

EXPERIMENT NO: 01

Title: Study of Lexical Analyzer Generator
Aim: Study of Lexical Analyzer Generator

Theory:

❖ Introduction of Lexical Analysis-

The first phase of a compiler is called lexical analysis (and is also known as a lexical scanner). As implied by its name, lexical analysis attempts to isolate the “words” in an input string. We use the word “word” in a technical sense. A word, also known as a lexeme, a lexical item, or a lexical token, is a string of input characters which is taken as a unit and passed on to the next phase of compilation.

Examples of words are

1. Key words - while, void, if, for
2. Identifiers-declared by the programmer
3. Operators +, -, *,/, =,==
4. Numeric constants-numbers such as 124, 12.35, 0.09E-23, etc.
5. Character constants- single characters or strings of characters enclosed in quotes.
6. Special characters- characters used as delimiters such as ()

Lexical Analysis is the very first phase in the compiler designing. A Lexer takes the modified source code which is written in the form of sentence. In other words, it helps you to convert a sequence of characters into a sequence of tokens. The lexical analyzer breaks this syntax into a series of tokens. It removes any extra space or comment written in the source code. Programs that perform Lexical Analysis in compiler design are called lexical analyzers or lexers. A lexer contains a tokenizer or scanner. If the lexical analyzer detects that the token is invalid, it generates an error. The role of Lexical Analyzer in compiler design is to read character streams from the source code, check for legal tokens, and pass the data to the syntax analyzer when it demands.

❖ **What is a token?**

Tokens in compiler design are the sequence of characters which represents a unit of information in the source program

Example of tokens:

1. Type token (id, number, real, . . .)
2. Punctuation tokens (IF, void, return, . . .)
3. Alphabetic tokens (keywords)

Example of Non-Tokens:

Comments, preprocessor directive, macros, blanks, tabs, newline, etc.

❖ **What is lexeme?**

A lexeme is a sequence of characters that are included in the source program according to the matching pattern of a token. It is nothing but an instance of a token.

e.g- “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;”

❖ **What is a Pattern:?**

A set of strings described by rule called patterns

Roles of the Lexical analyzer

Lexical analyzer performs below given tasks:

- Helps to identify token into the symbol table
- Removes white spaces and comments from the source program
- Correlates error messages with the source program
- Helps you to expands the macros if it is found in the source program
- Read input characters from the source program

❖ Working of LEX:

A Lex program is separated into three sections by %% delimiters. The format of Lex source is as

1. {Definitions}
2. %%
3. {Rules}
4. %%
5. {User subroutines}

1. Definitions include declarations of constant, variable and regular definitions.

2. Rules define the statement of form `p1{action1}p2{action2}.....pn{action}`.

3. User subroutines are auxiliary procedures needed by the actions.

The subroutine can be loaded with the lexical analyzer and compiled separately.

Representing valid tokens of a language in regular expression.

1. If x is a regular expression, then:

x^* means zero or more occurrence of x . (0.1.2...)

i.e., it can generate $\{e, x, xx, xxx, xxxx, \dots\}$

2. x^+ means one or more occurrence of x .

i.e., it can generate $\{x, xx, xxx, xxxx, \dots\}$ or $x.x^*$

3. $x^?$ Means at most one occurrence of x

i.e., it can generate either $\{x\}$ or $\{e\}$.

4. $[a-z]$ is all lower-case alphabets of English language.

5. $[A-Z]$ is all upper-case alphabets of English language.

6. $[0-9]$ is all natural digits used in mathematics.

Conclusion :

Thus we studied Lexical Analyzer Generator

Sample Questions:

- 1) What is the role of lexical analyzer?
- 2) Explain Lexical analysis
- 3) What are patterns, lexemes, tokens?
- 4) Explain 3 parts of the lex program
- 5) What are the Lex specifications?

EXPERIMENT NO: 02**Title:** Lex Program for Token Separation**Aim:** Lex Program for Token Separation**Algorithm:**

1. First, a specification of a lexical analyzer is prepared by creating a program lex.l in the LEX language.
 2. The Lex.l program is run through the LEX compiler to produce an equivalent code in C language named Lex.yy.c3.
- The program lex.yy.c consists of a table constructed from the Regular Expressions of Lex.l, together with standard routines that uses the table to recognize lexemes.
4. Finally, lex.yy.c program is run through the C Compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

Sample Code:**Token separation**

```
% {
#include<stdio.h>
% }

%%

[0-9]+[.][0-9]+ printf("%s is a floating point number\n",yytext);

int|float|char|double|voidprintf("%s is a datatype\n",yytext);

[0-9]+ printf("%s is an integer number\n",yytext);

[a-z]+[()] printf("%s is a function\n",yytext);

[a-z]+ printf("%s is an identifier\n",yytext);

[+*=/-] printf("%s is an operator\n",yytext);

printf("%s is an delimiter\n",yytext);

printf("%s is a separator\n",yytext);

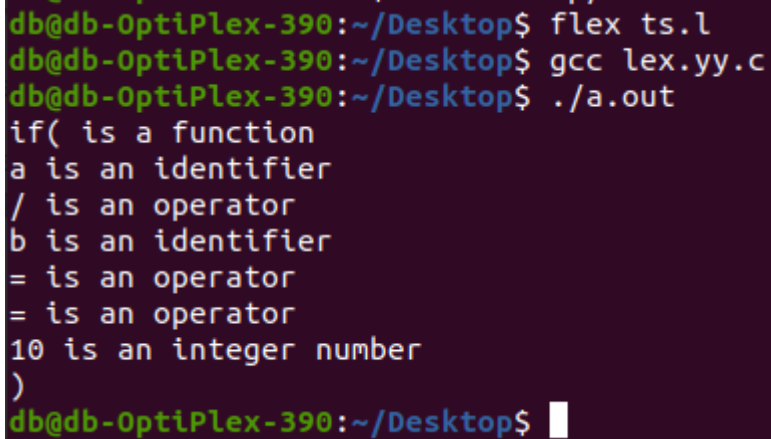
[#][a-z\.h]+ printf("%s is a preprocessor\n",yytext);
%%
```

```
int yywrap(void) { }

int main()
{
// reads input from a file named test.c rather than terminal
freopen("test.c", "r", stdin);
    yylex();
    return 0;
}
```

