Chapter 4 Macro Processors.

A macro instruction is simply a notational convenience for programmer.
Macro represents commonly used group of statements in the source program
the macro processor replace each macro instruction with corresponding group of source statements.
- this operation is called 'expanding the macro'.
using macros allows programmer to write a shorthand version of a program.
eg:- before calling a subroutine, the contents of all registers may need to be stored this routine work can be done using a macro.

Macroprocessors.

its function essentially involves the substitution of one group of lines or another.
it doesn't analyze the text it handles.
meaning of the statements are of no concern during macro expansion.
Hence, design of macro processor generally is machine independent.
Macro mostly are used on assembly language programming, also be used in high level programming language such as C, C++.

Macro definition

Two directives MACRO and MEND are used in macro definition.
Macro's name appear before the MACRO directive.
Macro's parameters appear afters the MACRO directive.

→ each parameter begins with '&'
→ between MACRO and MEND is the body of macro. here these are the statements that will be generated as the expansion of the macro definition.

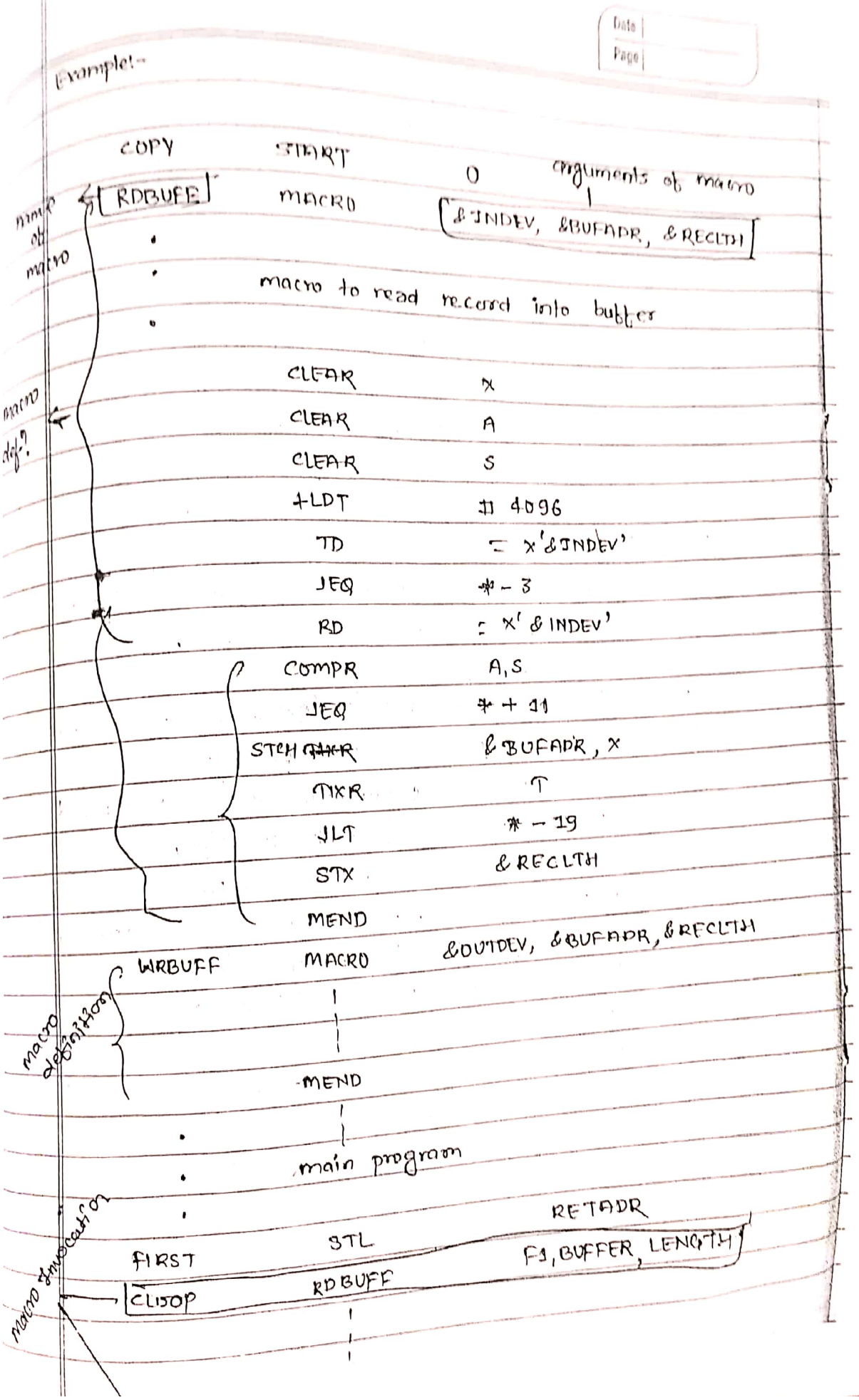
Macro Invocation

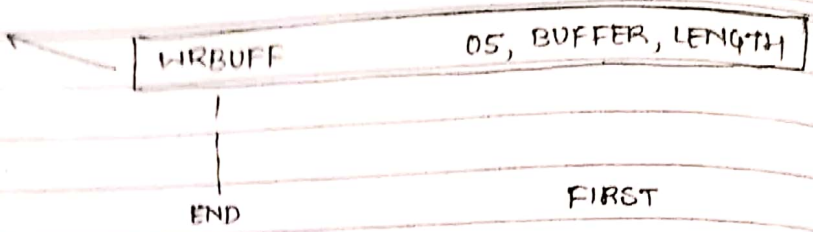
→ macro invocation statement (macro call) gives the name of the macro instruction being invoked and the arguments in expanding the macro.

