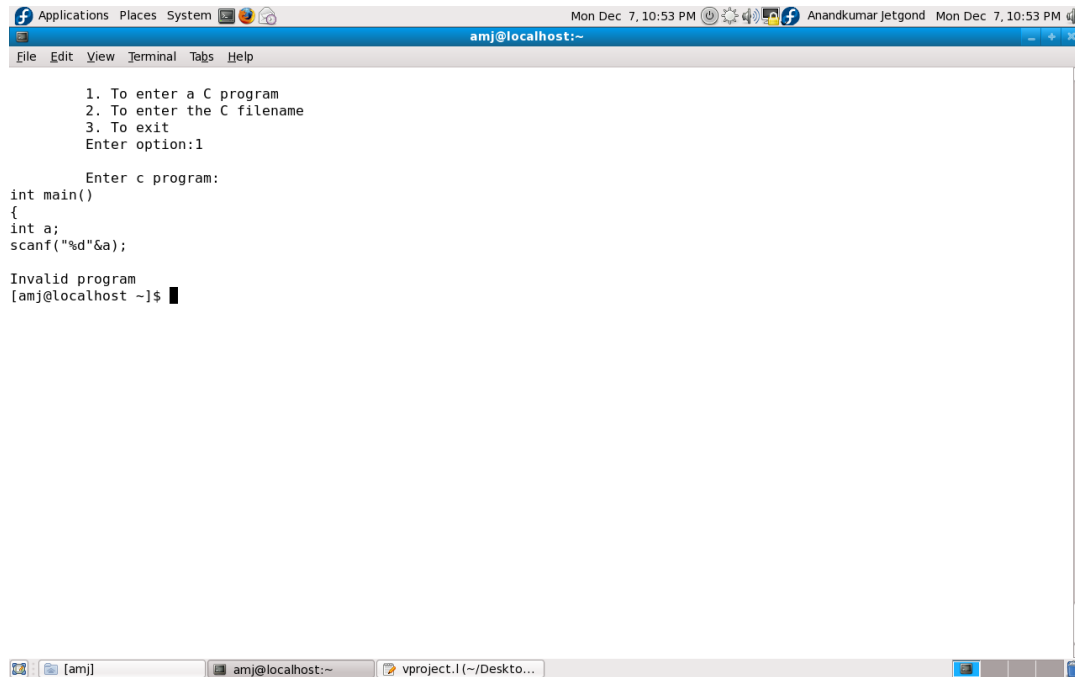
```
1. To enter a C program
2. To enter the C filename
3. To exit
Enter option:1

Enter c program:
int main()
{
printf(invalid print);
}

Invalid program
[amj@localhost ~]$
```

The terminal window has a menu bar with 'File', 'Edit', 'View', 'Terminal', 'Tabs', and 'Help'. The status bar at the bottom shows the current directory as '~' and the prompt as '[amj@localhost ~]\$'.

- Invalid scanf statement (comma missing):



The screenshot shows a terminal window titled 'amj@localhost:~'. The window contains the following text:

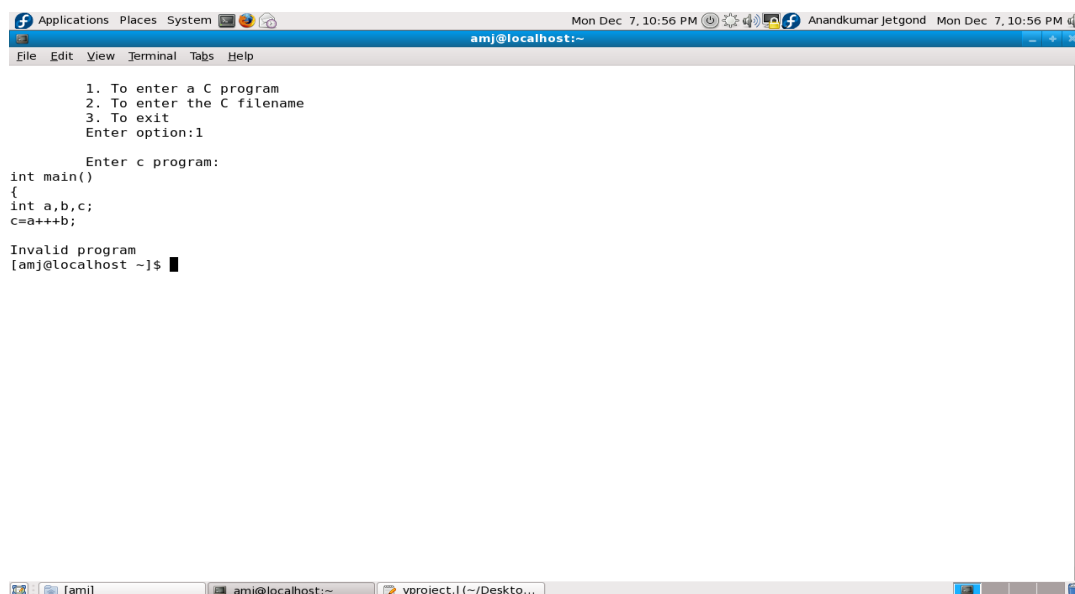
```
1. To enter a C program
2. To enter the C filename
3. To exit
Enter option:1

Enter c program:
int main()
{
int a;
scanf("%d"&a);

Invalid program
[amj@localhost ~]$
```

The terminal window has a menu bar with 'File', 'Edit', 'View', 'Terminal', 'Tabs', and 'Help'. The status bar at the bottom shows the current directory as '~' and the prompt as '[amj@localhost ~]\$'.

- Invalid expression (Wrong use of the operators):



The screenshot shows a terminal window titled 'amj@localhost:~'. The window contains the following text:

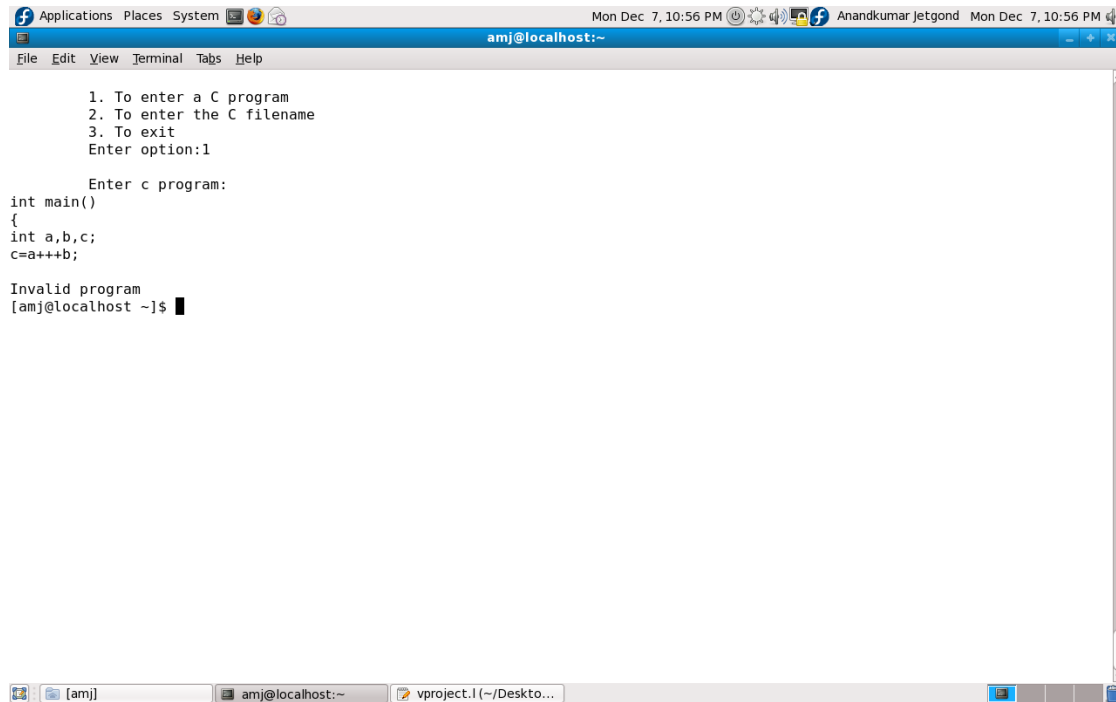
```
1. To enter a C program
2. To enter the C filename
3. To exit
Enter option:1

Enter c program:
int main()
{
int a,b,c;
c=a+++b;

Invalid program
[amj@localhost ~]$
```

The terminal window has a menu bar with 'File', 'Edit', 'View', 'Terminal', 'Tabs', and 'Help'. The status bar at the bottom shows the current directory as '~' and the prompt as '[amj@localhost ~]\$'.