INPUT= if (a/b==10)

OUTPUT=



```
db@db-OptiPlex-390:~/Desktop$ flex ts.l
db@db-OptiPlex-390:~/Desktop$ gcc lex.yy.c
db@db-OptiPlex-390:~/Desktop$ ./a.out
if( is a function
a is an identifier
/ is an operator
b is an identifier
= is an operator
= is an operator
10 is an integer number
)
db@db-OptiPlex-390:~/Desktop$
```

Conclusion :

Thus we learn token separation using lex tools

Sample Question:

1. What is mean by Lexical Analysis?
2. What is mean by Token?

EXPERIMENT NO: 03

Title: Lex program to implement simple calculator.
Aim: Lex program to implement simple calculator.

Theory:

Procedure:

- 1) Get the input from the user and Parse it token by token.
- 2) First identify the valid inputs that can be given for a program.
- 3) The Inputs include numbers, functions like LOG, COS, SIN, TAN, etc.and operators.
- 4) Define the precedence and the associability of various operators like +,-,/,*etc.
- 5) Write codes for saving the answer into memory and displaying the result on the screen.
- 6) Write codes for performing various arithmetic operations.
- 7) Display the output on the screen else display the error message on the screen.

Sample code:**Simple calculator**

```
% {
int op = 0,i;
float a, b;
% }

dig [0-9]+|([0-9]*)."([0-9]+)
add "+"
sub "-"
mul "*"
div "/"
pow "^"
ln \n
%%

{ dig } { digi();}
{ add } { op=1;}
{ sub } { op=2;}
{ mul } { op=3;}
{ div } { op=4;}
{ pow } { op=5;}
{ ln } { printf("\n The Answer :%f\n\n",a);}

%%
digi()
{
if(op==0)

a=atof(yytext);

else
{
b=atof(yytext);

switch(op)
{
case 1:a=a+b;
break;

case 2:a=a-b;
break;

case 3:a=a*b;
```

```

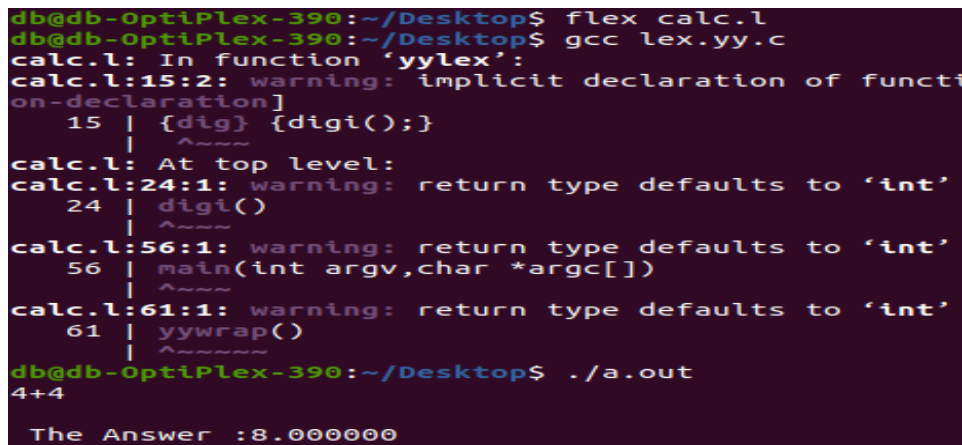
break;

case 4:a=a/b;
break;

case 5:for(i=a;b>1;b--)
a=a*i;
break;
}
op=0;
}
}
main(intargv,char *argc[])
{
yylex();
}

yywrap()
{
return 1;
}

```

Output:


```

db@db-OptiPlex-390:~/Desktop$ flex calc.l
db@db-OptiPlex-390:~/Desktop$ gcc lex.yy.c
calc.l: In function 'yylex':
calc.l:15:2: warning: implicit declaration of function 'yywrap' [-Wimplicit-declaration]
   15 | {dig} {digi();}
      | ^~~~~
calc.l: At top level:
calc.l:24:1: warning: return type defaults to 'int' [-Wimplicit-int]
   24 | digi()
      | ^~~~~
calc.l:56:1: warning: return type defaults to 'int' [-Wimplicit-int]
   56 | main(int argv,char *argc[])
      | ^~~~~
calc.l:61:1: warning: return type defaults to 'int' [-Wimplicit-int]
   61 | yywrap()
      | ^~~~~~
db@db-OptiPlex-390:~/Desktop$ ./a.out
4+4
The Answer :8.000000

```

Conclusion :

Thus we implement the LEX program to implement calculator.

