

**COMPILER DESIGN LAB MANUAL**



**AURORA’S SCIENTIFIC TECHNOLOGICAL &RESEARCHACADEMY**

**(Affiliated to JNTUniversityand Approved by AICTE)**

Bandlaguda, Hyderabad.

1. **OBJECTIVE AND RELEVANCE**

**Compiler Design** is the first course in complier design. The emphasis is on solving problems universally encountered in designing a language translator, regardless of the source or target machine. The first unit introduces the basic structure of a complier. It specifies the types of grammars used in complier construction and how we can eliminate the ambiguity in the grammar. It also covers major parsing techniques in depth, ranging from Brute Force technique to LR techniques that are used parser generators. It also introduces the principal ideas in syntax – directed translation, which is useful for both specifying and implementing translation.

It presents the main ideas for performing static semantic checking. Type checking and polymorphism functions are also dealt. Storage organizations used to support the run- time environment of a program is discussed. Data structure used in construction of symbol table is presented. It describes about intermediate languages and then shows how common programming language constructs can be translated into intermediate code. Different intermediate code representations are also presented. It also discusses about memory allocation and register allocation strategies.

**2.** **OUTCOMES**

Upon completion of the course, students should possess the following skills:

• Compiler Design: Given a basic compiler definition, students can decide the corresponding compiler structure and identify the relationships among different phases of the compiler.

• Compiler Implementation: Given a compiler design, students can implement required modules, which may include front-end, back-end, and a small set of middle-end optimizations.

• Documentation: Students can document the compiler structure and phases which conforms to the underlying programming language specifications.

• Cost Analysis: Given a high-level programming language construct, students can analyze and quantify (where appropriate) different potential implementations of the construct from the performance and space requirement viewpoints.

• Testing and Debugging: During compiler development process, students can utilize basic testing methodologies and debugging tools such as stubs, drivers, and integrated debuggers to identify fault points and possible error conditions, and exceptions with trycatch blocks.

• Broader Application: Students can apply classical compilation techniques to diverse data processing applications such as word processors and workflow managers

**3.**    **EQUIPMENT REQUIRED**

**Hardware**

No. of System                         :           60(IBM)

Processor                                :           PIV™ 1.67 GHz

RAM                                      :           512 MB

Hard Disk                               :           40 GB

Mouse                                     :           Optical Mouse

Network Interface card          :           Present

**Software**

Operating System                :           Window XP

Application Software             :           Turboc C++

Office                                    :           Ms-Office-2007

1. **CODE OF CONDUCT**

* **Students should report to the concerned lab as per the time table.**
* **Students who turn up late to the labs will in no case be permitted to do the program schedule for the day.**
* **After completion of the program, certification of the concerned staff in-charge in the observation book is necessary.**
* **Student should bring a notebook of 100 pages and should enter the readings/observations into the notebook while performing the experiment.**
* **The record of observations along with the detailed experimental procedure of the experiment in the immediate last session should be submitted and certified staff member in-charge.**
* **Not more than 3-students in a group are permitted to perform the experiment on the set.**
* **The group-wise division made in the beginning should be adhered to and no mix up of students among different groups will be permitted.**
* **The components required pertaining to the experiment should be collected from stores in-charge after duly filling in the requisition form.**
* **When the experiment is completed, should disconnect the setup made by them, and should return all the components/instruments taken for the purpose.**
* **Any damage of the equipment or burn-out components will be viewed seriously either by putting penalty or by dismissing the total group of students from the lab for the semester/year.**
* **Students should be present in the labs for total scheduled duration**
* **Students are required to prepare thoroughly to perform the experiment before coming to laboratory**

**5.**    **SYLLABUS ANALYSIS**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No** | **Name of the Experiment** | **Unit No** | **Text /Reference Books** |
| 1 | Design a Lexical analyzer. The lexical analyzer should ignore redundant s tabs and new lines. It should also ignore comments. Although the syntax specification s those identifiers can be arbitrarily long, you may restrict the length to some reasonable Value. | 1 | T1-Ch3 (109-114) |
| 2 | Implement the lexical analyzer using JLex, flex or lex other lexical analyzer generating tools | 1 | T1-Ch3 (140-146) |
| 3 | Design Predictive Parser for the given language | 2 | T1-Ch4 (222-231) |
| 4 | Design LALR bottom up parser for the above language | 3 | T1-Ch4 (259-277) |
| 5 | Convert the BNF rules into YACC form and write code to generate abstract syntax tree | 4 | T1-Ch4 (287-297) |
| 6 | Write program to generate machine code from the abstract syntax tree generated by the Parser .The following instruction set may considered as target code. | 5 | T1-Ch8 (505-544) |

**6. List of text book** (**JNTUH Prescribed Text Books)**

**TEXT BOOKS**

T1 : Principles of Compiler Design, ULLMAN –Narosa

**REFERENCE BOOKS**

ModeRN Compiler Implementation in C, Andrew N. Appel,Cambridge,University Press

Lex&yacc, JohnR.Levine,TonyMason,Doug,Brown,O’relly

.

**7.**    **SESSION PLAN**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SL.NO | Week no. | Unit no. | Activity | Remarks |
| 1 | 1 | 1 | **Write programs in ‘C’ Language to demonstrate the working of the following**  **constructs:**  **i) do…while ii) while…do iii) if …else iv) switch v) for**  **i.Program to find the factorial of a given number(using DO WHILE)** | Prerequisite |
| 2 | 2 | 1 | Design a Lexical analyzer. The lexical analyzer should ignore redundant s tabs and new lines. It should also ignore comments. Although the syntax specification s those identifiers can be arbitrarily long, you may restrict the length to some reasonable Value. | JNTUH |
| 3 | 3 | 1 | Implement the lexical analyzer using JLex, flex or lex other lexical analyzer generating tools | JNTUH |
| 4 | 4 | 2 | Design Predictive Parser for the given language | JNTUH |
| 5 | 5 | 3 | Design LALR bottom up parser for the above language | JNTUH |
| 6 | 6 | 4 | Convert the BNF rules into YACC form and write code to generate abstract syntax tree | JNTUH |
| 7 | 7 | 5 | Write program to generate machine code from the abstract syntax tree generated by the Parser .The following instruction set may considered as target code. | JNTUH |

