1. Fundamental of language processing.

Language Processing = Analysis of SP + Synthesis of TP.

 Analysis phase of Language processing

 Lexical rules which govern the formation of valid lexical units in the source language.

 Syntax rules which govern the formation of the valid
statements in the source language.

 Semantic rules which associate meaning with the valid statements of the language.

The synthesis phase is concerned with the construction of target language statement which have same meaning as a source statement.

◦ Creation of data structures in the target program(memory
allocation)

◦ Generation of target code.(Code generation)

Language Processor

2.Explain back end operation of toy compiler

# Back End

- Back End
  1. Memory Allocation
  2. Code Generation
- 1. Memory Allocation:
  - A memory allocation requirement of an identifier is computed from its
    - Size,
    - Length,
    - Type,
    - Dimension
  - And then memory is allocated to it.
  - The address of memory area is then entered in symbol table. See next figure.
  - i* & temp are not shown because memory allocation is not required now for this id.
  - i* and temp should be allocated memory but this decision are taken in preparatory steps of code generation.

| Sr. No. | Symbol | Type | Length | Address |
|---|---|---|---|---|
| 1 | i | Integer | - | 2000 |
| 2 | a | Real | - | 2016 |
| 3 | b | Real | - | 2020 |

- 2. Code Generation:
- Uses knowledge of target architecture.
  - 1. Knowledge of instruction
  - 2. Addressing mode in target computer.
- Important issues effecting code generation:
  - Determine the place where the intermediate results should be kept. i.e in memory or register?
  - Determine which instruction should be used for type conversion operation.
  - Determine which addressing mode should be used for accessing variable.
- Eg: for sequence of actions for assignment statement a:=b+c;
  1. Convert I to real, giving i*;
  2. Add i* to b, giving temp;
  3. Store temp in a.

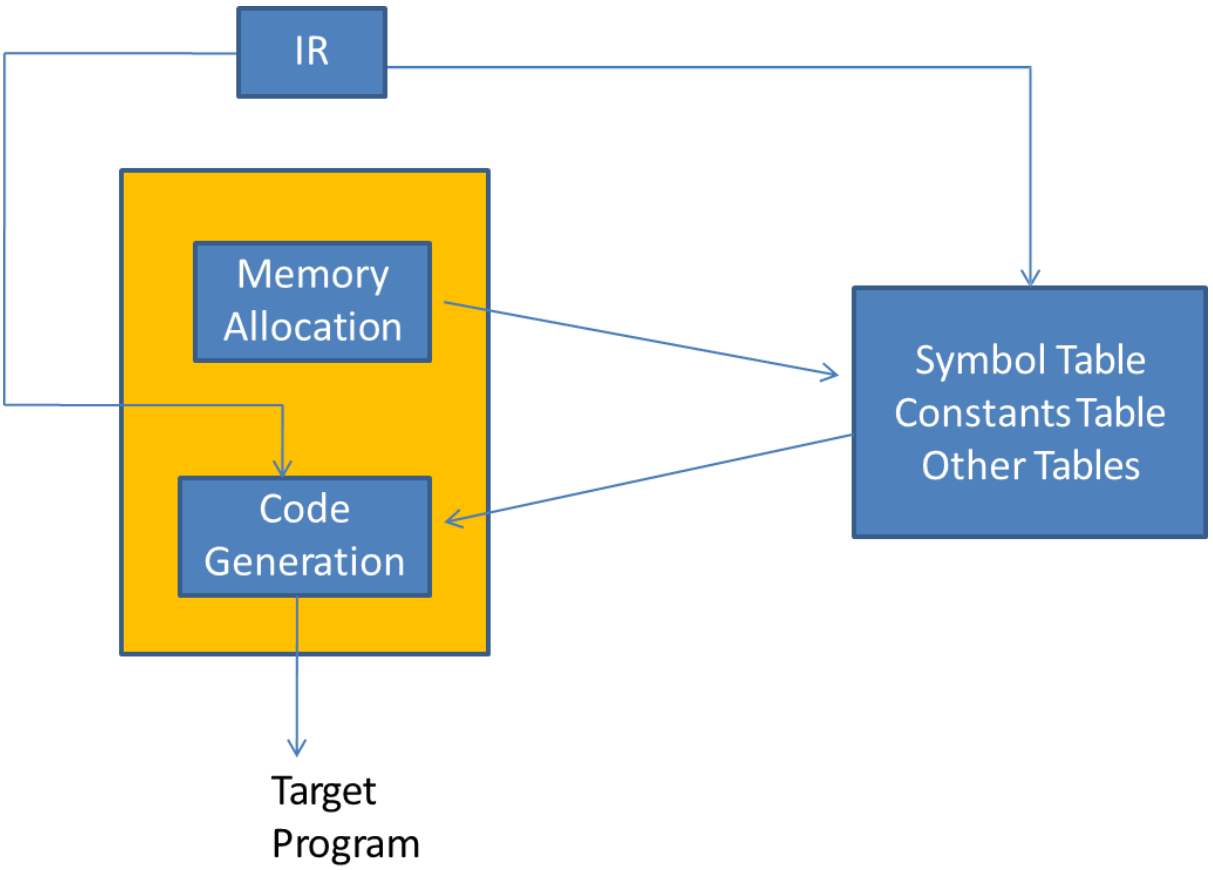
  The synthesis phase:

  a. decide to hold the values of i* & temp in machine register.

  b. generate following assembly code.

```
CONV_R      AREG, I
ADD_R       AREG,B
MOVE_M      AREG,A
```

- See next figure of Back End.



3. Expalin dd.differntiate char and block dd.

• A DD is the glue between an OS and its I/O devices
• DD act as a translators, converting the generic requests received from OS into commands that specific peripheral controllers can understand
• Ref. fig.1-1 pg. 2 where
– application s/w makes system calls to the OS requesting services
– the OS analyzes these request and when necessary
issues request to appropriate DD
– the DD analyzes the request from OS and issue command to the hardware interface to perform the operation needed to request)

Baki…

4.expalin function of
init(),open(),read(),poll(),strategy().