Sample Question:

1. Give the structure of lex program
2. Explain lex tools
3. Explain the variable yylex in the program
4. What is lexical analyzer?

EXPERIMENT NO: 04

Title: Lex program to identify consonants & vowels in word.
Aim: Lex program to identify consonants & vowels in word.

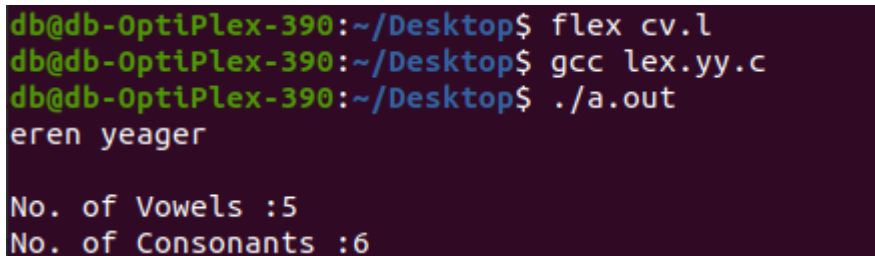
Theory:

Procedure:

1. Create a lex specification file to recognize single line and multiple comment statements
2. Compile it by using LEX compiler to get 'C' file
3. Now compile the c file using C compiler to get executable file
4. Run the executable file to get desired output by providing necessary input
5. Another method is to compile the lex program by using flex tool

Sample Code:**Consonants and vowels**

```
% {  
#include<stdio.h>  
int vcount=0,ccount=0;  
% }  
%%  
[a|i|e|o|u|E|A|I|O|U] {vcount++;}  
[a-zA-Z (^a|i|e|o|u|E|A|I|O|U) ] {ccount++;}  
%%  
int yywrap(void){}  
int main()  
{  
yylex();  
printf("No. of Vowels :%d\n",vcount);  
printf("No. of Consonants :%d\n",ccount);  
return 0;  
}
```

Output:

```
db@db-OptiPlex-390:~/Desktop$ flex cv.l  
db@db-OptiPlex-390:~/Desktop$ gcc lex.yy.c  
db@db-OptiPlex-390:~/Desktop$ ./a.out  
eren yeager  
  
No. of Vowels :5  
No. of Consonants :6
```

Conclusion:

Thus we implement the LEX program to count consonants & vowels.

Sample Question:

1. Give the structure of lex program
2. Explain lex tools
3. Explain the variable yylex in the program
4. What is lexical analyzer?

EXPERIMENT NO: 05

Title: Lex program to count number of words.
Aim: Lex program to count number of words.

Theory:

Procedure:

1. Take a string as input.
2. using for loop search for a empty space in between the words in the string.
3. Consecutively increment a variable. This variable gives the count of number of words.

Sample Code:

NUMBER OF WORDS

```
/*lex program to count number of words*/
% {
#include<stdio.h>
#include<string.h>
int i = 0;
% }
/* Rules Section*/
%%
([a-zA-Z0-9])* {i++;} /* Rule for counting number of words*/

"\n" {printf("%d\n", i); i = 0;}
%%
int yywrap(void){}
int main()
{
// The function that starts the analysis
    yylex();
return 0;
}
```

Output:

```
db@db-OptiPlex-390:~/Desktop$ flex no.l
db@db-OptiPlex-390:~/Desktop$ gcc lex.yy.c
db@db-OptiPlex-390:~/Desktop$ ./a.out
RIP Chadwik Boseman Wakanda Forever
5
```

Conclusion:

Thus we implement the LEX program to count number of words.

Sample Question:

1. Give the structure of lex program
2. Explain lex tools
3. Explain the variable yylex in the program
4. What is lexical analyzer?

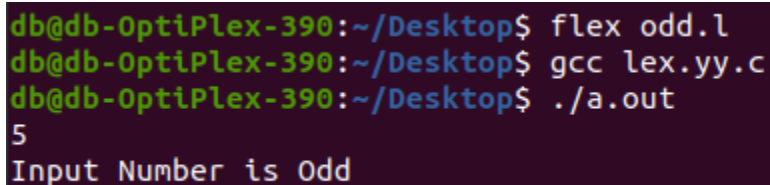
EXPERIMENT NO: 06**Title:** Lex program to find given number is odd or even.**Aim:** Lex program to find given number is odd or even.**Theory:****Procedure:**

Use following procedure to write lex program

- Step 1: Start
- Step 2: Read a number to N
- Step 3: Divide the number by 2 and store the remainder in R.
- Step 4: If R = 0 Then go to Step 6
- Step 5: Print "N is odd" go to step 7
- Step 6: Print "N is even"
- Step 7: Stop

Sample Code:**Odd or Even**

```
% {
int i;
% }
%%
[0-9]+ {i = atoi(yytext);
if(i%2==0)
printf("Input Number is Even");
else
printf("Input Number is Odd");
};
%%
int main()
{
    yylex();
    return 1;
}
```

Output:

```
db@db-OptiPlex-390:~/Desktop$ flex odd.l
db@db-OptiPlex-390:~/Desktop$ gcc lex.yy.c
db@db-OptiPlex-390:~/Desktop$ ./a.out
5
Input Number is Odd
```

Conclusion:

Thus we implement the LEX program to determine given number is odd or even.