- Invalid if statement (no condition specified):



The screenshot shows a terminal window titled 'amj@localhost:~'. The window contains the following text:

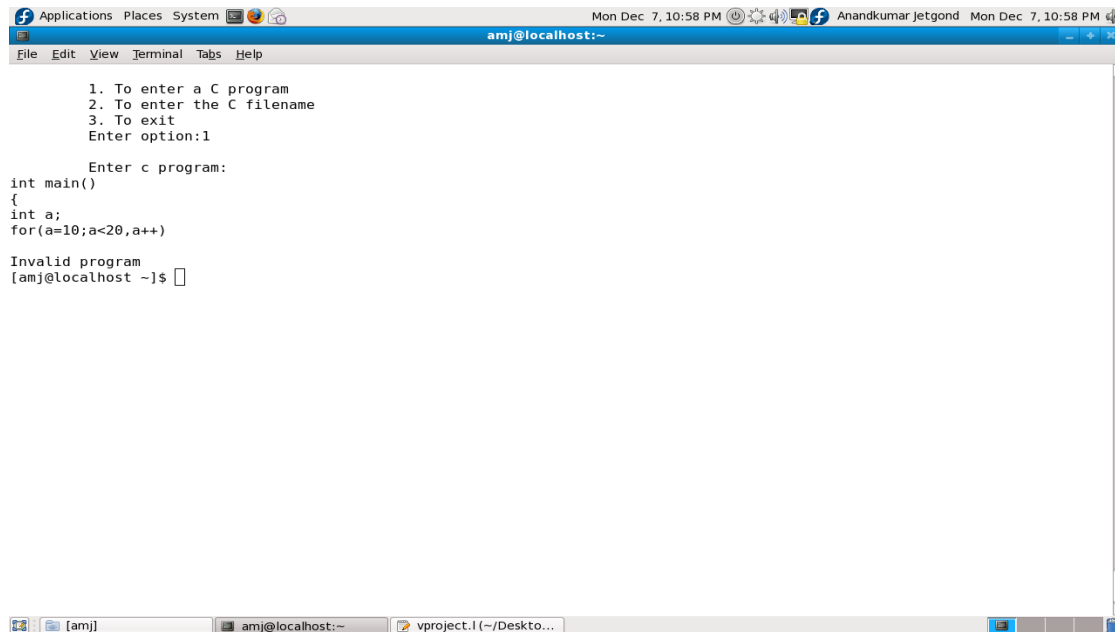
```
1. To enter a C program
2. To enter the C filename
3. To exit
Enter option:1

Enter c program:
int main()
{
int a,b,c;
c=a+++b;

Invalid program
[amj@localhost ~]$
```

The terminal window has a menu bar with 'File', 'Edit', 'View', 'Terminal', 'Tabs', and 'Help'. The status bar at the bottom shows three tabs: '[amj]', 'amj@localhost:~', and 'vproject.1 (~-/Deskto...').

- Invalid for statement (missing semi-colon):