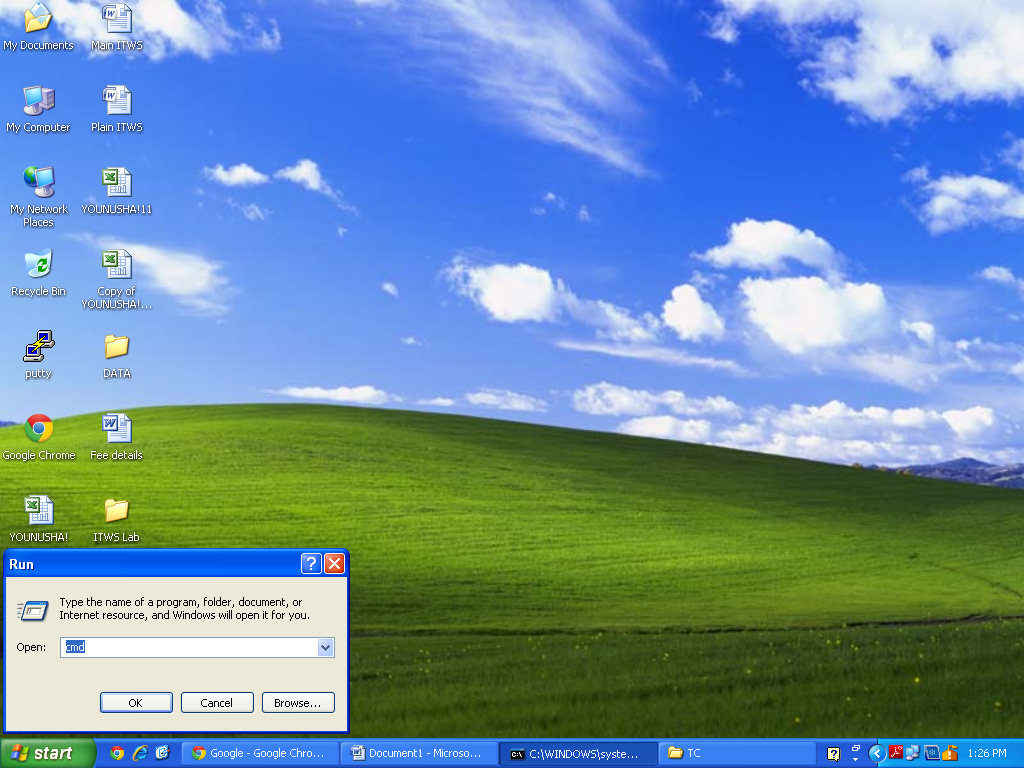
**8. Each Experiment Write Up**

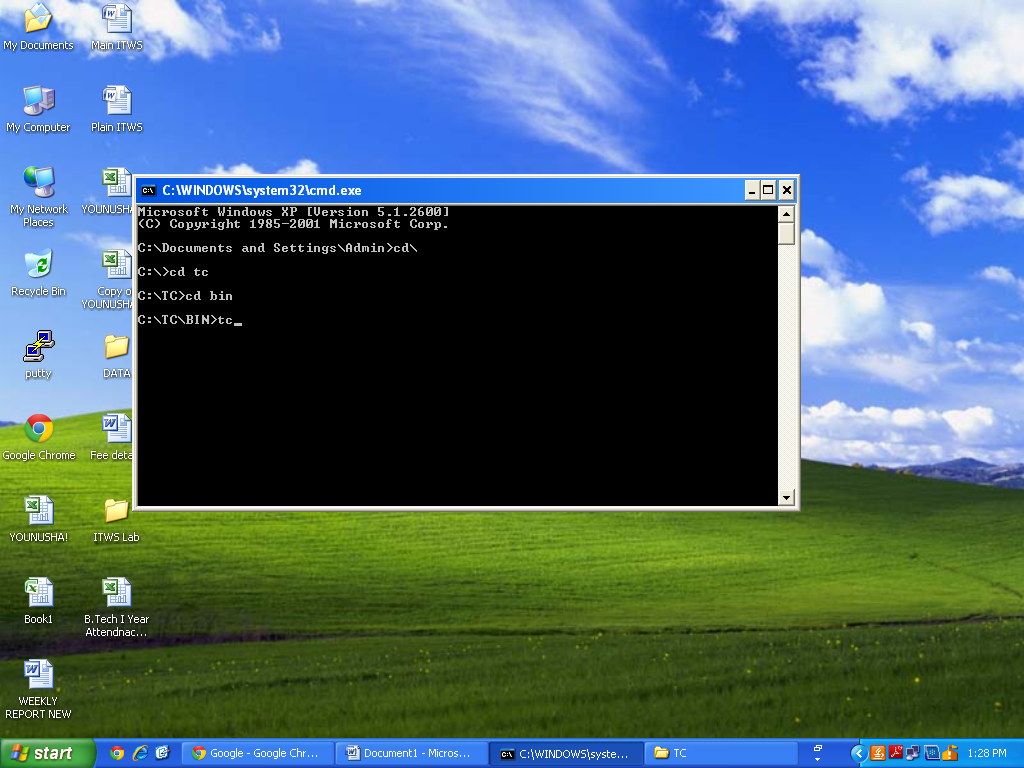
**8. 1    Introduction to Lab**

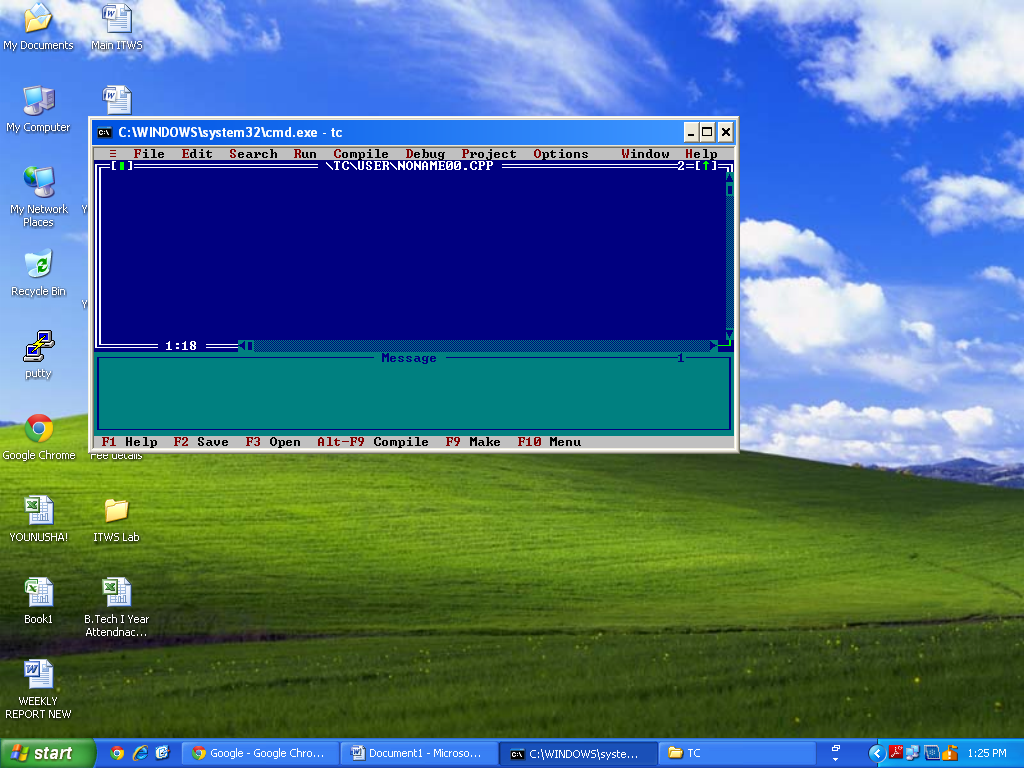
Get familiar with the C Programming Language: If you don't already know it, in this lab you will get started on learning a new programming language. Although this is not meant to be a tutorial, you will get exposed to the most basic features of the language by reading and analyzing some basic programs.