Sample Question:

1. Give the structure of lex program
2. Explain lex tools
3. Explain the variable yylex in the program
4. What is lexical analyzer?

7. Case Study on Assembler

Theory:

Assembler is a program for converting instructions written in low-level assembly code into relocatable machine code and generating along information for the loader.



It generates instructions by evaluating the mnemonics (symbols) in operation field and find the value of symbol and literals to produce machine code. Now, if assembler do all this work in one scan then it is called single pass assembler, otherwise if it does in multiple scans then called multiple pass assembler. Here assembler divides these tasks in two passes:

- **Pass-1:**

1. Define symbols and literals and remember them in symbol table and literal table respectively.
2. Keep track of location counter
3. Process pseudo-operations

- **Pass-2:**

1. Generate object code by converting symbolic op-code into respective numeric op-code
2. Generate data for literals and look for values of symbols

Firstly, We will take a small assembly language program to understand the working in their respective passes. Assembly language statement format:

[Label] [Opcode] [operand]

Example: M ADD R1, ='3'

where, M - Label; ADD - symbolic opcode;

R1 - symbolic register operand; ('=3') - Literal

Assembly Program:

Label Op-code operand LC value(Location counter)

JOHN START 200

```

    MOVER  R1, ='3'  200
    MOVEM  R1, X     201
L1  MOVER  R2, ='2'  202
    LTORG                203
X   DS     1         204
    END                205

```

Let's take a look on how this program is working:

1. **START:** This instruction starts the execution of program from location 200 and label with START provides name for the program.(JOHN is name for program)
2. **MOVER:** It moves the content of literal(='3') into register operand R1.
3. **MOVEM:** It moves the content of register into memory operand(X).
4. **MOVER:** It again moves the content of literal(='2') into register operand R2 and its label is specified as L1.
5. **LTORG:** It assigns address to literals(current LC value).
6. **DS(Data Space):** It assigns a data space of 1 to Symbol X.
7. **END:** It finishes the program execution.

Working of Pass-1: Define Symbol and literal table with their addresses.

Note: Literal address is specified by LTORG or END.

Step-1: START 200 (here no symbol or literal is found so both table would be empty)

Step-2: MOVER R1, ='3' 200 (='3' is a literal so literal table is made)

Literal	Address
= '3'	---

Step-3: MOVEM R1, X 201

X is a symbol referred prior to its declaration so it is stored in symbol table with blank address field.

Symbol	Address
X	---

Step-4: L1 MOVER R2, ='2' 202

L1 is a label and ='2' is a literal so store them in respective tables

Symbol	Address
X	---

Symbol	Address
L1	202
Literal	Address
= '3'	---
= '2'	---

Step-5: LTORG 203

Assign address to first literal specified by LC value, i.e., 203

Literal	Address
= '3'	203
= '2'	---

Step-6: X DS 1 204

It is a data declaration statement i.e X is assigned data space of 1. But X is a symbol which was referred earlier in step 3 and defined in step 6. This condition is called Forward Reference Problem where variable is referred prior to its declaration and can be solved by back-patching. So now assembler will assign X the address specified by LC value of current step.

Symbol	Address
X	204
L1	202

Step-7: END 205

Program finishes execution and remaining literal will get address specified by LC value of END instruction. Here is the complete symbol and literal table made by pass 1 of assembler.

Symbol	Address
X	204

Symbol	Address
L1	202
Literal	Address
= '3'	203
= '2'	205

Now tables generated by pass 1 along with their LC value will go to pass-2 of assembler for further processing of pseudo-opcodes and machine op-codes.

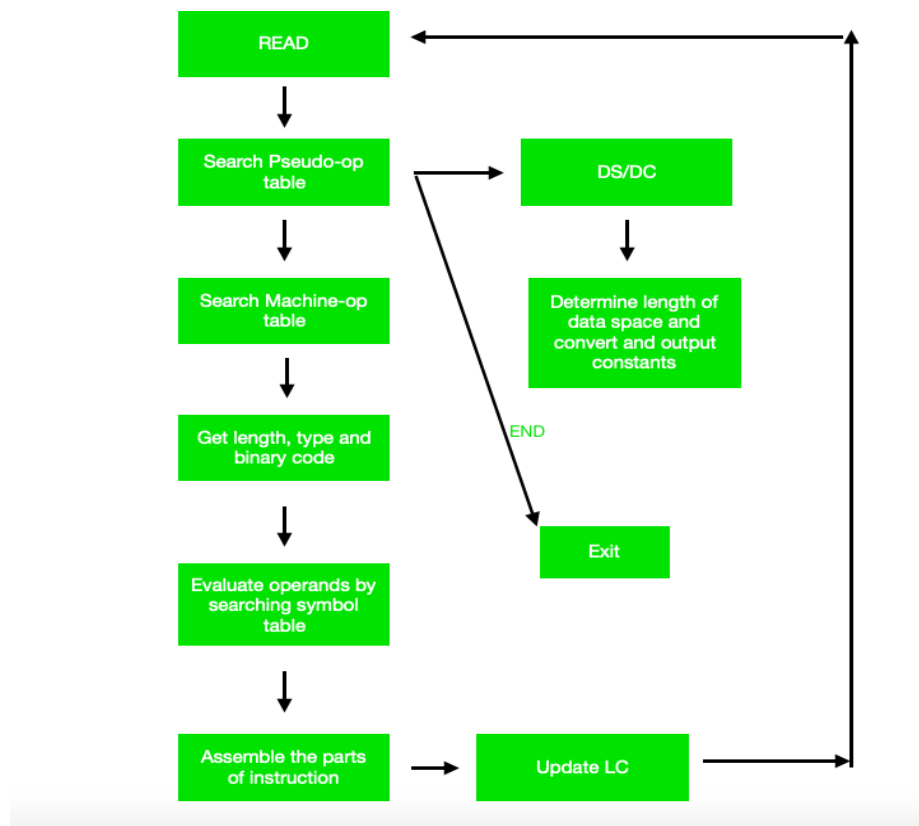
Working of Pass-2:

Pass-2 of assembler generates machine code by converting symbolic machine-opcodes into their respective bit configuration(machine understandable form). It stores all machine-opcodes in MOT table (op-code table) with symbolic code, their length and their bit configuration. It will also process pseudo-ops and will store them in POT table(pseudo-op table).

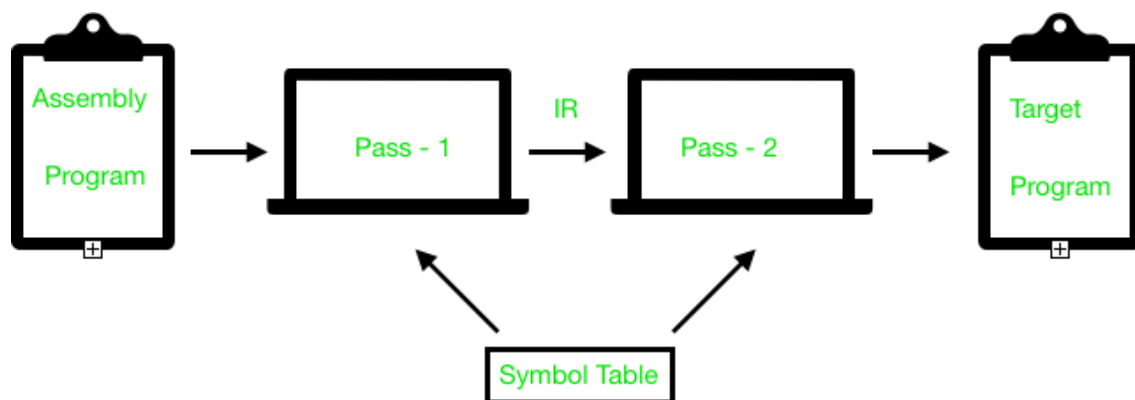
Various Data bases required by pass-2:

1. MOT table(machine opcode table)
2. POT table(pseudo opcode table)
3. Base table(storing value of base register)
4. LC (location counter)

Take a look at flowchart to understand:



As a whole assembler works as:



Viva voice:

1. Define the basic functions of assembler
2. What is mean by assembler directives?
3. What is forward reference?
4. What are the tables used in assembler
5. What is pass1 and pass2 assembler?
5. What is the need of symbol table(SYMTAB) in assembler?

8. Case Study on Macro preprocessor

Theory:

Introduction

Macros are used to provide a program generation facility through macro expansion. Many languages provide build-in facilities for writing macros like PL/I, C, Ada AND C++. Assembly languages also provide such facilities. When a language does not support build-in facilities for writing macros what is to be done? A programmer may achieve an equivalent effect by using generalized preprocessors or software tools like Awk of Unix.

A macro definition:

A macro is a unit of specification for program generation through expansion. A macro consists of a name, a set of formal parameters and a body of code. The use of a macro name with a set of actual parameters is replaced by some code generated from its body. This is called macro expansion. Two kinds of expansion can be identified.

Classification of macros:

Lexical expansion:

Lexical expansion implies replacement of a character String by another character string during program generation. Lexical expansion is to replace occurrences of formal parameters by corresponding actual parameters.

Semantic expansion:

Semantic expansion implies generation of instructions tailored to the requirements of a specific usage. Semantic expansion is characterized by the fact that different uses of a macro can lead to codes which differ in the number, sequence and opcodes of instructions. Eg: Generation of type specific instructions for manipulation of byte and word operands.

Example

The following sequence of instructions is used to increment the value in a memory word by a constant. 1. Move the value from the memory word into a machine register.

2. Increment the value in the machine register.

3. Move the new value into the memory word. Since the instruction sequence MOVE-ADD-MOVE may be used a number of times in a program, it is convenient to define a macro named INCR. Using Lexical expansion the macro call INCR A, B, AREG can lead to the generation of a MOVEADD-MOVE instruction sequence to increment A by the value of B using AREG to perform the arithmetic. Use of Semantic expansion can enable the instruction sequence to be adapted to the types of A and B. For example an INC instruction could be generated if A is a byte operand and B has the value

Macro definition and call

A macro definition is enclosed between a macro header statement and a macro end statement. Macro definitions are typically located at the start of a program. A macro definition consists of. A macro prototype statement, One or more model statements, Macro preprocessor statements, the macro prototype statement declares the name of a macro and the names and kinds of its parameters. It has the following syntax

<macro name> [< formal parameter spec > [...]]