# Purpose of init() entry-point

- To check that the device is actually installed on the machine
- To print a message indicating the presence (or absence) of the device and driver
- To initialize the device (if necessary) prior to first open
- To initialize the driver and allocate local memory (if necessary) prior to the first open

The read entry point of a character driver is called when the user's process has requested data from the device using the read SYSTEM CALL

while(the user's read request is not completely satisfied)

{

copy 1 byte

from: foxmessage[indexed by u.u_offset]

to: the user's buffer addressed by u.u_base

if (an error occurs)

set u.u_error and return

update u.u_offset, u.u_base, and u.u_count

}

- Responsible for handling data (in or out)
- Replaces both read() and write() entry points

•Block driver uses a pointer to buffer header in Kernel space rather than u. in User process Area.

•Pointer is passed to the strategy routine

•No concern with process scheduling

•Strategy routine cannot complete I/O, it returns.

•When I/O is completed, block driver notifies Kernel with *iodone(bp),* passing pointer to the processed buffer.

# strategy() entry-point

strategy entry point:

if(read request)

copy data from testpattern to buffer

set residual count to 0 (indicating a successful transfer)

report that I/O request has been completed

5. Explain various data sttructure used in assembler pase 1 and 2

# Data Structures in Pass I

- OPTAB – a table of mnemonic op codes
  - Contains mnemonic op code, class and mnemonic info
  - Class field indicates whether the op code corresponds to
    - an imperative statement (IS),
    - a declaration statement (DL) or
    - an assembler Directive (AD)
  - For IS, mnemonic info field contains the pair ( machine opcode, instruction length)
  - Else, it contains the id of the routine to handle the declaration or a directive statement
  - The routine processes the operand field of the statement to determine the amount of memory required and updates LC and the SYMTAB entry of the symbol defined

# Data Structures in Pass I

- SYMTAB - Symbol Table
  - Contains address and length
- LOCCTR - Location Counter
- LITTAB – a table of literals used in the program
  - Contains literal and address
  - Literals are allocated addresses starting with the current value in LC and LC is incremented, appropriately

# OPTAB (operation code table)

- Content
  - Menmonic opcode, class and mnemonic info
- Characteristic
  - static table
- Implementation
  - array or hash table, easy for search

# SYMTAB (symbol table)

- Content
  - label name, value, flag, (type, length) etc.
- Characteristic
  - dynamic table (insert, delete, search)
- Implementation
  - hash table, non-random keys, hashing function

li and example

# 6.All function of advance assembler directive

# Assembler Directives

- Assembler directives are pseudo instructions.
  - They provide instructions to the assemblers itself.
  - They are not translated into machine operation codes.like START , END.
- Advanced Assembler directives are
  - ORIGIN <address specification>

  Where <address specification> is <operand specification> or <constatnt> , this directive instruct the assembler to put <address specification> to location counter. Useful when lack of single contiguous area of memory.

```
 1          START   200
 2          MOVER   AREG, ='5'      200)  +04 1 211
 3          MOVEM   AREG, A         201)  +05 1 217
 4   LOOP   MOVER   AREG, A         202)  +04 1 217
 5          MOVER   CREG, B         203)  +05 3 218
 6          ADD     CREG, ='1'      204)  +01 3 212
 7          ...

12          BC      ANY, NEXT       210)  +07 6 214
13          LTORG
                    ='5'            211)  +00 0 005
                    ='1'            212)  +00 0 001
14          ...
15   NEXT   SUB     AREG, ='1'      214)  +02 1 219
16          BC      LT, BACK        215)  +07 1 202
17   LAST   STOP                    216)  +00 0 000
18          ORIGIN  LOOP+2
19          MULT    CREG, B         204)  +03 3 218
20          ORIGIN  LAST+1
21   A      DS      1               217)
22   BACK   EQU     LOOP
23   B      DS      1               218)
24          END
```

**Example 3.4 (The ORIGIN directive)** Statement number 18 of Figure 3.8(a), viz. `ORIGIN LOOP+2`, puts the address 204 in the location counter because the symbol `LOOP` is associated with the address 202. The next statement

```
MULT        CREG, B
```

is therefore given the address 204. The statement `ORIGIN LAST+1` puts the address 217 in the location counter. Note that an equivalent effect could have been achieved by using the statements `ORIGIN 204` and `ORIGIN 217` at these two places in the program; however, the absolute addresses used in these statements would have to be changed if the address specification in the `START` statement is changed.

# EQU

**EQU**

The EQU directive has the syntax

<*symbol*> EQU <*address specification*>

where <*address specification*> is either a <*constant*> or <*symbolic name*> ± <*displacement*>. The EQU statement simply associates the name <*symbol*> with the address specified by <*address specification*>. However, the address in the location counter is not affected.

**Example 3.5 (The EQU directive)** Before processing the $22^{nd}$ statement of Figure 3.8(a), the assembler had assembled the statement `A DS 1` and given the address 217 to `A`. Hence the location counter contains the address 218 at this time. On encountering the statement `BACK EQU LOOP`, the assembler associates the symbol `BACK` with the address of `LOOP`, i.e., with 202. Note that the address in the location counter is not affected by the EQU statement. That is how the symbol `B` defined in the next statement `B DS 1` is assigned the address 218. In the second pass, the $16^{th}$ statement, i.e., `BC LT, BACK`, is assembled as '+ 07 1 202'.