**How to Open Software Screen Shots**

How to Run C programs on Turbo C editor







#include

void main()

{

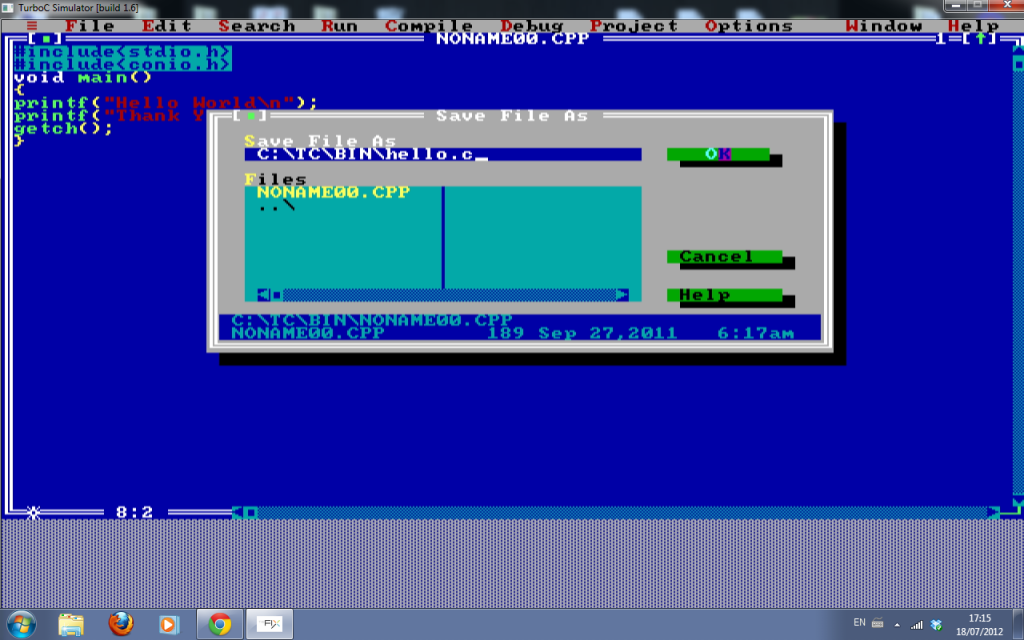
printf("Hello World\n");

printf("Thank You");

getch();

}

I hope that’s enough for a basic explanation of the program. If you still have doubts please ask through comments. Now let’s**RUN this program using Turbo C**. Before going into the steps, you may SAVE your C program. Select**“File”** from menu -> click-> **Save**. Name the files as ->**hello.c** or some other name with a .c extension. See the screen shot below.



**How to Compile a C program in Turbo C**

**The first step is compiling.** Compiling makes sure your program is **free of syntax errors**. How ever compiler won’t check for any logical/algorithmic errors. There is a lot of process that happens while the compiler compiles a program – which we will discuss later in coming articles. To do compiling -**Select -> Compile** from menu and**click-> compile**. See the image below.

[](http://www.circuitstoday.com/wp-content/uploads/2012/07/hello-2.png)

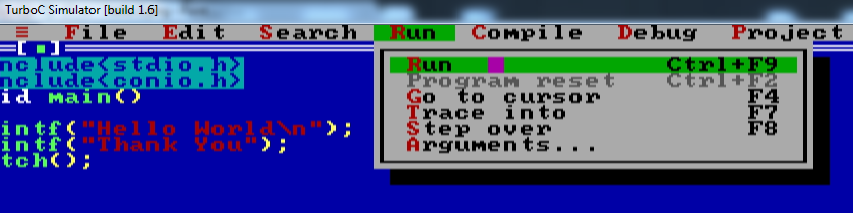
After compiling, you will see a dialog box as shown below.If the compilation is success – you will see a**“success”** message. Else you will see the number of errors. Both are shown using screen shots.

**The screen shot of a “success” compilation**

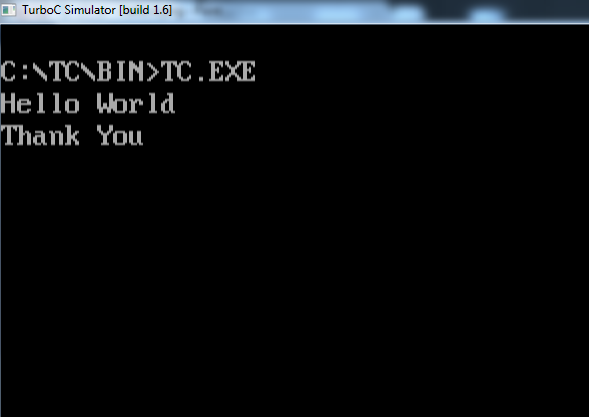
[](http://www.circuitstoday.com/wp-content/uploads/2012/07/hello-3.png)

**How to RUN a C Program in Turbo C compiler?**

To RUN the program – you may **select ->Run** from menu and**click -> Run** (as shown in the image below).



**Now you will see the output screen as shown in the screen shot below.**



**1. Write programs in ‘C’ Language to demonstrate the working of the following**

**constructs:**

**i) do…while ii) while…do iii) if …else iv) switchv) for**

**ANSWER:**

**I.Programto find the factorial of a given number (using DO WHILE)**

#include<stdio.h>

void main(){

intn,i,f;

f=i=1;

clrscr();

printf("enter a number");

scanf("%d",&n);

do

{

f\*=i;

i++;

}

while(i<=n);

printf("the factorial of %d is %d",n,f);

getch();

}

**Result:**Enter a number 5

The factorial of 5 is 120

**II.Programto find the factorial of a given number (using WHILE DO)**

#include<stdio.h>

void main(){

intn,i,f;

f=i=1;

clrscr();

printf("enter a number");

scanf("%d",&n);

while(i<=n)

{

f\*=i;

i++;

}

printf("the factorial of %d is %d",n,f);

getch();

}

**Result:** Enter a number 5

The factorial of 5 is 120

**III.Program to find Gratest of 3 numbers to print the given no in ascending order(using IF-ELSE)**

#include<stdio.h>

void main()

{

inta,b,c;

clrscr();

printf("enter the values of a,b and c");

scanf("%d%d%d",&a,&b,&c);

if(a<b && a<c)

{

if(b<c)

{

printf(" %d%d%d", a,b,c);

}

else

if(b>c)

printf(" %d%d%d",a,c,b);