Where <macro name> appears in the mnemonic field of an assembly statement and

< formal parameter spec> is of the form

&<parameter name> [<parameter kind>]

Nested macro calls

Macro calls appearing in the source program have been expanded but statements resulting from the expansion may themselves contain macro calls. The macro expansion can be applied until we get the code form which does not contain any macro call statement. Such expansion requires a number of passes of macro expansion. To increase the efficiency, another alternative would be to examine each statement generated during macro expansion to see if it is itself a macro call. If so, provision can be made to expand this call before continuing with the expansion of the parent macro call. This avoids multiple passes of macro expansion.

Design of a macro preprocessor

The macro preprocessor accepts an assembly program containing definitions and calls and translates it into an assembly program which does not contain any macro definitions and calls. The program form output by the macro preprocessor can be handed over to an assembler to obtain the target program.

Design overview

We begin the design by listing all tasks involved in macro expansion.

1. Identify macro calls in the program.
2. Determine the values of formal parameters.
3. Maintain the values of expansion time variables declared in a macro.
4. Organize expansion time control flow.
5. Determine the values of sequencing symbols.
6. Perform expansion of a model statement.

Following 4 step procedures is followed to arrive at a

1. Design specification for each task. Identify the information necessary to perform a task.
2. Design a suitable data structure to record the information.
3. Determine the processing necessary to obtain the information.
4. Determine the processing necessary to perform the task.

Tables are constructed for macro preprocessor.

Table	Fields
MNT (Macro Name Table)	Macro Name
	Number of Positional Parameter (#PP)
	Number of keyword parameter (#KP)
	Number of Expansion Time Variable (#EV)
	MDT pointer (MDTP)
	KPD TAB pointer (KPD TABP)
	SSTAB pointer (SSTP)

Table	Fields
PNTAB (Parameter Name Table)	Parameter name
EVNTAB (EV Name Table)	EV Name
SSNTAB (SS Name Table)	SS Name
KPD TAB (Keyword Parameter Default Table)	Parameter name, default value
MDT (Macro Definition Table)	Label, Opcodes, Operands Value
APTAB (Actual Parameter Table)	Value
EVTAB (EV Table)	Value
SSTAB (SS Table)	MDT entry #

Viva voice:

1. Define Macro preprocessor
2. What is a nested macro call
3. What are the data structures used in macro preprocessor
4. What is the use of macro time variable?
5. What is mean by positional parameters?

9. Case Study on Linker and Loader

Theory:

Introduction

The Source Program written in assembly language or high level language will be converted to object program, which is in the machine language form for execution. This conversion either from assembler or from compiler, contains translated instructions and data values from the source program, or specifies addresses in primary memory where these items are to be loaded for execution.

This contains the following three processes, and they are,

Loading - which allocates memory location and brings the object program into memory for execution - (Loader)

Linking- which combines two or more separate object programs and supplies the information needed to allow references between them - (Linker)

Relocation - which modifies the object program so that it can be loaded at an address different from the location originally specified - (Linking Loader)

Types:

1.Static Linking

It is performed during the compilation of source program. Linking is performed before execution in static linking. It takes collection of relocatable.It object file and command-line argument and generate fully linked object file that can be loaded and run.

Static linkers perform two major tasks:

- **Symbol resolution** – It associates each symbol reference with exactly one symbol definition .Every symbol have predefined task.
- **Relocation** – It relocates code and data section and modifies symbol references to the relocated memory location.

The linker copy all library routines used in the program into executable image. As a result, it require more memory space. As it does not require the presence of library on the system when it is run . so, it is faster and more portable. No failure chance and less error chance.

2. Dynamic linking –

Dynamic linking is performed during the run time. This linking is accomplished by placing the name of a shareable library in the executable image. There are more chances of error and failure chances. It requires less memory space as multiple programs can share a single copy of the library. Here we can perform code sharing. it means we are using a same object a number of times in the program. Instead of linking same object again and again into the library, each module share information of an object with other module having same object. The shared library needed in the

Linking is stored in virtual memory to save RAM. In this linking we can also relocate the code for the smooth running of code but all the code is not relocatable. It fixes the address at run time.

Loader:

The loader is special program that takes input of object code from linker, loads it to main memory, and prepares this code for execution by computer. Loader allocates memory space to program. Even it settles down symbolic reference between objects. It is in charge of loading programs and libraries in operating system. The embedded computer systems don't have loaders. In them, code is executed through ROM. There are following various loading schemes:

1. Absolute Loaders
2. Relocating Loaders
3. Direct Linking Loaders
4. Bootstrap Loaders

1. Absolute Loaders

Absolute loader is a kind of loader in which relocated object files are created, loader accepts these files and places them at specified locations in the memory. This type of loader is called absolute because no relocation information is needed; rather it is obtained from the programmer or assembler. The starting address of every module is known to the programmer, this corresponding starting address is stored in the object file, then task of loader becomes very simple and that is to simply place the executable form of the machine instructions at the locations mentioned in the object file. In this scheme the programmer or assembler should have knowledge of memory management. The resolution of external references or linking of different subroutines are the issues which need to be handled by the programmer

2. Relocating Loaders

The relocating loader will load the program anywhere in memory, altering the various addresses as required to ensure correct referencing. The decision as to where in memory the program is placed is done by the Operating System, not the programs header file. This is obviously more efficient, but introduces a slight overhead in terms of a small delay whilst all the relative offsets are calculated. The relocating loader can only relocate code that has been produced by a linker capable of producing relative code.