# LTORG

The assembler should put the values of literals in such a place that control does not reach any of them during execution of the generated program. The LTORG directive, which stands for 'origin for literals', allows a programmer to specify where literals should be placed. The assembler uses the following scheme for placement of literals: When the use of a literal is seen in a statement, the assembler enters it into a *literal pool* unless a matching literal already exists in the pool. At every LTORG statement, as also at the END statement, the assembler allocates memory to the literals of the literal pool and clears the literal pool. This way, a literal pool would contain all literals used in the program since the start of the program or since the previous LTORG statement. Thus, all references to literals are forward references by definition. If a program does not use an LTORG statement, the assembler would enter all literals used in the program into a single pool and allocate memory to them when it encounters the END statement.

7.compare and explain variant 1 and 2 with example

# IC

**Variant I**

Figure 3.12 shows an assembly program and its intermediate code using Variant I. The first operand in an assembly statement is represented by a single digit number which is either a code in the range 1...4 that represents a CPU register, where 1 represents AREG, 2 represents BREG, etc., or the condition code itself, which is in the range 1...6 and has the meanings described in Section 3.1. The second operand, which is a memory operand, is represented by a pair of the form

(*operand class, code*)

where *operand class* is one of C, S and L standing for constant, symbol and literal, respectively. For a constant, the *code* field contains the representation of the constant itself. For example, in Figure 3.12 the operand descriptor for the statement START 200 is (C, 200). For a symbol or literal, the *code* field contains the entry number of the operand in SYMTAB or LITTAB. Thus entries for a symbol XYZ and a literal ='25' would be of the form (S, 17) and (L, 35), respectively.

```
        START   200             (AD,01)   (C,200)
        READ    A               (IS,09)   (S,01)
LOOP    MOVER   AREG, A         (IS,04)   (1)(S,01)
          ⋮                       ⋮
        SUB     AREG, ='1'      (IS,02)   (1)(L,01)
        BC      GT, LOOP        (IS,07)   (4)(S,02)
        STOP                    (IS,00)
A       DS      1               (DL,02)   (C,1)
        LTORG                   (DL,05)
        ...                     ...
```

**Variant II**

Figure 3.13 shows an assembly program and its intermediate code using Variant II. This variant differs from Variant I in that the operand field of the intermediate code may be either in the processed form as in Variant I, or in the source form itself. For a declarative statement or an assembler directive, the operand field has to be processed in the first pass to support LC processing. Hence the operand field of its intermediate code would contain the processed form of the operand. For imperative statements, the operand field is processed to identify literal references and enter them in the LITTAB. Hence operands that are literals are represented as (L, *m*) in the intermediate code. There is no reason why symbolic references in operand fields of imperative statements should be processed during Pass I, so they are put in the source form itself in the intermediate code.

```
        START   200             (AD,01)   (C,200)
        READ    A               (IS,09)   A
LOOP    MOVER   AREG, A         (IS,04)   AREG, A
          ⋮                       ⋮
        SUB     AREG, ='1'      (IS,02)   AREG, (L,01)
        BC      GT, LOOP        (IS,07)   GT, LOOP
        STOP                    (IS,00)
A       DS      1               (DL,02)   (C,1)
        LTORG                   (DL,05)
        ...                     ...
```

**Figure 3.13** Intermediate code - Variant II

9.explain various ds used in macro def processing

Data structures
To obtain a detailed design of the data structure it is necessary to apply the practical criteria of processing efficiency and memory requirements.
The table APT,PDT and EVT contain pairs which are searched using the first component of the pairs as a key-the formal parameter name is used as the key to obtain its value from APT.
This search can be eliminated if the position of an entity within a table is known when its value is accessed.

The value of formal parameter ABC is needed while expanding a model statement using it
MOVERAREG, &ABC
Let the pair (ABC,5) occupy entry #5 in APT. the search in APT can be avoided if the model statement appears as
MOVERAREG, (P,5)
In the MDT, where (P,5) stand for the word „parameter #5".

The first component of the pairs stored in APT is no longer used during macro expansion e.g. the information (P,5) appearing in model statement is

sufficient to access the value of formal parameter ABC.
APT containing (<formal parameter name>,<value>) pairs is replaced by another table called APTAB which only contains <value>"s.
Ordinal number are assigned to all parameters of macro, a table named parameter name table (PNTAB) is used for this purpose.
Parameter name are entered in PNTAB in same order in which they appear in the prototype statement

The information (<formal parameter name>,<value>) in APT has been split into two tables
PNTAB-which contains formal parameter names
APTAB-which contains formal parameter values
PNTAB is used while processing a macro definition while APTAB is used during macro expansion

MNT has entries for all macros defined in a program, each entry contains three pointers MDTP,KPDTP and SSTP which are pointers to MDT,KPDTAB and SSNTAB for the macro respectively.
Macro preprocessor data structure can be summarized as follows:
PNTAB and KPDTAB are constructed by processing the prototype statement.

Entries are added to EVNTAB and SSNTAB as EV declarations and SS definitions/references are encountered.
MDT entries are constructed while processing model statements and preprocessor statements in macro body.
SSTAB entries, when the definition of sequencing symbol in encountered.
APTAB is constructed while processing a macro.
EVTAB is constructed at the start of expansion of macro

8.lexical and semantic expansion of macro with example

Two kind of expansion
Lexical expansion:
Lexical expansion implies replacement of character string by another character string during program generation.
Lexical expansion is typically employed to replace occurrences of formal parameter by corresponding actual parameters.
Semantic Expansion:

Semantic expansion implies generation of instructions tailored to the requirements of a specific usage
Example: generation of type specific instruction for manipulation of byte and word operands.