Macro Expansion

→ each ~~more~~ macro invocation statement will be expanded into the statements that form the body of the macro.
→ arguments from the macro invocation are substituted for the parameters in macro prototype.
→ the arguments and parameters are associated with are another according to their positions
- the first argument in the macro invocation corresponds to the first parameter in macro prototype
→ Comment lines within the macro body have been deleted but comments on individual statements have been retained.
→ Macro invocation, statement itself has been included as a comment line.

Example:-





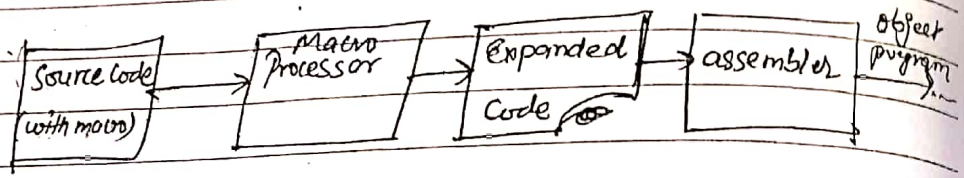
```
COPY    START   0
FIRST   STL     RETADR
CLOOP   RDBUFF  F1,BUFFER,LENGTH

CLOOP   CLEAR   X
        CLEAR   A
        CLEAR   S
        +LDT    #4096
        TD      =X'F1'
        JEQ     *-3
        RD      =X'F1'
        COMPR   A,S
        JEQ     *+11
        STCH    BUFFER,X
        TIXR    T
        JLT     *-19
        STX     LENGTH
```

Fig:- Macro Expression Example.

→ after macro processing the expanded file can be use as input to the assembler.

→ the statements generated from the macro expansions will be assembled exactly as they had been written directly by the programmer



Difference between Macro and Subroutine

→ Statements that form the body of macro are generated each time a macro is expanded.

→ Statements in subroutine appear only once, regardless of how many times times the subroutine is called.

Macro processor algorithm and data structures.

→ Two pass macro processor can be changed where
- all macro definition are processed during first pass.
- all macro invocations statements are expanded during second pass.
- two pass macro processor wouldn't allow the body of one macro instruction to contain definition of other ~~words~~ macros.
- because all macros would have to be defined during the first pass ~~define~~ before any macro invocation were expanded.