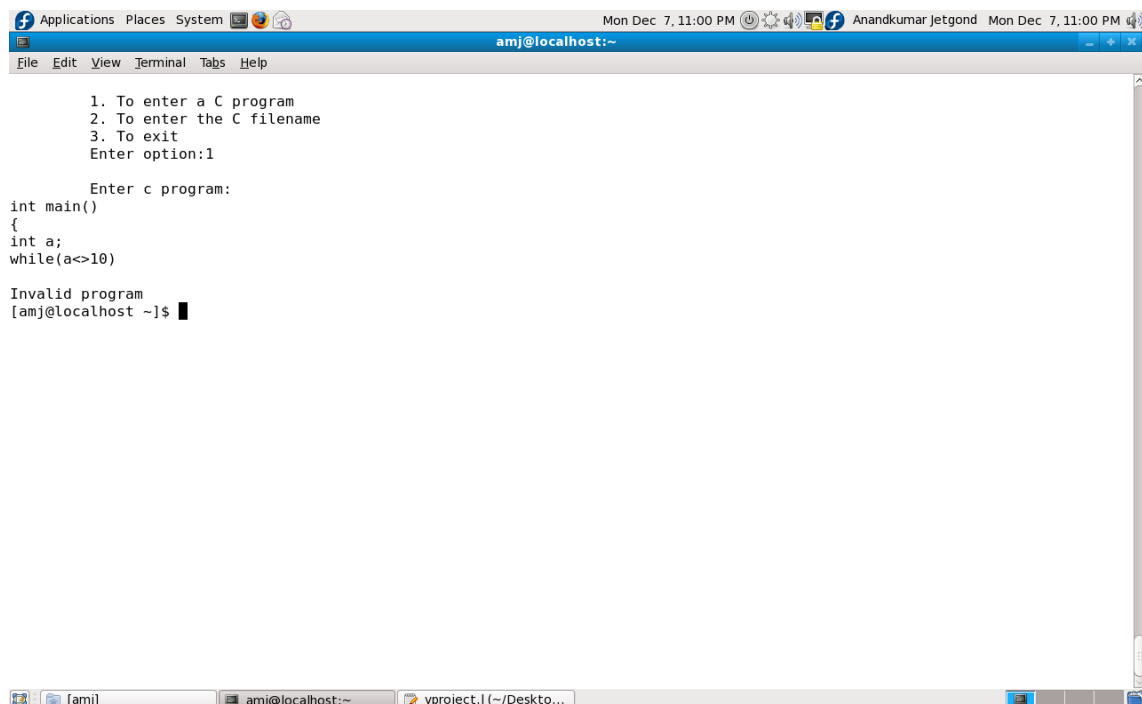


```
1. To enter a C program
2. To enter the C filename
3. To exit
Enter option:1

Enter c program:
int main()
{
int a;
for(a=10; a<20; a++)
}

Invalid program
[amj@localhost ~]$
```

- Invalid while statement (error in looping condition):

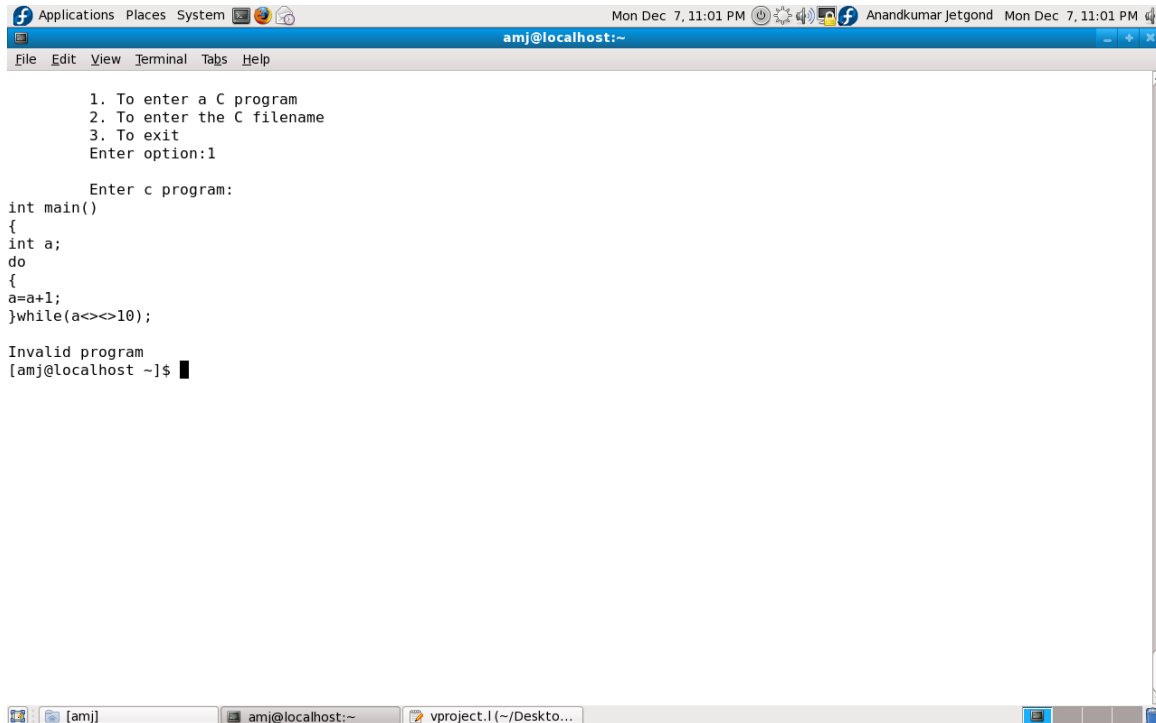


```
1. To enter a C program
2. To enter the C filename
3. To exit
Enter option:1

Enter c program:
int main()
{
int a;
while(a<>10)
}

Invalid program
[amj@localhost ~]$
```