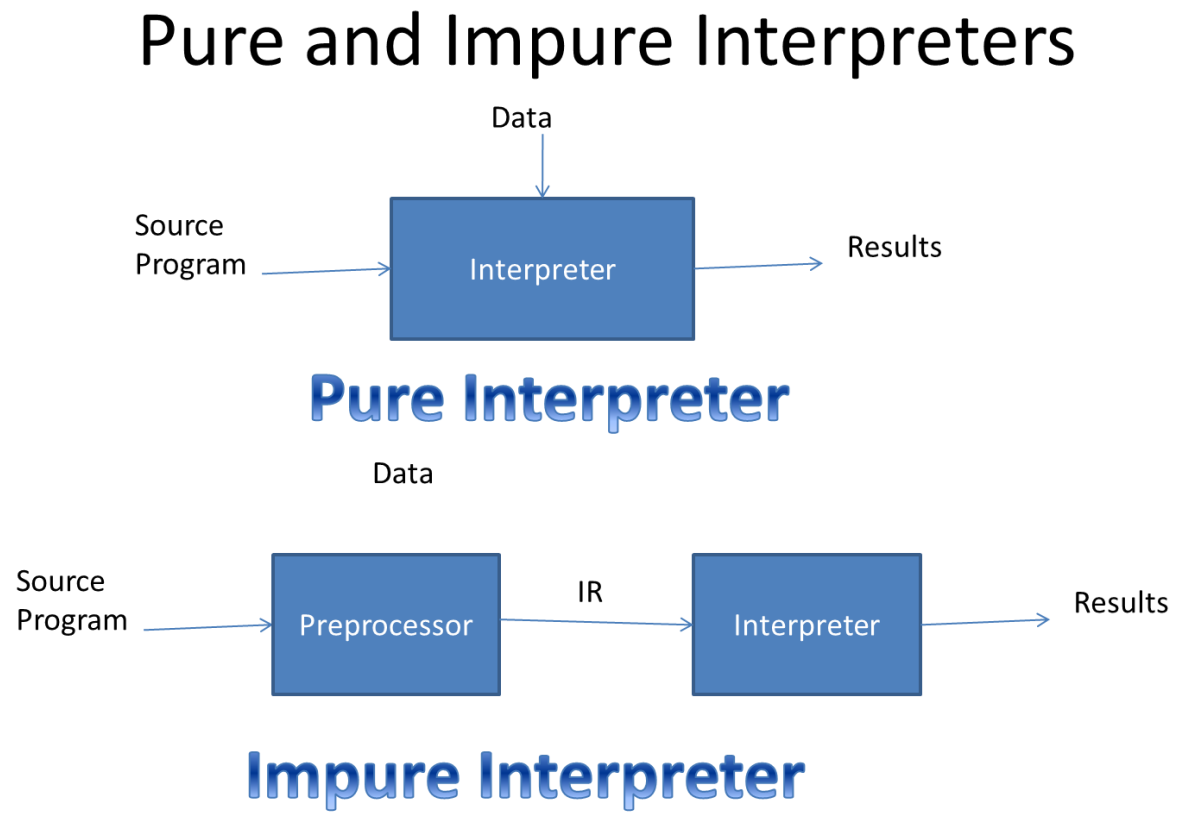
Using lexical expansion the macro call INCR A,B,AREG can lead to the generation of a MOVE-ADD-MOVE instruction sequence to increment A by the value B using AREG.

Example
Macro
INCR &MEM_VAL, &INCR_VAL, &REG
MOVER &REG, &MEM_VAL
ADD &REG,&INCR_VAL
MOVEM &REG, &MEM_VAL
MEND

Semantic Expansion:
Semantic expansion is the generation of instructions tailored to the requirements of a specific usage.
Example:
MACRO
CREATE_CONST&X, &Y
AIF(T‟&X EQ B).BYTE

&YDW25
AGO.OVER
.BYTEANOP
&YDB25
.OVERMEND

9.expain with the figure pure and impure interpreter.



# Pure & Impure Interpreters:

- **Pure Interpreters:**
  - Here, source program is retained in source form all through interpretation.
  - Dis-advantage: This arrangement incurs substantial analysis overheads while interpreting the statement.
  - Eliminates most of the analysis during interpretation except type analysis.
  - For type analysis pre processor is needed.
- – See fig. 6.34 (a). Pg. No. 217
  - See Ex. 6.42. IC intermediate code for postfix notation.

# Impure Interpreters:

See fig. 6.34 (b). Pg. no. 217.
See Ex. 6.43. IC intermediate code for postfix notation.

- Perform some preliminary processing of the source program to reduce the analysis overhead during interpretation.
- Pre-processor converts program to an IR which is used during interpretation.
- IC can be analysed more efficiently then source program.
- Thus, speed up interpretation.
- Dis-advantage: Use of IR implies that entire program has to
- be pre-processed after any modifications.
- Thus, incurs fixed overhead at the start of interpretation.
- Conclusion Statement: Thus, eliminates most of the analysis during interpretation.

10. Using nested macro call memory management using extented stack.

Two basic alternatives exist for processing nested macro calls.
In this code macro calls appearing in the source program have been expanded but statements resulting from the expansion may themselves contain macro calls.
This first level expanded code to expand these macro calls, until we obtain a code form which dose not contain any macro calls.
This scheme would require a number of passes of macro expansion, which makes it quite expensive

Two provisions are required to implement the expansion of nested macro calls:
Each macro under expansion must have its own set of data structures, (MEC,APTAB,EVTAB,APTAB_ptrand EVTAB_ptr).
An expansion nesting counter(Nest_cntr) is maintained to count the number of nested macro calls. Nest_cntris incremented when a macro call is recognized and decremented when MEND statement is encountered.

Creating many copies of the expansion time data structure, this arrangement provides access efficiency but it is expensive in terms of memory requirements.
Difficult in design decision-how many copies of the data structures should be created?
If too many copies are created then some may never be used.
If too few are created, some assembly programs may have to be rejected.

Macro calls are expanded in LIFO manner, the stack consists of expansion records, each expansion record accommodating one set of expansion time data structures.
Expansion record at the top of stack corresponds the macro call currently being expanded.
When a nested macro call is recognized, a new expansion record is pushed on the stack to hold the data structures for the call.
At MEND an expansion record is popped off the stack.

Record base (RB) is a pointer pointing to the start of this expansion record.
TOS point to the last occupied entry in stack.