}

else

if(b<c && b<a)

{

if(c<a)

printf(" %d%d%d",b,c,a);

else

printf("%d%d%d",b,a,c);

}

else

if(b<a)

printf("%d%d%d",c,b,a);

else

printf(%d%d%d",c,a,b);

}

}

**Result:**Enter the values of a, b and c

6

4

5

4 5 6

**IV.Program to perform the arithmetic expression using switch statement (using SWITCH STMT)**

#include<stdio.h>

#include<conio.h>

void main()

{

inta,b;

int op;

clrscr();

printf(" 1.addition\n 2.subtraction\n 3.multiplication\n 4.division\n");

printf("enter the values of a & b");

scanf("%d%d",&a,&b);

printf("enter your choice : ");

scanf("%d",&op);

switch(op)

{

case 1 :printf("sum of %d and %d=%d",a,b,a+b);

break;

case 2 :printf("difference of %d and %d=%d",a,b,a-b);

break;

case 3 :printf("multiplication of %d and %d=%d",a,b,a\*b);

break;

case 4 :printf("Divisionn of two numbers is %d=",a/b);

break;

default : printf(" Enter Your Correct Choice.");

break;

}

getch();

}

**Result:**

1. Addition

2. Substraction

3. Multiplication

4. Division

Enter your choice : 1

Enter a and b values 10 20

Sum of 10 and 20 = 30

**2. A program written in ‘C’ language for matrix multiplication fails” Introspect the causes for its failure and write down the possible reasons for its failure**

#include<stdio.h>

main()

{

int m1[10][10],i,j,k,m2[10][10],mult[10][10],r1,c1,r2,c2;

printf("Enter number of rows and columns of first matrix (less than 10)\n");

scanf("%d%d",&r1,&c1);

printf("Enter number of rows and columns of second matrix (less than 10)\n");

scanf("%d%d",&r2,&c2);

if(r2==c1)

{

printf("Enter rows and columns of First matrix \n");

printf("Row wise\n");

for(i=0;i<r1;i++)

for(j=0;j<c1;j++)

scanf("%d",&m1[i][j]); printf("First Matrix is :\n"); for(i=0;i<r1;i++)

{

for(j=0;j<c1;j++)

printf("%d\t",m1[i][j]);

printf("\n");

}

printf("Enter rows and columns of Second matrix \n");

printf("Row wise\n");

for(i=0;i<r2;i++)

for(j=0;j<c2;j++)

scanf("%d",&m2[i][j]);

printf("Second Matrix is:\n"); for(i=0;i<r2;i++)

{

for(j=0;j<c2;j++)

printf("%d\t",m2[i][j]);

printf("\n");

}

printf("Multiplication of the Matrices:\n");

for(i=0;i<r1;i++)

{

for(j=0;j<c2;j++)

{

mult[i][j]=0;

for(k=0;k<r1;k++)

mult[i][j]+=m1[i][k]\*m2[k][j];

printf("%d\t",mult[i][j]);

}

printf("\n");

}

}

else

{

printf("Matrix multiplication cannot be done");

}

return 0;

}

**RESULT**

Enter the elements of matrix a

1 2 4 5 2 1 4 5 2

Enter the elements of matrix b

1 2 4 5 2 1 4 5 2

10 18 28

50 18 7

40 45 14

**8.2.1**

Design a Lexical analyzer. The lexical analyzer should ignore redundant s tabs and new lines. It should also ignore comments. Although the syntax specification s those identifiers can be arbitrarily long, you may restrict the length to some reasonable Value

**PREAMBLE:**

Lexical analysis in ANTLR is treated similarly to parsing. You specify rules that correspond to tokens, and define which sequences of characters match those rules (and thus form those tokens) using an EBNF-like syntax. The rules are specified in “grammar” files which are text files with a “.g” extension (by convention). In this tutorial we will be using a single grammar file to describe the lexical analyser, parser and tree parser. Let’s start with an overview of the ANTLR syntax for specifying lexers (literal parts enclosed in quotes):

**AIM** :

Design a Lexical analyzer. The lexical analyzer should ignore redundant s tabs and new lines. It should also ignore comments. Although the syntax specification s those identifiers can be arbitrarily long, you may restrict the length to some reasonable Value.

**THEOREY :**

**The first phase of a compiler is called lexical analysis . The lexical analyser reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme , the lexical analyzer produces as output a token, that it passes on to the subsequent phase, syntax analysis.**

**PROCEDURE:**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*  C Program to Design Lexical Analyzer\*/

/\*  <span class="IL\_AD" id="IL\_AD4">Download</span> more programs at <http://sourcecode4u.com/> \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include<string.h>

#include<ctype.h>

#include<stdio.h>

voidkeyword(charstr[10])

{ if(<span class="IL\_AD"id="IL\_AD2">strcmp</span>("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||

    strcmp("int",str)==0||strcmp("float",str)==0||strcmp("char",str)==0||

    strcmp("double",str)==0||strcmp("static",str)==0||strcmp("switch",str)==0||

    strcmp("case",str)==0)

    <span class="IL\_AD"id="IL\_AD9">printf</span>("\n%s is a keyword",str);

    else

    printf("\n%s is an identifier",str);

}

<span class="IL\_AD"id="IL\_AD10">main</span>()

{

    FILE\*f1,\*f2,\*f3;

    charc,str[10],st1[10];

    intnum[100],lineno=0,tokenvalue=0,i=0,j=0,k=0;

    printf("\nEnter<span class="IL\_AD" id="IL\_AD3">the c program</span>");/\*gets(st1);\*/

    f1=fopen("input","w");

    while((c=getchar())!=EOF)

    putc(c,f1);

    fclose(f1);

    f1=fopen("input","r");

    f2=fopen("identifier","w");

    f3=fopen("specialchar","w");

    while((c=getc(f1))!=EOF)

    {

        if(isdigit(c))

        {

            tokenvalue=c-'0';

            c=getc(f1);

            while(isdigit(c))

            {

                tokenvalue\*=10+c-'0';

                c=getc(f1);

            }

            num[i++]=tokenvalue;

            ungetc(c,f1);

        } elseif(isalpha(c))

        {

            putc(c,f2);

            c=getc(f1);

            while(isdigit(c)||isalpha(c)||c=='\_'||c=='$')

            {

                putc(c,f2);

                c=getc(f1);

            }

            putc(' ',f2);

            ungetc(c,f1);

        } elseif(c==' '||c=='\t')

        printf(" ");

        elseif(c=='\n')

        lineno++;

        else

        putc(c,f3);

    }

    fclose(f2);

    fclose(f3);

    fclose(f1);

    printf("\nThe no's in the program are");

    for(j=0;j<i;j++)

    printf("%d",num[j]);

    printf("\n");

    f2=fopen("identifier","r");

    k=0;

    printf("The <span class="IL\_AD" id="IL\_AD8">keywords</span> and identifiersare:");

    while((c=getc(f2))!=EOF)