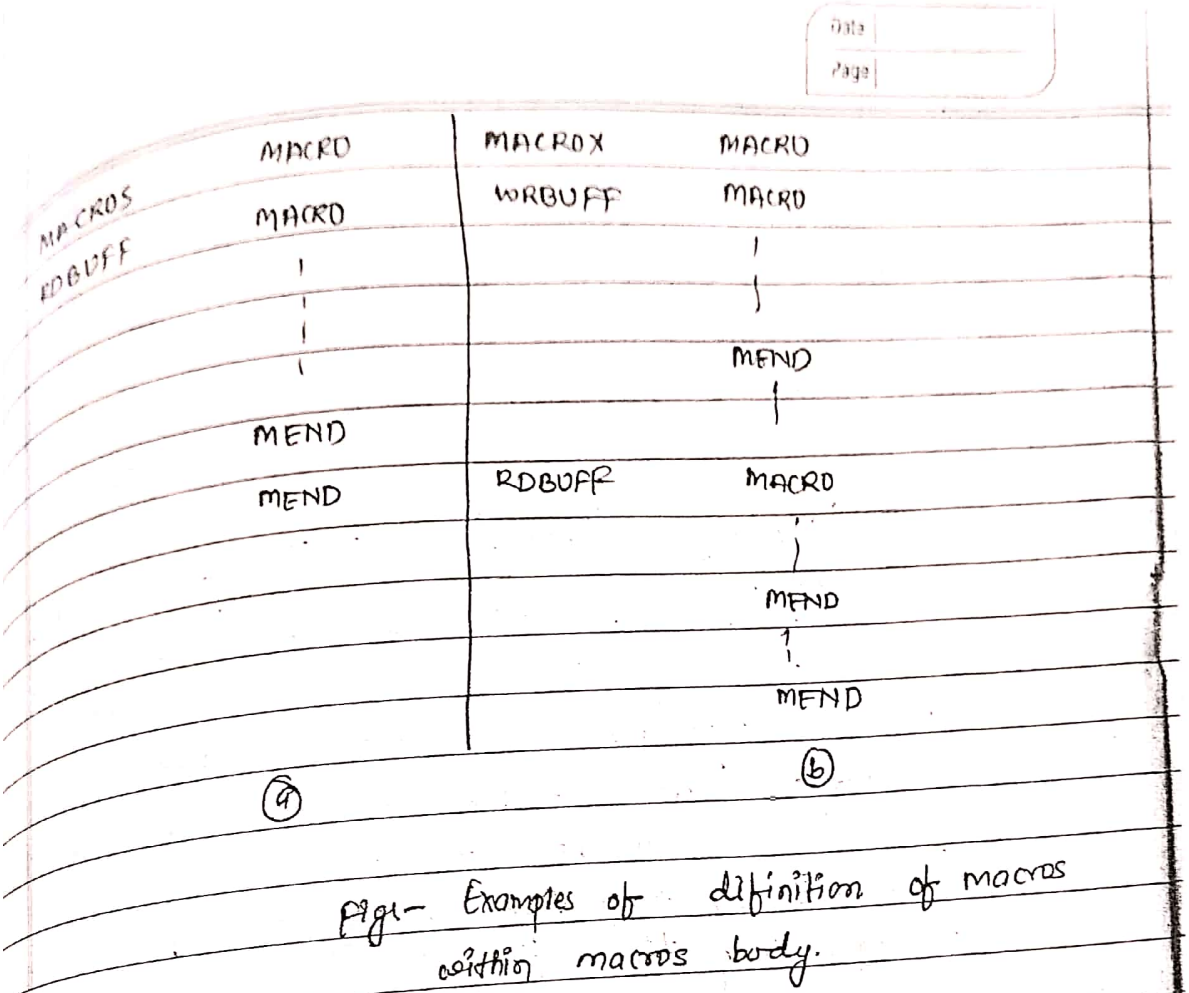


Fig- Examples of difinition of macros within macros body.

Date
Page

(c)

- Defining MACROS and MACROX does not define RDBUFF and other macro instructions.
- these definitions are processed only when an invocations of MACROS and MACROX is expanded.
- a one pass macro processor that can alternate between macro definition and macro expanssion is able to handle macro like these
- There are 3 main data structures in macro processor

(a) Definition Table (DEFTAB)

- The macro definition themselves are stored in definition table (DETTAB) which contains macro prototype and statements that makeup macro body.
- Comment lines from macro definition are not entered into DEFTAB ∴ they will not be part of macro expansion.

(b) Name Table (NAMTAB)

- references to macro instructions parameters are corrected to a positional entered into NAMTAB, which serves index to DEFTAB.

- For each macro instruction defined, NAMTAB contains pointers to begining and end of definition in DEFTAB.

(c) Argument table (ARGTAB)

- is used during exponssion of macro invocation.
- when macro invocation statements are recognized, the arguments are stored in ARGTAB according to their position in argument list.

Date
Page

An the macro is exponded, arguments from ARGTAB are substitute for the corresponding parameters in macro body.

NAMTAB

RDBUFF

ARGTAB

1	F1
2	BUFFER
3	LENGTH

DEFTAB

RDBUFF	&INDEV, &BUFADR, &RECLTH
CLEAR	X
CLEAR	A
CLEAR	S
+LDT	#4096
TD	=X'&&1'
JEQ	*-3
RD	=X'&&1'
COMPR	A,S
JEQ	*+11
STCH	&&2,X
TIXR	T
JLT	*-19
STX	&3
MEND	

Date
Page

→ when the ?n notation is ~~reg~~ recognised in a line form from DEFTAB, a simple indexing operation supplies the property argument from ARGTAB.

Algorithm

1) Procedure DEFINE
   - Called when the begining of a macro definition is recognized.
   - make appropriate enters in DEFTAB and NAMTAB.

2) Procedure ~~Expand~~ EXPAND
   - Called to set up the argument values in ARGTAB and expand a macro invocation statement

3) Procedure GETLINE
   - get the nextline to be processed.

Date
Page

Machine Independent Macroprocessor Features.

Concatenation of Macro parameters.

→ Most macro processors allow parameters to be concatenated with other character string.

→ Say, a program contains one series of variables named by the symbols XA1, XA2, XA3, ---. another series named by XB1, XB2, XB3, ... etc.

→ The body of macro definition might contain statements like

```
SUM     MACRO   &ID
        LDA     X&ID1
        ADD     X&ID2
        ADD     X&ID3
        STA     X&ID4
        |
        |
        MEND
```

→ Pre Concatenation
LDA X&ID1

→ Post Concatenation
LDA X&ID → 1

→ here, the begining of macro parameter is identified by starting symbol & however the end of parameter is not marked.

→ The problem is that the end of the parameter is not marked so, X&ID1 may mean 'X'+&ID+1 or 'X'+&ID1

→ To avoid this ambiguity, a special concentration operator '→' is used.

→ Now, new form becomes X&ID → 1
here, '→' will not appear in macro expansion.