When nested macro call is detected, another set of data structure is allocated on the stack.

Figures… book

11.overview of interpretation

# Overview of Interpretation

- Consists of the following three components:
  - Symbol Table
    - Information regarding entities in the source program
  - Data Store
    - Values of data-items
  - Data Manipulation Routines
    - Routines for executable statements
- Example, a:= b + c ;
  - add (ad_b, ad_c, ad_result)
  - assign(ad_a, ad_result)

12.explain code optimization technique.

# A. Optimizing Transformation

a) Compile Time Evaluation

b) Combination of Common Sub-expression

c) Dead Code Elimination

d) Frequency Reduction

e) Strength Reduction

f) Local & Global Optimization

## B. Local Optimization

a) Value Numbers

## C. Global Optimization

a) Program Representation

b) Control & Data Flow Analysis

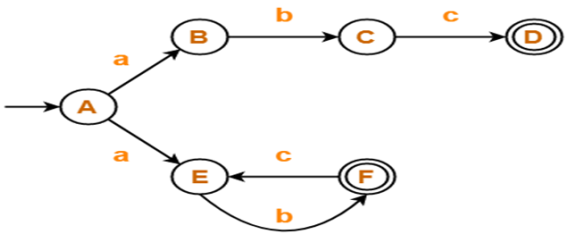
13. Explain nfa and dfa.

# Finite state automata

- Finite automata are graphs that decide whether a word is in the language (set of valid string generated by regular expression) or not.
- A finite state automaton (FSA) is triples (S, Σ, T) where
  - S : is a finite set of states, one of which is the initial state $S_{init,}$ and one or more of which are the final states.
  - Σ : is the alphabet of source symbols.
  - T : is finite set of state transitions defining transitions out of each si ϵ S in encountering the symbols of Σ.

- A finite automaton can be of two types:
- Deterministic and non-deterministic

- NFA means there can be more than one transition out of the state for the same input symbol.




Example of Non-Deterministic Finite Automata

(Without Epsilon)

- DFA has a unique transition for every state character transition.

# DFA

- Deterministic finite state automaton (DFA) is an FSA such that t1∈ T, t1=(si,symb,sj)
- At most one transition exists in state si for symbol symb.
- The DFA stops when all symbols in the source string are recognized, or error condition is encountered.

a single DFA as a recognizer for valid lexical strings in the language. Such a DFA would have a single initial state and one or more final states for each lexical unit. Example 6.13 illustrates use of a DFA for scanning.

**Example 6.13 (DFA as a scanner)** Figure 6.5 shows a DFA for recognizing identifiers, unsigned integers and unsigned real numbers with fractions. The DFA has 3 final states—*Id*, *Int* and *Real* that correspond to identifier, unsigned integer and unsigned real numbers, respectively. It reaches one of these final states on recognizing the corresponding category. Note that a string like '25.' is invalid because it leaves the DFA in state $s_2$ which is not a final state.

| State | Next Symbol | | |
|---|---|---|---|
| | *l* | *d* | . |
| *Start* | *Id* | *Int* | |
| *Id* | *Id* | *Id* | |
| *Int* | | *Int* | $s_2$ |
| $s_2$ | | *Real* | |
| *Real* | | *Real* | |

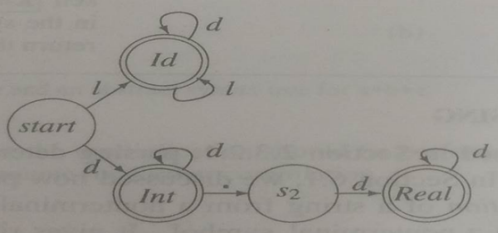



**Figure 6.5** A DFA for recognizing integers, real numbers and identifiers

A scanner needs to perform semantic actions like table building and construction

12.explain global optimization with ex.

- Require more analytic efforts to establish the feasibility of an optimization.
- Global common sub expression elimination is done here.
- Occurrence can be eliminated if it **satisfy two condition**:
  - 1. Basic Block bj is executed only after some block bk $\epsilon$ SB has been executed one or more times (Ensure x*y is evaluated before bj)
  - 2. No assignment to x or y have been executed after the last (or only) evaluation of x*y in block bj.
- By analyzing program using two techniques:
  - Control Flow Analysis
  - Data Flow Analysis
- let us see program representation which is done in the form of PFG.

## PFG: Program Flow Graph

- Def: A PFG for a program P is directed graph Gp = (N,E,n0)

- Where,
  - N : set of blocks
  - E : set of directed edges (bi, bj) indicating the possibility of control flow from the last statement of bi(source node) to first statement of bj(destination node).
  - n0 : start node of program.

- **Control & Data Flow Analysis:** Used to determine whether the condition governing and optimizing transformation are satisfied or not.

**1. Control Flow Analysis:** Collects information concerning its structure i.e nesting of loops.

- Few concepts:
  - **Predecessors & Successor**:- If (bi,bj) ϵ E, bi is a predecessor of bj & bj is a successor of bj.
  - **Paths**:- A path is a sequence of edges such that destination node of one edge is the source node of the following edge.
  - **Ancestors & Descendants** :- If path exist from bi to bj, bj is an ancestor of bj and bj is a descendant of bi.
  - **Dominators & Post Dominators**:- Block bi is a dominator of block bj if

every path from n0 to bj is passed through bi. And bi is the post dominator of bj if every path from bj to an exit node passes through bi.

- Advantages: Control flow concepts

answer the question of condition 1.

- Drawbacks: Incurs overhead at every

expression evaluation.

- Solution? Restrict the scope of

optimization.

## 2. Data Flow Analysis:

- Analyse the use of data in the program.

- Data flow information is computed for the purpose of optimization at entry & exit of each basic block.
- Determines whether optimization transformation can be applied or not.
- Concepts:

  – Available Expression

- Live Variables
- Reaching Definition

- Use:
  - Common sub expression elimination.
  - Dead code elimination

– Constant variable

propagation

14.linking for overlay with exampple.