- Invalid do-while loop (error in the looping construct):



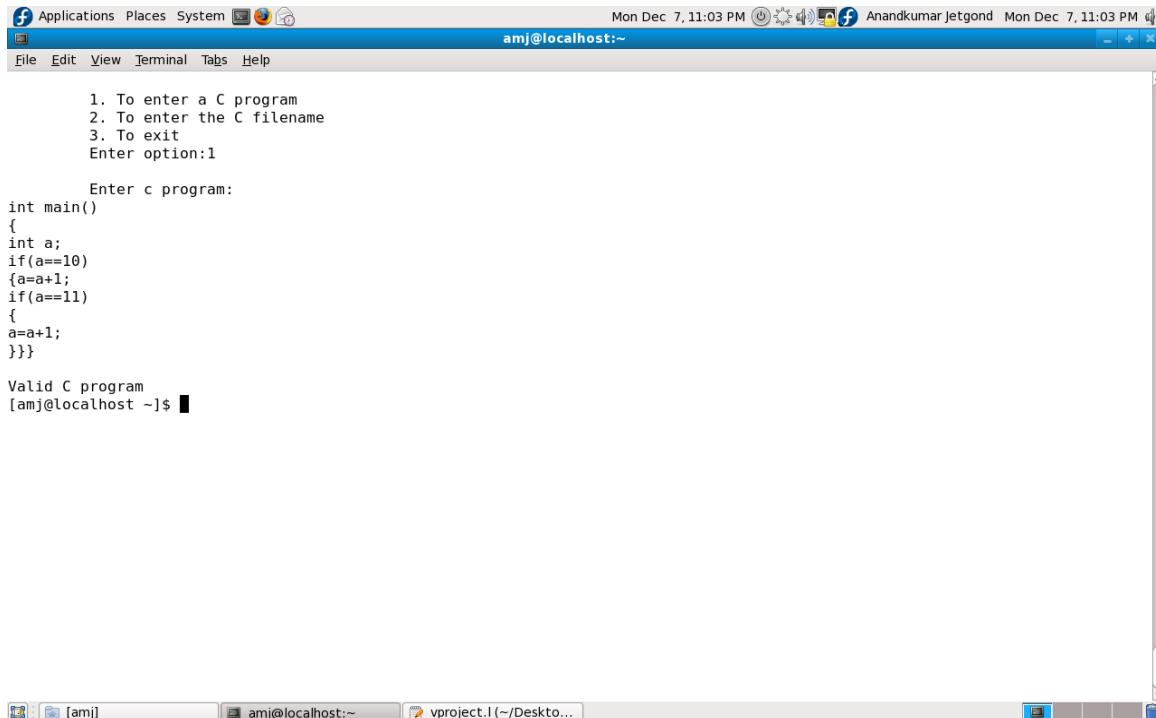
```
Applications Places System Mon Dec 7, 11:01 PM Anandkumar Jetgond Mon Dec 7, 11:01 PM
amj@localhost:~
File Edit View Terminal Tabs Help

1. To enter a C program
2. To enter the C filename
3. To exit
Enter option:1

Enter c program:
int main()
{
int a;
do
{
a=a+1;
}while(a<>10);

Invalid program
[amj@localhost ~]$
```