Date
Page

eg:-

```
SUM     MACRO     &ID
        LDA       X&ID → 1
        ADD       X&ID → 2
        ADD       X&ID → 3
        STA       X&ID → 5
        MEND
```

Now,

SUM A	SUM BETA
↓	↓
LDA XA1	LDA XBETA1
ADD XA2	ADD XBETA2
ADD XA3	ADD XBETA3
STA XA5	STA XBETA5

Date
Page

* Generation of unique labels.

- If a macro is invocated and expanded multiple times, labels in the macro body may cause duplicate labels.
- to generate unique labels for each macro invocation, we must begin a label ~~with $ while writing macro invocation, we must begin a label with $ while~~ with $ while writing macro definition.
- during macro expansion, the $ will be replaced with $XX where XX is a two character alphanumeric counter of the number of macro instruction expanded.
  - XX will start from AA, AB, AC, ----
- Consider definition of WRBUFF

```
5       COPY    START   0
        :
135     TD      =X'&OUTDEV'
        :
140     JEQ     *-3
        |
155     JLT     *-14
        :
255     END     FIRST.
```

- here, if a label was placed on TD instruction on line 135, this label would be defined twice, once for each invocation of WRBUFF.
- This duplicate definition would prevent correct assembly of the resulting expanded program.
- the relative addressing on line 140 & 155, may be acceptable for short jumps.

Date
Page

→ for longer jumps spanning several instructions, such notation is very inconvenient, error prone and difficult to read.

→ these problems are avoided using special types of labels within macro instructions.

eg:-

RDBUFF Definition.

```
RDBUFF   MACRO    &INDEV, &BUFADR, &RECLTH
         CLEAR    X
         CLEAR    A
         CLEAR    S
         +LDT     #4096
$LOOP    TD       =X'&INDEV'
         JEQ      $LOOP
         RD       =X'&INDEV'
         COMPR    A,S
         JEQ      $Exit
         STCH     &BUFADR,X
         TIXR     T
         JLT      $LOOP
$Exit    STX      &RECLTH
         MEND
```

Date
Page

⇒ Expansion.

Macro Expansion.

```
RDBUFF      F1, BUFFER, LENGTH
              ⇓
            CLEAR   X
            CLEAR   A
            CLEAR   S
            +LDT    #4096
$AALOOP     TD      =X'F1'
            JEQ     $AALOOP
            RD      =X'F1'
            COMPR   A,S
            JEQ     $AALOOP Exit
            STCH    BUFFER,X
            TIXR    T
            JLT     $AALOOP
$AAExit     STX     LENGTH
```

Date
Page

Conditional Macro Expansion.

- So far, when a macro instruction is invoked, the same sequence of statements are used to expand macro.
- Here, depending on the arguments supplied in the macro invocation, the sequence of statements generated for macro expansion can be modified.
- ~~For this~~ this adds greatly to power and flexibility of macro language.

Macro time variable.

- is a variable that begins with '&' and that is not a macro instruction parameter.
- Can be initialized to a value of 0.
- Can be set by macro processor directive SET.
- Can be used to
  (i) Store working values during expansion.
  (ii) Store the evaluation result of Boolean Expansion.
  (iii) Control macro time conditional structures.

Macro time Conditional Structure.

i) IF - ELSE - END IF
ii) WHILE - ENDW

Implementation of Conditional Macro Expansion.

(i) IF - ELSE - ENDIF

- Firstly a symbol table is maintained by macro processor that contains the values of all macro time variables used
- entries in this table are made or modified when SET statements are processed.
- this table is used to look up the current value of

Date
Page

- macro time variable whenever it is required.
- the testing of condition and looping are done while macro is being expanded.
- when an IF statement is encountered during the expansion of a macro, the specified boolean expression is evaluated.
- if value is TRUE
  - the macro processor continues to process lines from DEFTAB until it encounters the next ELSE or ENDIF statement.
  - if ELSE is encountered, then skips to ENDIF.
- if value is FALSE
  - the macroprocessor skips ahead in DEFTAB until it finds the next ELSE or ENDIF statement.

(ii) WHILE - ENDW

- When a WHILE statement is encountered during the expansion of a macro, the specified Boolean expression is evaluated.
- if value is true
  - the macroprocessor continues to process lines from DEFTAB until it encounters next ENDW statement.
  - when ENDW is encountered, the macro processor returns to the preceding WHILE, re evaluates boolean expression and taken action again.
- if value is false.
  - the macro processor skips a head in DEFTAB until it finds the next ENDW statement and then resumes normal macro expression.

Date
Page

Keyword Macro Parameters.

1. Positional Parameter.

→ Parameters and arguments are associated according to their positions in the macro prototype and invocation.

→ programmer must specify the arguments in proper order

→ If an argument is to be omitted, a null argument should be used to maintain proper order in macro invocation statement.

→ For eg:- suppose a macro instruction GENER has 10 possible parameters, but in a particular invocation of the macro, only the 3rd and 9th parameters are to be specified.

Then statement is

GENER ,,DIRECT,,,,,,3,

→ It is not suitable if a macro has a large no. of parameters and only a few of these are given values in a typical invocation.

2. Keyword Parameter.

→ Each argument value is written with a keyword that names the corresponding parameter.