Def: (Overlay)An overlay is a part of a program ( or software package) which has the same load origin as some other parts of the program.

•Overlays are used to reduce the main memory requirement of a program.

•Overlay structured programs:

–We refer to program containing overlays as an overlay structured program.

–Such program consist of:

1.A permanently resident portion, called the root.

2.A set of overlays.

•Execution of overlay structured program:

–1. root is loaded in the memory

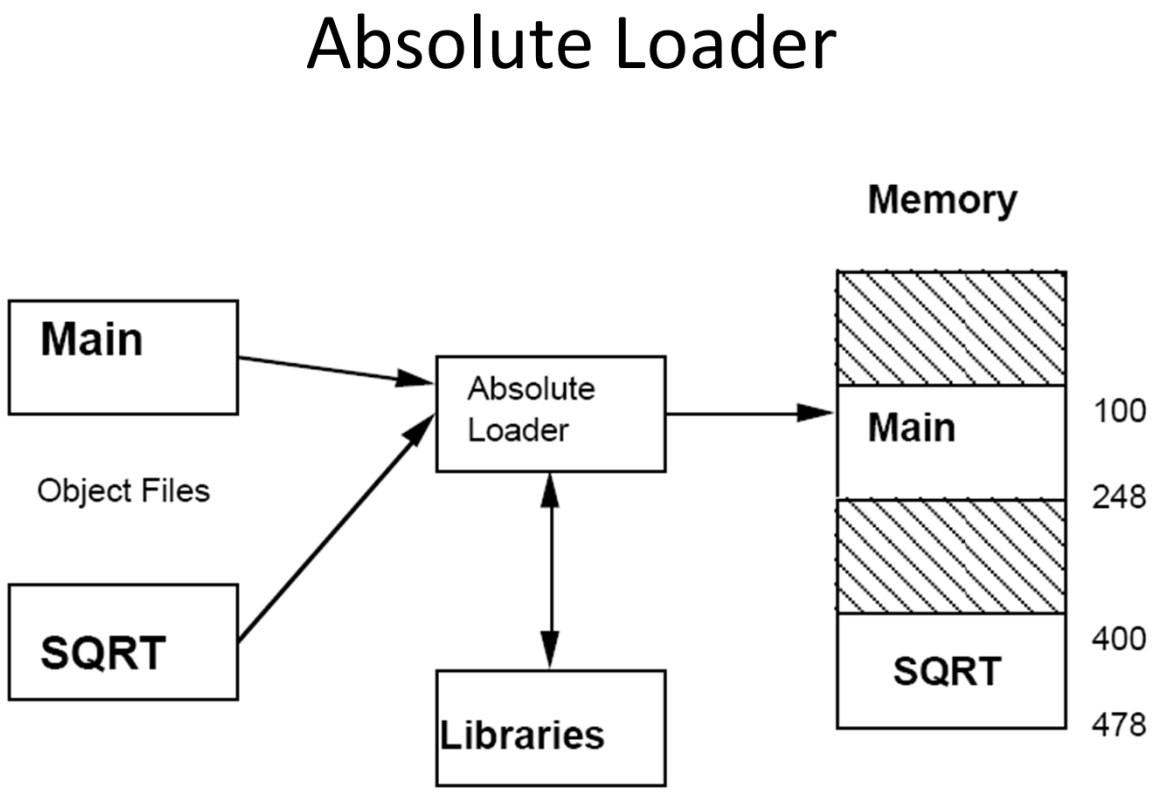
–2. other overlays are loaded as and when needed.

–3. loading of an overlay overwrites a previously loaded overlay with the same load origin.
•This reduces the memory requirement of the program.
•Hence, facilitate execution of those programs who's size exceeds memory size.
•How to design overlay structure?
–By identifying mutually exclusive module.
–Modules that do not call each other.
–These modules do not need to reside simultaneously in the memory.
–Hence keep those modules in different overlays with same load origin.

Execution of an overlay structured program:
–Overlay manager moduleis included in executable file which is responsible for overlay load as and when required.
–Interrupt producing instructionreplaces all cross overlay boundary calls.
•Changes in Linker Algorithm?
–Assignment of load address to segments after execution of root.
–Mutually exclusive modules are assigned same program_load_origin.
–Also algorithm has to identify inter-overlay call and determine destination overlay.

•See example 7.15 on pg. no. 245.

15. Absoluate loader with ex



# Design of an Absolute Loader

- Its operation is very simple
  - no linking or relocation
- Single pass operation
  - check **H record to verify that correct program** has been presented for loading
  - read each **T record, and move object code into** the indicated address in memory
  - at **E record, jump to the specified address to** begin execution of the loaded program.

# Disadvantages of Absolute Loaders

- Actual load address must be specified
- The programmer must be careful not to assign two subroutines to the same or overlapping locations
- Difficult to use subroutine libraries (scientific and mathematical) efficiently
  - important to be able to select and load exactly those routines that are needed

# Disadvantages of Absolute Loaders

- Allocation - by programmer
- Linking - by programmer
- Relocation - None required-loaded where assembler assigned
- Loading - by loader

14. General loader scheme with figure.

# Loader Schemes

- Compile and Go
  - The assembler run in one part of memory
  - place the assembled machine instructions and data, as they are assembled, directly into their assigned memory locations
  - When the assembly is completed, the assembler causes a transfer to the starting instruction of the program

1..1.1

# General Loader Scheme

- Linking
- Relocation
- Loading

# Two Pass Direct Linking Loader

- Pass 1
  - **Allocate** and assign each program location in core.
  - Create a symbol table filling in the values of the external symbols.
- Pass 2
  - **Load** the actual program text.
  - Perform the **relocation** modification of any address constants needing to be altered.
  - Resolve external references. (**linking**)