    {

        if(c!=' ')

        str[k++]=c;

        else

        {

            str[k]='\0';

            keyword(str);

            k=0;

        }

    }

    fclose(f2);

    f3=fopen("specialchar","r");

    printf("\nSpecial characters are");

    while((c=getc(f3))!=EOF)

    printf("%c",c);

    printf("\n");

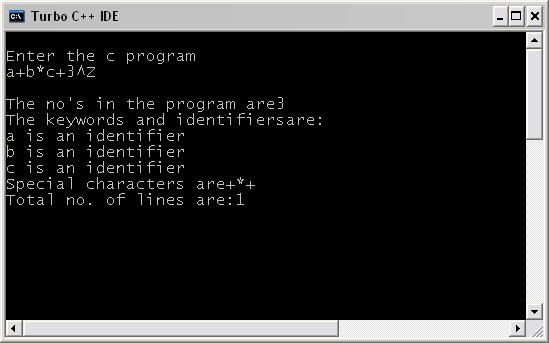
    fclose(f3);

    printf("Total no. of lines are:%d",lineno);

}

**RESULT &CONCLUSION :**

**SAMPLEOUTPUT:**



**RESULT:**

Thus,the C program for tokens recognised was executed and the programwas identified.

**APPLICATION**

**Lexical analyzer Used mainly in identifying tokens in Compiler Construction.**

**SAMPLE QUESTIONS**

1. What is Parsing?
2. What is Token?
3. What is Jlex?
4. What is Flex?
5. What is LALR parsing?
6. What is Shift reduced parser?
7. What are the operations of Parser?
8. What is the use of parsing table?
9. What is bottom up parsing?

**VIVA QUESTIONS**

1. .Whatat is lexical analyzer?
2. Which compiler is used for lexical analyzer?
3. What is YACC?
4. What is the output of Lexical analyzer?
5. What is LEX source Program?

**8.2.2**Implement the lexical analyzer using JLex, flex or lex other lexical analyzer generating tools

**PREAMBLE** :

A Flex lexical analyzer usually has time complexity O(n)in the length of the input. That is, it performs a constant number of operations for each input symbol. This constant is quite low: [GCC](http://en.wikipedia.org/wiki/GNU_Compiler_Collection) generates 12 instructions for the DFA match loop. Note that the constant is independent of the length of the token, the length of the regular expression and the size of the DFA.

However, one optional feature of Flex can cause Flex to generate a scanner with non-linear performance: The use of the REJECT macro in a scanner with the potential to match extremely long tokens. In this case, the programmer has explicitly told flex to "go back and try again" after it has already matched some input. This will cause the DFA to backtrack to find other accept states. The REJECT feature is not enabled by default, and because of its performance implications its use is discouraged in the Flex manual.[[7]](http://en.wikipedia.org/wiki/Flex_lexical_analyser#cite_note-performance-7)

**AIM :**

Implement the lexical analyzer using JLex, flex or lex other lexical analyzer generating tools

**THEOREY :**

These programs perform character parsing, and tokenizing via the use of a [deterministic finite automaton](http://en.wikipedia.org/wiki/Deterministic_finite_automaton) (DFA). A DFA is a theoretical machine accepting [regular languages](http://en.wikipedia.org/wiki/Regular_language). These machines are a subset of the collection of [Turing machines](http://en.wikipedia.org/wiki/Turing_machine). DFAs are equivalent to [read-only right moving Turing machines](http://en.wikipedia.org/wiki/Read-only_right_moving_Turing_machines). The syntax is based on the use of [regular expressions](http://en.wikipedia.org/wiki/Regular_expressions). See also [nondeterministic finite automaton](http://en.wikipedia.org/wiki/Nondeterministic_finite_automaton).

**PROCEDURE :**

Step1: Start

Step2: Declare the declarations for the given language tokens like digit, alphabet, white space, delimiters, etc.

digit[0-9]

letter[A-Z a-z]

delim[\t\n]

W${delim}+ID{(letter)(letter/digit)}+

Integer {digit}+

% %

{ws} {print (“SpecialCharacters”)}

{ID} {print(“Identifiers”)}

{digit} {print(“\n Integer)}

if {printf(“keyword”)}

else {print(keyword)}

“&&” {print(logoical operators)}

“>”{print(logoical operators)}

“<”{print(logoical operators)}

“<=”{print(logoical operators)}

“>=”{print(logoical operators)}

“=” {printf(“\n \n”)}

“!”{printf(“\n \n”)}

“+” {printf(“arithmetic operator”)}

“-“ {printf (“arithmetic”)

“\*” {printf(“arithmetic”)}

“%” {printf(arithmetic”)}

% % {printf(“arithmetic”)}

Step3: Write the auxillaryprocedure in main() function

Step4: end

Step5: Stop.

**RESULT :**

Input:

$vivar.c

#include<stdio.h>

main()

{

inta,b;

}

Output:

$lexlex.l

$cclex.yy.c

$./a.outvar.c

#include<stdio.h>isa PREPROCESSOR DIRECTIVE FUNCTION

main() BLOCK BEGINS

int is a Keyword

aIDENTIFIER

b IDENTIFIER

BLOCK ENDS

**APPLICATION**

**JLEX Used mainly in identifying tokens in Compiler Construction**

**SAMPLE QUESTIONS**

1. What is Predictive parser?
2. How many types of analysis can we do using Parser?
3. What is Recursive Decent Parser?
4. How many types of Parsers are there?
5. What is LR Parser?

**VIVA QUESTIONS**

1. What is Parsing?
2. What is Token?
3. What is Jlex?
4. What is Flex?

**8.2.3.** Design Predictive Parser for the given language

**Preamble:**

Predictive Parsing is a parsing (Top-Down Parsing) .In Which Recursive Descent and Non Recursive Descent Parsing Techniques are used.

**Aim:**

Design Predictive Parser for the given language

**Theory :**

**Predictive Parsing is a Parsing In Which Back Tracking Does not takes place.**

**It Uses The Following three cases to parse a string**

**If X=a=$ , then Parser Announces Parsing is Success fully Completed**

**If X=a=!$, then Parser pops X off the stack and moves look ahead pointer advance**

**If X=!a=!$ then Parser Places Production on M[X,a] then Proceeds the above actions.**

**PROCEDURE:**

#include<stdio.h>

#include<ctype.h>

#include<string.h>

#include<stdlib.h>

#define SIZE 128

#define NONE -1

#define EOS '\0'

#define NUM 257

#define KEYWORD 258

#define ID 259

#define DONE 260

#define MAX 999

char lexemes[MAX];

char buffer[SIZE];

intlastchar=-1;

intlastentry=0;

inttokenval=DONE;

intlineno=1;

intlookahead;

struct entry

{

char \*lexptr;

int token;