4. Bootstrap Loaders

Alternatively referred to as bootstrapping, boot loader, or boot program, a bootstrap loader is a program that resides in the computer's EPROM, ROM, or another non-volatile memory. It is automatically executed by the processor when turning on the computer. The bootstrap loader reads the hard drives boot sector to continue to load the computer's operating system. The term bootstrap comes from the old phrase "Pull yourself up by your bootstraps."

When the computer is turned on or restarted, the bootstrap loader first performs the power-on self-test, also known as POST. If the POST is successful and no issues are found, the bootstrap loader loads the operating system for the computer into memory. The computer can then access, load, and run the operating system.

The bootstrap loader was replaced in computers that have an EFI (Extensible Firmware Interface) and is now part of the EFI BIOS.

Viva Voice

1. What is the type of loader?
2. What are the basic functions of loader?
3. What is relocation loader?
4. What is absolute loader?
5. What is linker and its function
6. What is relocation?

10.Case Study on Assembler

Theory

A compiler is a computer program that translates computer code written in one programming language (the source language) into another language (the target language). The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g. assembly language, object code, or machine code) to create an executable program.

- A compiler is a translator that converts the high-level language into the machine language.
- High-level language is written by a developer and machine language can be understood by the processor.
- Compiler is used to show errors to the programmer.
- The main purpose of compiler is to change the code written in one language without changing the meaning of the program.
- When you execute a program which is written in HLL programming language then it executes into two parts.
- In the first part, the source program compiled and translated into the object program (low level language).
- In the second part, object program translated into the target program through the assembler

Aspects of compilation

Four Steps of Compilation: preprocessing, compiling, assembly, linking.

Preprocessing:

Preprocessing is the first step. The preprocessor obeys commands that begin with # (known as directives) by:

- removing comments
- expanding macros
- expanding included files

If you included a header file such as `#include <stdio.h>`, it will look for the `stdio.h` file and copy the header file into the source code file.

The preprocessor also generates macro code and replaces symbolic constants defined using `#define` with their values.

Compiling:

Compiling is the second step. It takes the output of the preprocessor and generates assembly language, an intermediate human readable language, specific to the target processor.

Assembly:

Assembly is the third step of compilation. The assembler will convert the assembly code into pure binary code or machine code (zeros and ones). This code is also known as object code.

Linking:

Linking is the final step of compilation. The linker merges all the object code from multiple modules into a single one. If we are using a function from libraries, linker will link our code with that library function code.

In static linking, the linker makes a copy of all used library functions to the executable file. In dynamic linking, the code is not copied; it is done by just placing the name of the library in the binary file.

Memory Allocation:

Memory allocation is an action of assigning the physical or the virtual memory address space to a process (its instructions and data). The two fundamental methods of memory allocation are static and dynamic memory allocation. To get a process executed it must be first placed in the memory. Assigning space to a process in memory is called memory allocation. Memory allocation is a general aspect of the term binding. Static memory allocation method assigns the memory to a process, before its execution. On the other hand, the dynamic memory allocation method assigns the memory to a process, during its execution. In this section, we will be discussing what is memory allocation, its types (static and dynamic memory allocation) along with their advantages and disadvantages. So let us start.

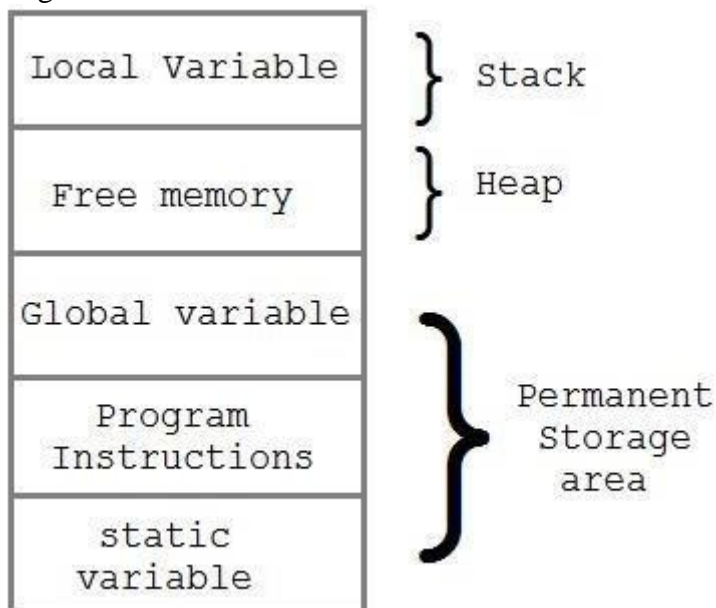
Memory allocation has two core types;

- Static Memory Allocation: The program is allocated memory at compile time.
- Dynamic Memory Allocation: The programs are allocated with memory at run time.

Static Memory Allocation

In general, static memory allocation is the allocation of memory at compile time, before the associated program is executed, unlike dynamic memory allocation or automatic memory allocation where memory is allocated as required at run time.