# Pass II

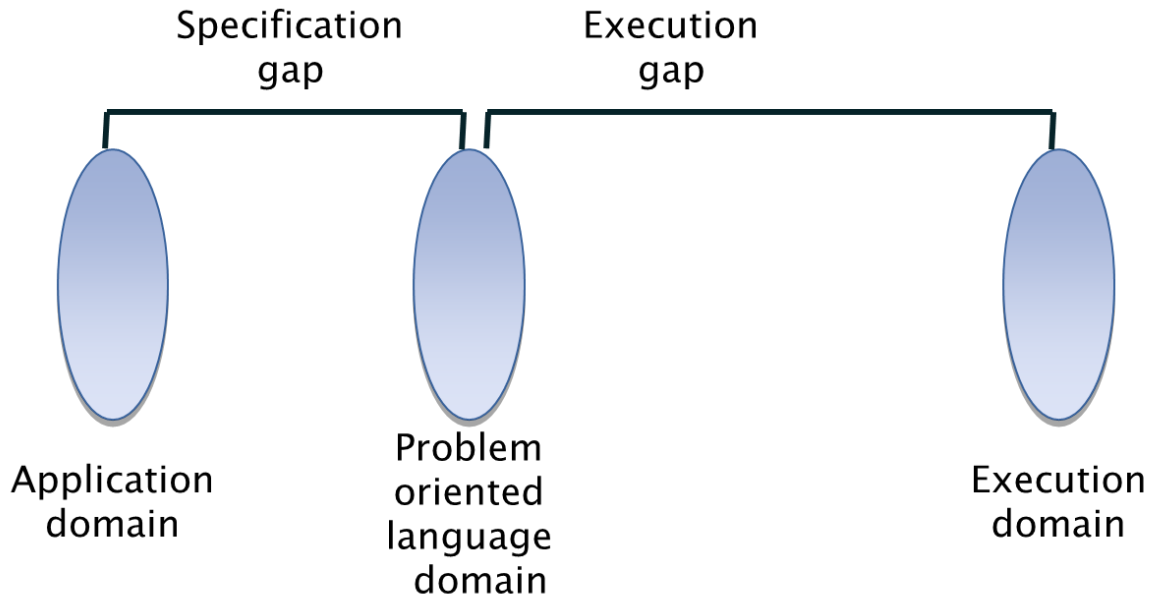
- Does actual loading, relocation, and linking

1.. Problem and procedure oriented lang.

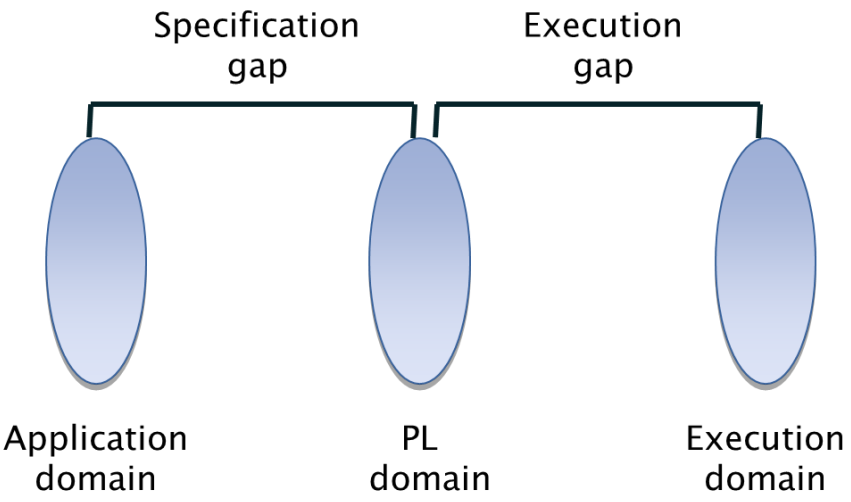
}Three consequences of the semantic gap are in fact the consequences of specification gap.

}A classical solution is to develop a PL such that the PL domain is very close or identical to the application domain.

}Such PLs can only used for specific applications, they are problem oriented languages.




- A procedure oriented language provides general purpose facilities required in most application domains.




17. Advance macro facility with ex.

Advance macro facilities are aimed at supporting semantic expansion.

Facilities for alteration of flow of control during expansion.
Expansion time variables
Attributes of parameters.

Alteration of flow of control during expansion
Alteration of flow of control during expansion:
Expansion time sequencing symbols (SS).
Expansion time statements AIF, AGO and ANOP.
Sequencing symbol has syntax

.<ordinary string>
A SS is defined by putting it in the label field of statement in the macro body.
It is used as operand in an AIF, AGO statement for expansion control transfer.

An AIF statement has syntax
AIF (<expression>) <sequencing symbol>
Where, <expression> is relational expression involving ordinary strings, formal parameters and their attributes, and expansion time variables.
If the relational expression evaluates to true, expansion time control is transferred to the statement containing <sequencing symbol> in its label field.

An AGO statement the syntax
AGO <sequencing symbol>
Unconditionally transfer expansion time control to the statement containing <sequencing symbol> in its label field.
An ANOP statement is written as
<sequencing symbol> ANOP
Simply has the effect of defining the sequencing symbol.

Expansion Time Variable (EV‟s)
Expansion Time Variable

Expansion time variable are variables which can only be used during the expansion of macro calls.
Local EV is created for use only during a particular macro call.
Global EV exists across all macro calls situated in program and can be used in any macro which has a declaration for it.
LCL <EV specification>[,<EV specification>…]
GBL <EV specification>[,<EV specification>…]

<EV specification>has syntax &<EV name>, where EV name is ordinary string.
Initialize EV by preprocessor statement SET.
<EV Specification> SET <SET-expression>

```
        MACRO
        CONSTANTS
        LCL     &A
&A      SET     1
        DB      &A
&A      SET     &A+1
        DB      &A
        MEND
```

A call on macro CONSTANTS is expanded as follows: The local EV A is created. The first SET statement assigns the value '1' to it. The first DB statement thus declares a byte constant '1'. The second SET statement assigns the value '2' to A and the second DB statement declares a constant '2'.