→ arguments may appear in any order.

→ null arguments no longer need to be used.

→ if the 3rd parameter is named &TYPE and 9th parameter is named &CHANNEL, the macro invocation would be

GENER TYPE = DIRECT, CHANNEL=3

→ it is easier to read and much less error prone than positional method.

Date
Page

Macroprocessor Design option

→ ~~Recsive~~ Recursive macro invocation and expansion can't be handled by previous macro processor design.

→ Reasons:-

1) the procedure EXPAND would be called recursively, thus invocation arguments in the ARGTAB will be overwritten,
2) the boolean variable EXPANDING would be set to FALSE when the inner macro expansion is finished, that is, the macro processor would forget that it had been in the middle of expanding 'outer' macro.
3) A similar problem would occur with PROCESSLINE since this procedure too would be called recursively.

→ Solutions:-

1) Write the macro processor in a programming language that allows recursive calls, thus local variables will be retained,
2) Use a stack to take care of pushing and popping local variables and return address.

(a) Two pass macro processor.

→ Pass 1
- process macro definition.

→ Pass 2
- expand all macro invocation statements.

→ Problems
- this kind of macro processor can not allow resursive macro definition; i.e. the body of a macro contains definitions of other macros.

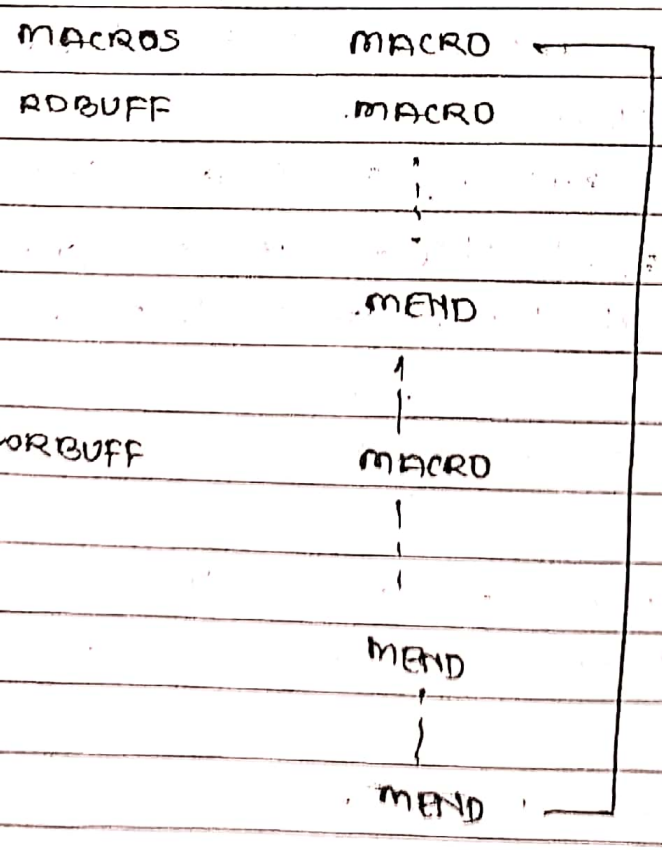
Example of recursive macro definition.

→ Name MACROS (for SIC)
- Contains the definition of RDBUFF and WRBUFF written in SIC instruction.

→ MACROX (for SIC/XE)
- Contains definitions of RDBUFF and WRBUFF written in SIC/XE instructions.

⇒ A program that is to be run on SIC system could invoke MACROS whereas a program that is to be run on SIC/XE system could invoke MACROX.

⇒ Defining MACROS or MACROX doesnot define RDBUFF and WRBUFF. These definitions are processed only when an invocation of MACROS or MACROX is expanded.



Date
Page

(5) one pass assembler.

→ a one pass macroprocessor alternates between macro definition and macro expansion in a recursive way, to handle recursive macro definition.

→ Because of ~~one~~ one pass structure, the definition of a macro must appear in source program before any statements that invoke that macro.

Handling recursive macro definition:

→ in DEFINE procedure

- when a macro definition is being entered into DEFTAB, the normal approach is to continue until an MEND directive is reached.
- this wouldnot work for recursive macros definition because the first MEND encountered in the inner macro will terminate whole macro definition process.
- to solve this problem, a counter LEVEL is used to keep track of level of macro definition.
- increase level by 1 each time a MACRO directive is read.
- decrease level by 1 each time a MEND directive is read.
- A MEND can terminate the whole macro definition process only when LEVEL reaches 0.

(6) General purpose macro processor

Goal:-

→ macro processors that do not depend on any particular programming language, but can be used

with a variety of different languages.

advantages:-

→ programmers do not need to learn many macro languages.

→ although its dependent development costs are somewhat greater than those for a language specific macro processor, this expense does not need to repeated for each language, thus save overall cost.

disadvantages.

→ large number of details must be dealt with real programming language.

→ simulation in which normal macro parameter substitution should not occur.

→ Syntax.

(d) Macro processing within language translations.

→ Macro processor can be

(a) Preprocessors

→ process macro definition

→ expand macro invocation

→ produce an expanded version of source program which is then used as input to an assembler or compiler.