}

symtable[100];

struct entry

keywords[]= {"if",KEYWORD,"else",KEYWORD,"for",KEYWORD,"int",KEYWORD,"float",KEYWORD,"double",KEYWORD,"char",KEYWORD,"struct",KEYWORD,"return",KEYWORD,0,0

};

voidError\_Message(char \*m)

{

fprintf(stderr,"line %d, %s \n",lineno,m);

exit(1);

}

intlook\_up(char s[ ])

{

int k;

for(k=lastentry; k>0; k--)

if(strcmp(symtable[k].lexptr,s)==0)

return k;

return 0;

}

int insert(char s[ ],inttok)

{

intlen;

len=strlen(s);

if(lastentry+1>=MAX)

Error\_Message("Symbpl table is full");

if(lastchar+len+1>=MAX)

Error\_Message("Lexemes array is full");

lastentry=lastentry+1;

symtable[lastentry].token=tok;

symtable[lastentry].lexptr=&lexemes[lastchar+1];

lastchar=lastchar+len+1;

strcpy(symtable[lastentry].lexptr,s);

returnlastentry;

}

/\*void Initialize()

{

struct entry \*ptr;

for(ptr=keywords;ptr->token;ptr+1)

insert(ptr->lexptr,ptr->token);

}\*/

intlexer()

{

int t;

intval,i=0;

while(1)

{

t=getchar();

if(t==' '||t=='\t');

else if(t=='\n')

lineno=lineno+1;

else if(isdigit(t))

{

ungetc(t,stdin);

scanf("%d",&tokenval);

return NUM;

}

else if(isalpha(t))

{

while(isalnum(t))

{

buffer[i]=t;

t=getchar();

i=i+1;

if(i>=SIZE)

Error\_Message("Compiler error");

}

buffer[i]=EOS;

if(t!=EOF)

ungetc(t,stdin);

val=look\_up(buffer);

if(val==0)

val=insert(buffer,ID);

tokenval=val;

returnsymtable[val].token;

}

else if(t==EOF)

return DONE;

else

{

tokenval=NONE;

return t;

}

}

}

void Match(int t)

{

if(lookahead==t)

lookahead=lexer();

else

Error\_Message("Syntax error");

}

void display(intt,inttval)

{

if(t=='+'||t=='-'||t=='\*'||t=='/')

printf("\nArithmetic Operator: %c",t);

else if(t==NUM)

printf("\n Number: %d",tval);

else if(t==ID)

printf("\n Identifier: %s",symtable[tval].lexptr);

else

printf("\n Token %d tokenval %d",t,tokenval);

}

void F()

{

//void E();

switch(lookahead)

{

case '(' :

Match('(');

E();

Match(')');

break;

case NUM :

display(NUM,tokenval);

Match(NUM);

break;

case ID :

display(ID,tokenval);

Match(ID);

break;

default :

Error\_Message("Syntax error");

}

}

void T()

{

int t;

F();

while(1)

{

switch(lookahead)

{

case '\*' :

t=lookahead;

Match(lookahead);

F();

display(t,NONE);

continue;

case '/' :

t=lookahead;

Match(lookahead);

display(t,NONE);

continue;

default :

return;

}

}

}

void E()

{

int t;

T();

while(1)

{

switch(lookahead)

{

case '+' :

t=lookahead;

Match(lookahead);

T();

display(t,NONE);

continue;

case '-' :

t=lookahead;

Match(lookahead);

T();

display(t,NONE);

continue;

default :

return;

}

}

}

void parser()

{

lookahead=lexer();

while(lookahead!=DONE)

{

E();

Match(';');

}

}

main()

{

charans[10];

printf("\n Program for recursive decent parsing ");

printf("\n Enter the expression ");

printf("And place ; at the end\n");

printf("Press Ctrl-Z to terminate\n");

parser();

}

RESULT/CONCLUSION:

Program for recursive decent parsing

Enter the expression And place ; at the end

Press Ctrl-Z to terminate

a\*b+c;

Identifier: a

Identifier: b

Arithmetic Operator: \*

Identifier: c

Arithmetic Operator: +

5\*7;

Number: 5

Number: 7

Arithmetic Operator: \*

\*2;

line 5, Syntax error

**APPLICATION**

**Predictive Parser is Used to Recognize Tokens in Compiler Construction**

**SAMPLE QUESTIONS**

1. What is Parsing?
2. What is Token?
3. What is Jlex?
4. What is Flex?

**VIVA QUESTIONS**

1. What is Predictive parser?
2. How many types of analysis can we do using Parser?
3. What is Recursive Decent Parser?
4. How many types of Parsers are there?
5. What is LR Parser?

**8.2.4**Design LALR bottom up parser for the above language

**PREAMBLE :**

an **LALR parser**[[a]](http://en.wikipedia.org/wiki/LALR_parser#cite_note-1) or **Look-Ahead LR parser** is a simplified version of a [canonical LR parser](http://en.wikipedia.org/wiki/Canonical_LR_parser), to parse (separate and analyze) a text according to a set of [production rules](http://en.wikipedia.org/wiki/Production_%28computer_science%29) specified by a [formal grammar](http://en.wikipedia.org/wiki/Formal_grammar) for a [computer language](http://en.wikipedia.org/wiki/Computer_language). ("LR" means left-to-right, [rightmost derivation](http://en.wikipedia.org/wiki/Rightmost_derivation).)

The LALR parser was invented by [Frank DeRemer](http://en.wikipedia.org/w/index.php?title=Frank_DeRemer&action=edit&redlink=1) in his 1969 PhD dissertation, *Practical Translators for LR(k) languages*,[[1]](http://en.wikipedia.org/wiki/LALR_parser#cite_note-FOOTNOTEDeRemer1969-2) in his treatment of the practical difficulties at that time of implementing LR(1) parsers. He showed that the LALR parser has more language recognition power than the LR(0) parser, while requiring the same number of states as the LR(0) parser for a language that can be recognized by both parsers. This makes the LALR parser a memory-efficient alternative to the LR(1) parser for languages that are not LR(0). It was also proved that there exist LR(1) languages that are not LALR. Despite this weakness, the power of the LALR parser is enough for many mainstream computer languages,[[2]](http://en.wikipedia.org/wiki/LALR_parser#cite_note-chapman-3) including [Java](http://en.wikipedia.org/wiki/Java_technology),[[3]](http://en.wikipedia.org/wiki/LALR_parser#cite_note-4) though the reference grammars for many languages fail to be LALR due to being [ambiguous](http://en.wikipedia.org/wiki/Ambiguous_grammar).[[2]](http://en.wikipedia.org/wiki/LALR_parser#cite_note-chapman-3)

The original dissertation gave no algorithm for constructing such a parser given some formal grammar. The first algorithms for LALR parser generation were published in 1973.[[4]](http://en.wikipedia.org/wiki/LALR_parser#cite_note-5) In 1982, DeRemer and Penello published an algorithm that generated highly memory-efficient LALR parsers.[[5]](http://en.wikipedia.org/wiki/LALR_parser#cite_note-DeRemer82-6) LALR parsers can be automatically generated from some grammar by an [LALR parser generator](http://en.wikipedia.org/wiki/LALR_parser_generator) such as [Yacc](http://en.wikipedia.org/wiki/Yacc) or [GNU Bison](http://en.wikipedia.org/wiki/GNU_Bison). The automatically generated code may be augmented by hand-written code to augment the power of the resulting parser.