- Nested if statement:



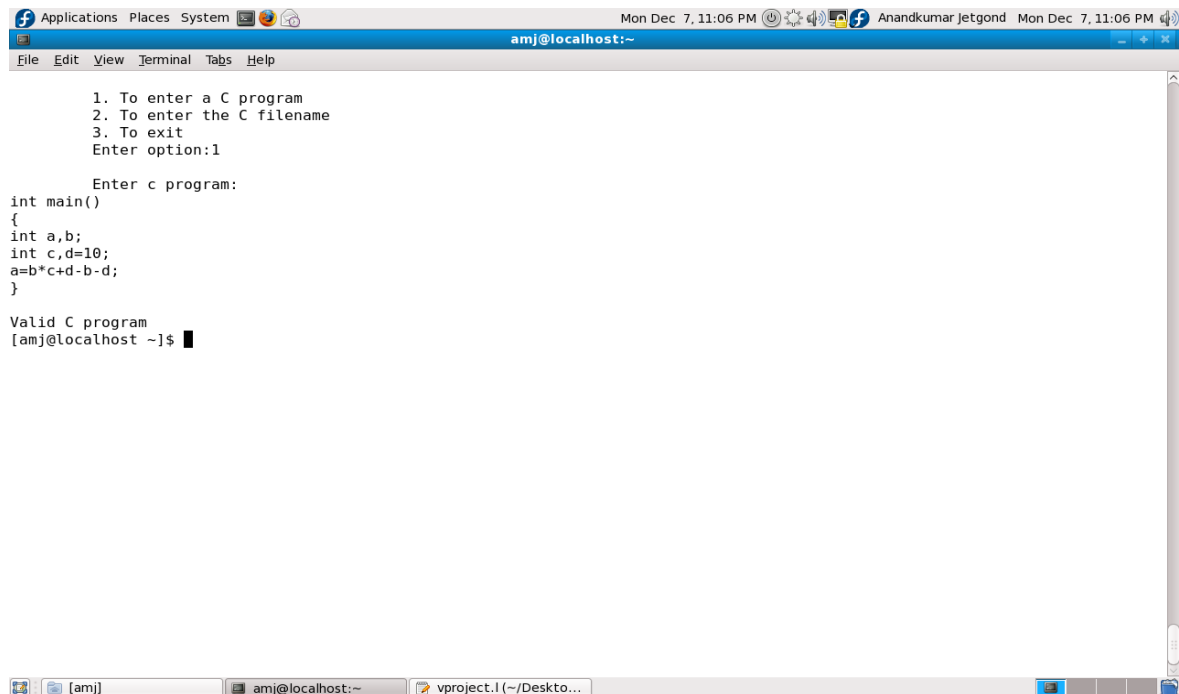
```
Applications Places System Mon Dec 7, 11:03 PM Anandkumar Jetgond Mon Dec 7, 11:03 PM
amj@localhost:~
File Edit View Terminal Tabs Help

1. To enter a C program
2. To enter the C filename
3. To exit
Enter option:1

Enter c program:
int main()
{
int a;
if(a==10)
{a=a+1;
if(a==11)
{
a=a+1;
}}}

Valid C program
[amj@localhost ~]$
```

- Valid expression:



The screenshot shows a terminal window titled "amj@localhost:~" with a menu bar (File, Edit, View, Terminal, Tabs, Help) and a system bar at the top. The terminal displays a menu with three options: 1. To enter a C program, 2. To enter the C filename, and 3. To exit. The user has entered option 1. The prompt "Enter option:1" is followed by "Enter c program:". The user has entered a C program snippet:

```
int main()
{
int a,b;
int c,d=10;
a=b*c+d-b-d;
}
```

 The terminal then displays "Valid C program" and the prompt "[amj@localhost ~]\$". The taskbar at the bottom shows three open windows: "[amj]", "amj@localhost:~", and "vproject.l (~/.Deskto...".

```
1. To enter a C program
2. To enter the C filename
3. To exit
Enter option:1

Enter c program:
int main()
{
int a,b;
int c,d=10;
a=b*c+d-b-d;
}

Valid C program
[amj@localhost ~]$
```