(b) line by line macro processor.

→ used as a sort of input routine for the assembler or compiler.

→ read source program.

→ process macro definition and expand macro invocations
→ pass output lines to the assembler or compiler.
→ it avoids making an extra pass over the source program.
→ Data structure required by the macro processor and the language translator can be combined.
→ Utility subroutines can be used by both processor and the language translator.
  - Scanning input lines
  - Searching table.
  - data format conversion

② Integrated macroprocessor:
→ an integrated macro processor can potentially make use of any ~~instruction~~ information about the source program that is extracted by language translator.
→ it can support macro instructions that depend upon the context in which they occur.

Drawback of line by line or integrated macro processor
→ they must be specially designed and written to work with a particular implementation of assemble or compiler.
→ the cost of macro processor development is added to the costs of language translator, which results expensive development.
→ the assembler or compiler will be large and complex.

M.T.V. = macro time variable.

Date
Page

* Conditional macro expansion (example)

```
RDBUFF    MACRO    &INDEV, &BUFFADR, &RECLTH, &EOR, &MAXLEN
          IF       (&EOR NE '')
&EORCK    SET      1
          ENDIF
          CLEAR    X
          CLEAR    A
          IF       (&EORCK EQ 1)
          LDCH     =X'&EOR'
          RMO      A,S
          ENDIF
          IF       (&MAXLEN .EQ '')
          +LDT     #4096
          ELSE
          +LDT     #&MAXLEN
          ENDIF
          .
          .
          MEND
```

* condition RDBUFF F2, BUFFER, LENGTH, 3, ,
                     1   2   3   4 5

1. ~~Step~~ Checks If (&EOR NE '') → true
2. checks If (&EORCK EQ 1) → true
3. checks If (&MAXLEN EQ '') → true

```
4.        CLEAR   X
          CLEAR   A
          LDCH    =X'3'
          RMO     A,S
          +LDT    #4096
```

//

Date
Page

Ans Assignment:

1. Consider the macro definition given below and show macro expansion for the macro call statement "print G4, F1". Show all data structures used by macro processor clearly.

```
                        $    ?2
print   MACRO    &ch, &od
$Repeat  TD      &od
         JEQ     $Repeat
         LDCH    #&ch
         WD      &od
         MEND
```

Fig: Data structure

DEFTAB

Print		&ch, &od
$Repeat	TD	?2
	JEQ	$Repeat
	LDCH	#?1
	WD	?2
	MEND	

NAMTAB

Print

ARGTAB

1	G4
2	F1

Pg 2

Expansion

```
• print     64, F1
$AAARepeat  TD   F1
            JEQ  $AAARepeat
            LDCH #64
            WD   F1
```


chapter 5:- Object oriented System Design.

→ Focus on the objects handled by the system, rather than algorithms.

→ Programs are designed and implemented as collection of objects not as collection of procedures.

principles of object programming.

ⓐ **Object**

- is a basic unit of oop.
- is a component of a program that knows how to perform certain actions and how to interact with other elements of the program.
- Contains some data and defines a set of operations on that data that can be invoked by other parts of program.

eg:- Consider symbol-table as an object used by assembler. there, set of operation or methods are like
invert_symbol and lookup_symbol
its data would be contents of hash table used to store symbols and their addresses.

ⓑ **Class**

- is a blueprint or template or set of instructions to build a specific type of object.
- defines the instance variables and methods of an object.
- an instance is a specific object from specific class.
- many objects can be created from same class.

eg:- for an assembler to translate programs for different versions of machine, class could be opcode_table.
from this class, object could be created to define instruction set for machine.

ⓒ **Encapsulation**

- means that the internal representation of an object is generally hidden from view outside of objects definition.
- is the hiding of data implementation by restricting access to accessors and mutators.

③ Abstraction

- is a model, a view or some other focused representation for an actual item.
- is the implementation of an object that contains same essential properties and actions we can find in the original object we are representing.

④ Inheritance

- is a way to reuse code existing objects or to establish a subtype from an existing object.
- the relationship of classes through inheritance gives rise to a hierarchy.

Subclass :- is a modular, derivative class that inherits one or more properties from another class.

Superclass :- establishes a common interface and foundation functionality, which specialized subclass can inherit modify and supplement.

⑤ Polymorphism

- means one name, many forms
- manifests itself having multiple methods all with some name, but slightly different functionality.

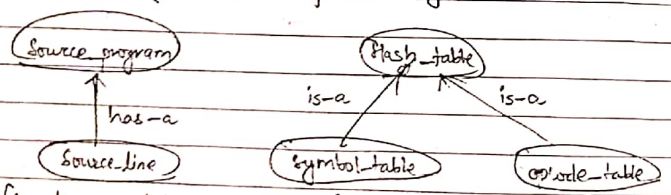


fig:- has-a relationship

fig 2:- is-a relationship or inheritance

→ Hash_table is base class
→ other two are subclasses.
→ if insert_item and search_item are methods of base class then other subclasses automatically contains definition of methods.