**AIM :**Design LALR bottom up parser for the above language

**THEOREY:**

**LALR Stands For Look Ahead LR . It is one of the Shift-Reduce Bottom Up parser .**

**It Per performs the following Four Actions**

**1.Shift**

**2.Reduce**

**3.Accept**

**4.Error**

**PROCEDURE :**

Step1: Start

Step2: Initially the parser has s0 on the stack where s0 is the initial state and w$ is in buffer

Step3: Set ip point to the first symbol of w$

Step4: repeat forever, begin

Step5: Let S be the state on top of the stack and a symbol pointed to by ip

Step6: If action [S, a] =shift S then begin

Push S1 on to the top of the stack

Advance ip to next input symbol

Step7: Else if action [S, a], reduce A->B then begin

Pop 2\* |B| symbols of the stack

Let S1 be the state now on the top of the stack

Step8: Output the production A->B

End

Step9: else if action [S, a]=accepted, then return

Else

Error()

End

Step10: Stop

**RESULTS :**

$lexparser.l

$yacc–d parser.y

$cclex.yy.cy.tab.c –ll–lm

$./a.out

2+3

5.0000

**APPLICATION**

**LALR is Used to Recognize Tokens in Compiler Construction**

**SAMPLE QUESTIONS**

1. What is Predictive parser?
2. How many types of analysis can we do using Parser?
3. What is Recursive Decent Parser?
4. How many types of Parsers are there?
5. **What is LR Parser?**

**VIVA QUESTION**

1. What is LALR parsing?
2. What is Shift reduced parser?
3. What are the operations of Parser?
4. What is the use of parsing table?
5. What is bottom up parsing?

**8.2.5**

Convert the BNF rules into YACC form and write code to generate abstract syntax tree

**PREAMBLE:**

The most common formal system for presenting such rules for humans to read is **Backus-Naur Form** or "BNF", which was developed in order to specify the language Algol 60. Any grammar expressed in BNF is a context-free grammar. The input to Bison is essentially machine-readable BNF.

Not all context-free languages can be handled by Bison, only those that are LALR(1). In brief, this means that it must be possible to tell how to parse any portion of an input string with just a single token of look-ahead. Strictly speaking, that is a description of an LR(1) grammar, and LALR(1) involves additional restrictions that are hard to explain simply; but it is rare in actual practice to find an LR(1) grammar that fails to be LALR(1). See section [Mysterious Reduce/Reduce Conflicts](http://dinosaur.compilertools.net/bison/bison_8.html#SEC79), for more information on this.

**AIM** :

Convert the BNF rules into YACC form and write code to generate abstract syntax tree

**THEOREY:**

Grammars for yacc are described using a variant of Backus Naur Form (BNF). This technique, pioneered by John Backus and Peter Naur, was used to describe ALGOL60. A BNF grammar can be used to express *context-free* languages. Most constructs in modern programming languages can be represented in BNF. For example, the grammar for an expression that multiplies and adds numbers is

1 E -> E + E

2 E -> E \* E

3 E -> id

Three productions have been specified. Terms that appear on the left-hand side (lhs) of a production, such as E, are nonterminals. Terms such as id (identifier) are terminals (tokens returned by lex) and only appear on the right-hand side (rhs) of a production. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier. We can use this grammar to generate expressions:

E -> E \* E (r2)

-> E \* z (r3)

-> E + E \* z (r1)

-> E + y \* z (r3)

-> x + y \* z (r3)

At each step we expanded a term and replace the lhs of a production with the corresponding rhs. The numbers on the right indicate which rule applied. To parse an expression we a need to do the reverse operation. Instead of starting with a single nonterminal (start symbol) and generating an expression from a grammar we need to *reduce* an expression to a single nonterminal. This is known as *bottom-up* or *shift-reduce* parsing and uses a stack for storing terms. Here is the same derivation but in reverse order:

1 . x + y \* z shift

2 x . + y \* z reduce(r3)

3 E . + y \* z shift

4 E + .y \* z shift

5 E + y . \* z reduce(r3)

6 E + E . \* z shift

7 E + E \* .z shift

8 E + E \* z .reduce(r3)

9 E + E \* E .reduce(r2) emit multiply

10 E + E .reduce(r1) emit add

11 E .accept

Terms to the left of the dot are on the stack while remaining input is to the right of the dot. We start by shifting tokens onto the stack. When the top of the stack matches the rhs of a production we replace the matched tokens on the stack with the lhs of the production. In other words the matched tokens of the rhs are popped off the stack, and the lhs of the production is pushed on the stack. The matched tokens are known as a *handle* and we are *reducing* the handle to the lhs of the production. This process continues until we have shifted all input to the stack and only the starting nonterminal remains on the stack. In step 1 we shift the x to the stack. Step 2 applies rule r3 to the stack to change x to E. We continue shifting and reducing until a single nonterminal, the start symbol, remains in the stack. In step 9, when we reduce rule r2, we emit the multiply instruction. Similarly the add instruction is emitted in step 10. Consequently multiply has a higher precedence than addition.

Consider the shift at step 6. Instead of shifting we could have reduced and apply rule r1. This would result in addition having a higher precedence than multiplication. This is known as a *shift-reduce* conflict. Our grammar is *ambiguous* because there is more than one possible derivation that will yield the expression. In this case operator precedence is affected. As another example, associativity in the rule

E -> E + E

is ambiguous, for we may recurse on the left or the right. To remedy the situation, we could rewrite the grammar or supply yacc with directives that indicate which operator has precedence. The latter method is simpler and will be demonstrated in the practice section.

The following grammar has a *reduce-reduce* conflict. With an id on the stack we may reduce to T or E.

E -> T

E -> id

T -> id

Yacc takes a default action when there is a conflict. For shift-reduce conflicts yacc will shift. For reduce-reduce conflicts it will use the first rule in the listing. It also issues a warning message whenever a conflict exists. The warnings may be suppressed by making the grammar unambiguous. Several methods for removing ambiguity will be presented in subsequent sections.