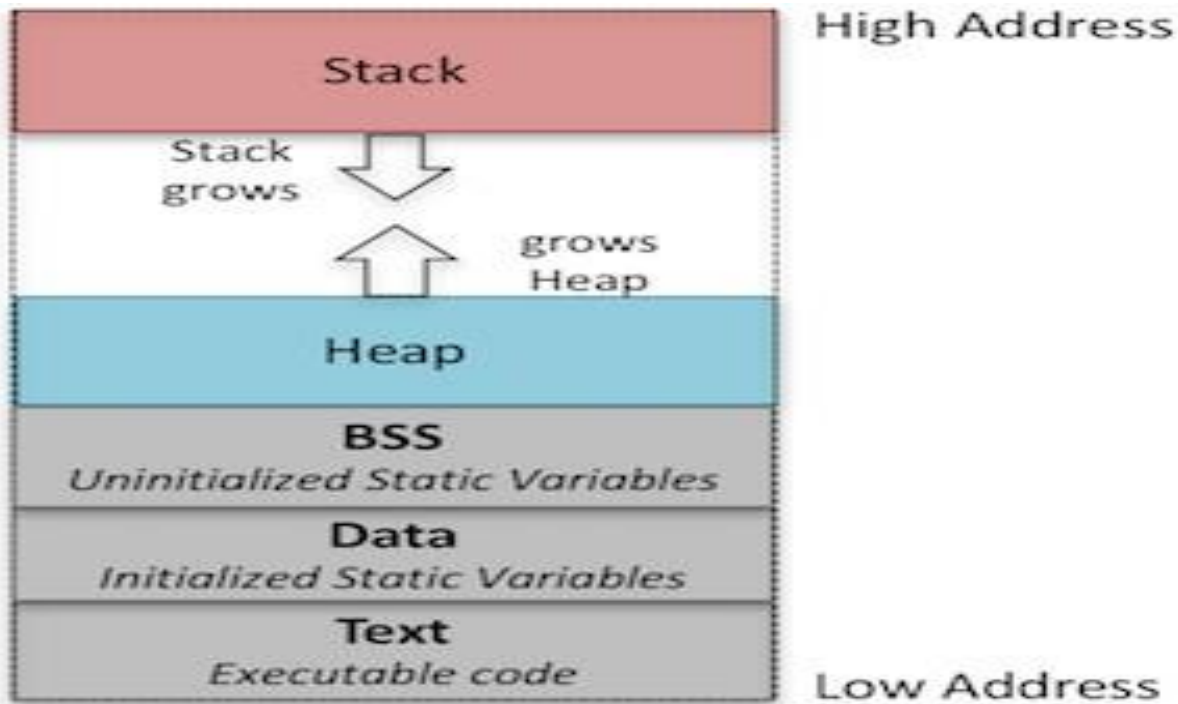
Diagram



Dynamic Memory Allocation

Dynamic memory allocation is the process of assigning the memory space during the execution time or the run time. Reasons and Advantage of allocating memory dynamically: When we do not know how much amount of memory would be needed for the program beforehand

Diagram



Memory allocation in block structured languages

- A block is a program segment that contains data declarations
- There can be nested blocks.
- Uses dynamic memory allocation.

Code optimization

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result. Compiler optimizing process should meet the following objectives:

- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.

Local Optimization

A local optima is the extrema (minimum or maximum) of the objective function for a given region of the input space, e.g. a basin in a minimization problem. An objective function may have many local optima, or it may have single local optima, in which case the local optima are also the global optima. Local Optimization: Locate the optima for an objective function from a starting point believed to contain the optima (e.g. a basin). Local optimization or local search refers to searching for the local optima. A local optimization algorithm, also called a

Local search algorithm is an algorithm intended to locate a local optima. It is suited to traversing a given region of the search space and getting close to (or finding exactly) the extrema of the function in that region

A local optimization algorithm will locate the global optimum:

- If the local optima is the global optima, or
- If the region being searched contains the global optima.

These define the ideal use cases for using a local search algorithm. There may be debate over what exactly constitutes a local search algorithm; nevertheless, three examples of local search algorithms using our definitions include:

Nelder-Mead Algorithm
BFGS Algorithm
Hill-Climbing Algorithm

Global Optimization

A global optimum is the extrema (minimum or maximum) of the objective function for the entire input search space. An objective function may have one or more than one global optima, and if more than one, it is referred to as a multimodal optimization problem and each optimum will have a different input and the same objective function evaluation. Global Optimization: Locate the optima for an objective function that may contain local optima. An objective function always has a global optima (otherwise we would not be interested in optimizing it), although it may also have local optima that have an objective function evaluation that is not as good as the global optima. Global search algorithms may involve managing a single or a population of candidate solutions from which new candidate solutions are iteratively generated and evaluated to see if they result in an improvement and taken as the new working state. There may be debate over what exactly constitutes a global search algorithm; nevertheless, three examples of global search algorithms using our definitions include:

Genetic Algorithm
Simulated Annealing
Particle Swarm Optimization

Optimization techniques

- Optimization is a program transformation technique, which tries to improve the code by making it consume less resource (i.e. CPU, Memory) and deliver high speed.
- In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:
- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and should not delay the overall compiling process. Efforts for an optimized code can be made at various levels of compiling the process.
- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Optimization can be categorized broadly into two types: machine independent and machine dependent.

Interpreter

An interpreter is a computer program that is used to directly execute program instructions written using one of the many high-level programming languages.

The interpreter transforms the high-level program into an intermediate language that it then executes, or it could parse the high-level source code and then performs the commands directly, which is done line by line or statement by statement.

Viva Voice

1. What is Assembler?
2. What are techniques of memory allocation?
3. What are the techniques of code optimization?
4. What is Interpreter?