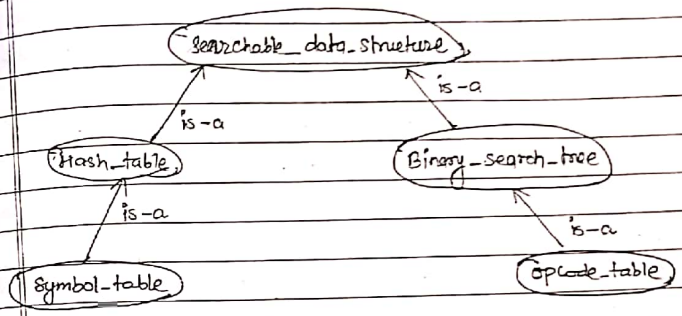


Fig 3 polymorphism.

→ here,
Superclass Searchable_data_structure defines two methods insert_item and search_for_item
→ Hash_table and Binary_search_tree are subclasses, so inherits above methods
→ implementation of the methods are different ~~developers processes is micro its macro~~
→ ~~Booch's macro process represents overall activities of development on a long~~
in these subclasses, but ~~name~~ names of methods and way of invokation are same.
→ if Search_for_item method is invoked as instance of Symbol_table, it will result in retrival from hash table
→ if same method is invoked on an instance of opcodetable it will result in binary_search_tree.
→ This shows polymorphism.

Object oriented design of an assembler

→ According to Booch, two different development processes (i) micro (ii) macro

→ Booch's macro process represents overall activities of development on a long range scale.

i) Establish the requirement for the sw (conceptualization)
ii) Develop an overall model of system behaviour (analysis)
iii) create an architecture for the implementation (design)
iv) Develop the implementation through successive refinements. (evolution)
v) Manage the continued evolution of a delivered system (maintenance)

→ this macro process repeats itself after each release of s

→ Similar to waterfall model.

→ Booch's micro process ~~elements~~ represents daily activities of system developer.

i) Identify the class and objects of system.
ii) Establish the behaviour and other attributes of the classes and objects.
iii) analyze the relationship among the classes and obje
iv) specify the implementation of classes and objects.

→ These activities may be repeated as needed wit increasing level of details.

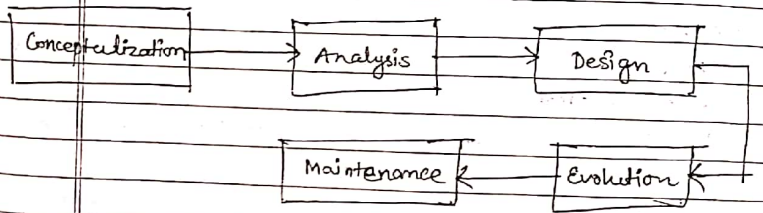


Fig:- Booch's macro process.

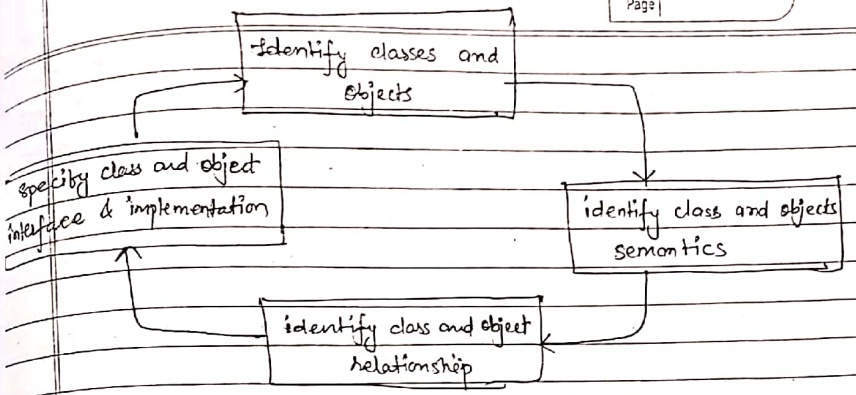


Fig:- Booch's micro process.

* Objects identified during design of assembler.

1) Source_program
   i) Contents
      - program
      - current location counter values, errors.
      - one object of class source_line, for each line of program.
   ii) methods
      ⓐ Assemble
      - translate source program, produce an object program, and an assembly listing

2) Source_line
   i) Contents.
      - line of source program
      - location counter value, error
   ii) methods
      ⓐ create
      - Create and initialize new instance of source_line
      ⓑ Assign_location
      - assign location counter value to line
      - return updated location counter value.
      - enter label on line (if any) in symbol table.
      ⓒ Translate
      - translate the instruction or

data definition on the line into machin language.
- make entries in object program & assembly listing.

ⓓ Record_errors
- record error detected.

3. Symbol_table

(i) Contents
- Labels defined in src program with its location counter value.

(ii) Methods

ⓐ Error
- enter a label and location counter value into table.
- return errors if label is already defined.

ⓑ Search
- Search table for specified label.
- return location counter value of label or error if label is not defined.

4. Opcode_table

(i) Contents
- mnemonic instruction
- includes machine instruction format and code.