**PROCEDURE :**

Step1: Start

Step2: declare the declarations as a header file

{include<ctype.h>}

Step3: token digit

Step4: define the translations rules like line, expr, term, factor

Line:exp ‘\n’ {print(“\n %d \n”,$1)}

Expr:expr’+’ term ($$=$1=$3}

Term:term ‘+’ factor($$ =$1\*$3}

Factor

Factor:’(‘enter’) ‘{$$ =$2)

% %

Step5: define the supporting C routines

Step6: Stop

**RESULT :**

{

inta,b,c;

if(a<b)

{

a=a+b;

}

while(a<b)

{

a=a+b;

}

if(a<=b)

{

c=a-b;

}

else

{

c=a+b;

Output:

$lexint.l

$yacc–d int.y

$gcclex.yy.cy.tab.c –ll–lm

$./a.outtest.c

**APPLICATION**

**Abstract syntax tree is used to check the syntax of expressions while determining tokens in compiler construction**

**SAMPLE QUESTIONS**

1. What is target code?
2. What is machine code?
3. What is Cross compiler?
4. Give the example for cross compiler?
5. What is the difference between syntax & Semantics?

**VIVA QUESTIONS**

1. What is Abstract Syntax tree?
2. What are BNF Rules?
3. What is DAG representation?
4. How LALR(1) states are generates?
5. In which condition the user has to supply more information to YACC?

**8.2.6**

Write a program to generate machine code from the abstract syntax tree generated by the Parser .The following instruction set may considered as target code.

**PREAMBLE** :

**code generation** is the process by which a [compiler](http://en.wikipedia.org/wiki/Compiler)'s **code generator** converts some [intermediate representation](http://en.wikipedia.org/wiki/Intermediate_representation) of [source code](http://en.wikipedia.org/wiki/Source_code) into a form (e.g., [machine code](http://en.wikipedia.org/wiki/Machine_code)) that can be readily executed by a machine.

Sophisticated compilers typically perform multiple passes over various intermediate forms. This multi-stage process is used because many [algorithms](http://en.wikipedia.org/wiki/Algorithm) for [code optimization](http://en.wikipedia.org/wiki/Code_optimization) are easier to apply one at a time, or because the input to one optimization relies on the completed processing performed by another optimization. This organization also facilitates the creation of a single compiler that can target multiple architectures, as only the last of the code generation stages (the *backend*) needs to change from target to target. (For more information on compiler design, see [Compiler](http://en.wikipedia.org/wiki/Compiler).)

The input to the code generator typically consists of a [parse tree](http://en.wikipedia.org/wiki/Parse_tree) or an [abstract syntax tree](http://en.wikipedia.org/wiki/Abstract_syntax_tree). The tree is converted into a linear sequence of instructions, usually in an [intermediate language](http://en.wikipedia.org/wiki/Intermediate_language) such as [three address code](http://en.wikipedia.org/wiki/Three_address_code). Further stages of compilation may or may not be referred to as "code generation", depending on whether they involve a significant change in the representation of the program. (For example, a [peephole optimization](http://en.wikipedia.org/wiki/Peephole_optimization) pass would not likely be called "code generation", although a code generator might incorporate a peephole optimization pass.)

**AIM** :

Write program to generate machine code from the abstract syntax tree generated by the Parser .The following instruction set may considered as target code.

**THEOREY:**

In addition to the basic conversion from an intermediate representation into a linear sequence of machine instructions, a typical code generator tries to optimize the generated code in some way.

Tasks which are typically part of a sophisticated compiler's "code generation" phase include:

* [Instruction selection](http://en.wikipedia.org/wiki/Instruction_selection): which instructions to use.
* [Instruction scheduling](http://en.wikipedia.org/wiki/Instruction_scheduling): in which order to put those instructions. Scheduling is a speed optimization that can have a critical effect on [pipelined](http://en.wikipedia.org/wiki/Instruction_pipeline) machines.
* [Register allocation](http://en.wikipedia.org/wiki/Register_allocation): the allocation of [variables](http://en.wikipedia.org/wiki/Variable_%28programming%29) to [processor registers](http://en.wikipedia.org/wiki/Processor_register)[[1]](http://en.wikipedia.org/wiki/Code_generation_%28compiler%29#cite_note-ASU-1)
* [Debug data](http://en.wikipedia.org/wiki/Debugging_data_format) generation if required so the code can be [debugged](http://en.wikipedia.org/wiki/Debugging).

Instruction selection is typically carried out by doing a [recursive](http://en.wikipedia.org/wiki/Recursion)[postorder traversal](http://en.wikipedia.org/wiki/Postorder_traversal) on the abstract syntax tree, matching particular tree configurations against templates; for example, the tree W := ADD(X,MUL(Y,Z)) might be transformed into a linear sequence of instructions by recursively generating the sequences for t1 := X and t2 := MUL(Y,Z), and then emitting the instruction ADD W, t1, t2.

In a compiler that uses an intermediate language, there may be two instruction selection stages — one to convert the parse tree into intermediate code, and a second phase much later to convert the intermediate code into instructions from the [instruction set](http://en.wikipedia.org/wiki/Instruction_set) of the target machine. This second phase does not require a tree traversal; it can be done linearly, and typically involves a simple replacement of intermediate-language operations with their corresponding [opcodes](http://en.wikipedia.org/wiki/Opcode). However, if the compiler is actually a [language translator](http://en.wikipedia.org/wiki/Transcompiler) (for example, one that converts [Eiffel](http://en.wikipedia.org/wiki/Eiffel_%28programming_language%29) to [C](http://en.wikipedia.org/wiki/C_%28programming_language%29)), then the second code-generation phase may involve *building* a tree from the linear intermediate code.

**PROCEDURE:**

Step1: Start

Step2: for every three –address statement of the form x=y op z

Step3: begin

Step4: Call getreg() to obtain the location L which the computation y op z should be performed

Step5: Obtain the current location of the operand y by consulting its address descriptor ,and if the values of Y are currently both in the memory location as well as in the register, then prefer the register.If the value of y is not currently available in 1,then generate an instruction MOV y,l

Step6: Generate the instruction OP Z,l and update the address descriptor of X to indicate that X is now available in l and in register then update t\ its descriptor to indicate that it will contain the run time value of x

Step7: If the current values of y ad/or z are in register and we have no further use for them,and they are live at the end of the block,then after the register descriptor to indicate that after the execution of the statem,ent x=y opz,those registers will no longer contain y and / or z.

Step8: store all results

Step9: Stop

**RESULT :**

{

inta,b,c;

if(a<b)

{

a=a+b;

}

while(a<b)

{

a=a+b;

}

if(a<=b)

{

c=a-b;

}

else

{

c=a+b;

Output:

$lexint.l

$yacc–d int.y

$gcclex.yy.cy.tab.c –ll–lm

$./a.outtest.c

**APPLICATION**

**Abstract syntax tree is used to check the syntax of expressions while determining tokens in compiler construction**

**SAMPLE QUESTIONS**

1. What is Abstract Syntax tree?
2. What are BNF Rules?
3. What is DAG representation?
4. How LALR(1) states are generates?
5. In which condition the user has to supply more information to YACC?

**VIVA QUESTIONS**

1. What is target code?
2. What is machine code?
3. What is Cross compiler?
4. Give the example for cross compiler?
5. What is the difference between syntax & Semantics?