(ii) Methods

ⓐ Search:-
- Search table for specified mnemonic instruction
- return information about instruction format and operand required.

- return error if mnemonic instruction not defined.

5. Object_program

i) Content
- object program after assembly.
- includes machine language translation of instruction and data de from object program.
- includes program length.

ii) Methods

ⓐ Enter_text
- enter machine language translation of an instruction or data def^n into object program.

ⓑ Complete.
- enter program length and complete generation of external obj program file.

6. Assembly_listing

i) Contents
- listing of lines of source program and corresponding machin language translation.
- includes errors for each line & summary of errors in progr

(ii) Methods

ⓐ Enter_line
- Enter source_line, the corresponding machine language translation and description of errors detected for the line into assembly listing.

ⓑ Complete.
- Enter summary of errors detected and complete the generati

of external assembly listing file

* Object program

- indicates the methods that are invoked by each object.
- eg:- source_program object invokes method create, Assign_loca and translate on source_line object.

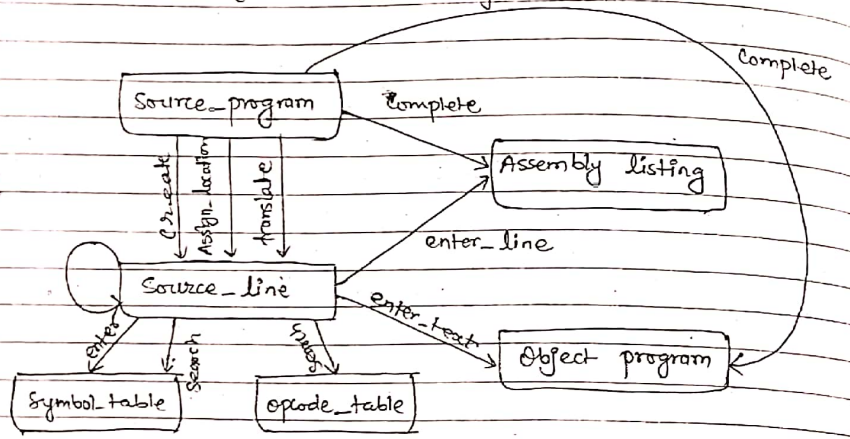


Fig:- Object diagram of assembler.

- Object diagram ~~many~~ or may also indicate the class of each object.

* Interaction Diagram.

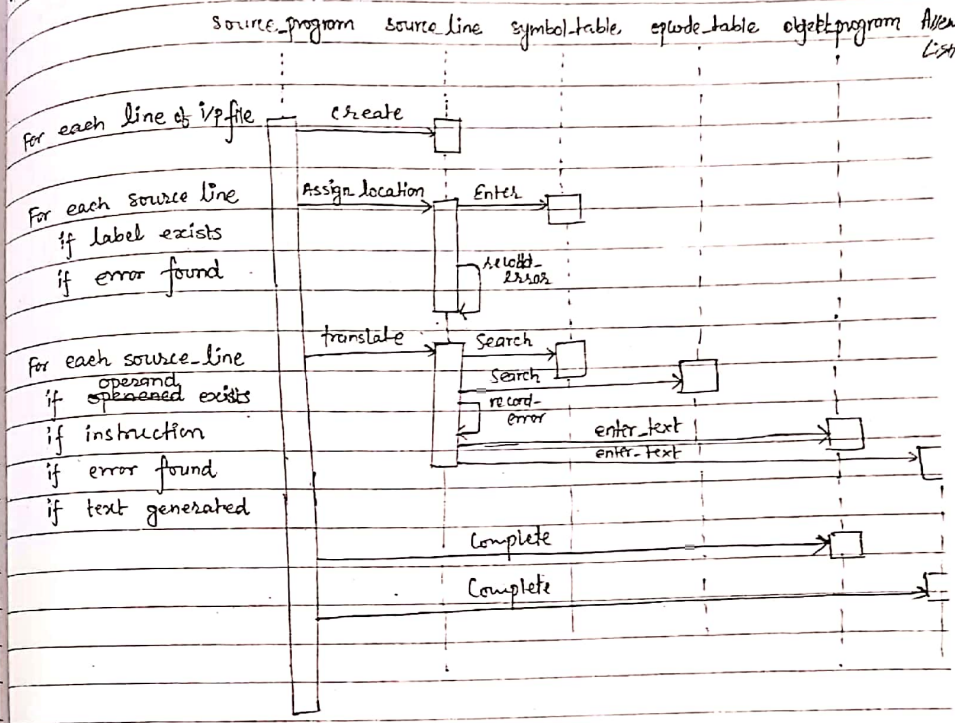


Fig:- Interaction Diagram.

- interaction diagram makes easy to visualize the sequence of object invocation and flow of control between objects.
- each object is represented by dashed vertical line.
- invocation of method is shown by horizontal line between objects.
- the sequence is indicated by their vertical position in diagram.
- a script is often written at L.H.S of diagram to describe

condition and iteration.

→ a narrow vertical box can be used to indicate the time flow of control is focused in each object.