Attributes of formal parameters
Attributes of formal parameters:
<attribute name>" <formal parameter spec>
Represents information about the value of the formal parameter about corresponding actual parameter.

The type, length and size attributes have the name T,L and S.

```
          MACRO
          DCL_CONST   &A
          AIF         (L'&A EQ 1) .NEXT
          - -
.NEXT     - -
          - -
          MEND
```

Here expansion time control is transferred to the statement having .NEXT in its label field only if the actual parameter corresponding to the formal parameter A has the length of '1'.

types :
Conditional expansion->Expansion time loops
Other facility for expansion time loop
Semantic expansion

Conditional expansion:
Conditional expansion helps in generating assembly code specifically suited to the parameters in macro call.
A model statement is visited only under specific conditions during the expansion of a macro.
AIF and AGO statement used for this purpose.

Example: evaluate A-B+C in AREG.

```
MACRO
EVAL    &X, &Y, &Z
AIF     (&Y EQ &X)      .ONLY
MOVE    RAREG           , &X
SUBA    REG,             &Y
ADD     AREG        , &Z
AGO                 .OVER
.ONLy  MOVE  RAREG,  &Z
.OVER  MEND
```

Expansion time loop

To generate many similar statements during the expansion of a macro.

This can be achieved by similar model statements in the macro.

Example:

```
MACRO
CLEAR&A
MOVERAREG, =„0‟
MOVEMAREG, &A
MOVEMAREG, &A+1
MOVEMAREG, &A+2
MEND
```

Expansion time loops can be written using expansion time variables and expansion time control transfer statement AIF and AGO.

Example:

```
MACRO
CLEAR&X, &N
LCL&M
&MSET0
MOVERAREG,=„0‟
.MOVE MOVEMAREG, &X+&M
&MSET&M+1
AIF(&M NEN).MORE
MEND
```

Other facilities for expansion time loops:
REPT statement
Syntax: REPT <expression>
<expression> should evaluate to a numerical value during macro expansion.
The statements between REPT and an ENDM statement would be processed for expansion <expression> number of times.

Example

```
MACRO
CONST10
LCL&M
&MSET1
REPT10
DC„&M‟
```

```
&MSETA&M+1
ENDM
MEND
```

IRP statement
IRP <formal parameter>, <argument-list>
Formal parameter mentioned in the statement takes successive values from the argument list.
The statements between the IRP and ENDM statements are expanded once

MACRO
CONSTS&M, &N, &Z
IRP&Z, &M=7, &N
DC„&Z‟
ENDM
MEND
A MACRO call CONSTS 4, 10 leads to declaration of 3 constants with the values 4,7 and 10.

Semantic Expansion:
Semantic expansion is the generation of instructions tailored to the requirements of a specific usage.
Example:
MACRO
CREATE_CONST&X, &Y
AIF(T‟&X EQ B).BYTE
&YDW25
AGO.OVER
.BYTEANOP
&YDB25

.OVERMEND

16.parameter in macro.

Positional parameters
Keyword parameters
Default specification of parameter
Macro with mixed parameter lists
Other uses of parameters

Positional parameters
Positional parameters
A positional formal parameter is written as &<parameter name>. The <actual parameter spec> in call on a macro using positional parameters is simply an <ordinary string>.
Step-1 find the ordinal position of XYZ in the list of formal parameters in the macro prototype statement.
Step-2 find the actual parameter specification occupying the same ordinal position in the list of actual parameters in macro call statement.
INCR A, B, AREG
The rule of positional association values of the formal parameters are:
Formal parametervalue
MEM_VALA

INCR_VALB
REGAREG
Lexical expansion of model statement now leads to the code
+MOVERAREG, A
+ADDAREG, B
+MOVEMAREG, A

Keyword parameters
<parameter name > is an ordinary string and <parameter kind> is the string „=„ in syntax rule.
The <actual parameter spec> is written as <formal parameter name>=<ordinary string>.
The keyword association rules:
Step-1 find the actual parameter specification which has the form XYZ=<ordinary string>
Step-2 Let <ordinary string> in the specification be the string ABC. Then the value of formal parameter XYZ is ABC

```
INCR_M      MEM_VAL=A, INCR_VAL=B, REG=AREG
...
INCR_M      INCR_VAL=B, REG=AREG, MEM_VAL=A

MACRO
INCR_M      &MEM_VAL=, &INCR_VAL=, &REG=
MOVER       &REG, &MEM_VAL
ADD         &REG, &INCR_VAL
MOVEM       &REG, &MEM_VAL
MEND
```

Default specification of parameters
Default specification of parameters

A default is a standard assumption in the absence of an explicit specification by programmer.
Default specification of parameters is useful in situations where a parameter has the same value in most calls.
When desired value is different from the default value, the desired value can be specified explicitly in a macro call.

Example:
Call the macro
INCR_DMEM_VAL=A, INCR_VAL=B
INCR_DINCR_VAL=B, MEM_VAL=A
INCR_DINCR_VAL=B, MEM_VAL=A, REG=BREG
MARCO DIFINITION
MACRO
INCR_D&MEM_VAL=,&INCR_VAL=,&REG=AREG
MOVER&REG, &MEM_VAL
ADD&REG, &INC_VAL
MOVEM&REG, &MEM_VAL
MEND

Macro with mixed parameter lists
Macro with mixed parameter lists
A macro may be defined to use both positional and keyword parameters.

All positional parameters must precede all keyword parameters.
Example: SUMUP A,B,G=20,H=X
Where A,B are positional parameters while G,H are keyword parameters.

Other uses of parameters
Other uses of parameters
The model statements have used formal parameters only in operand fields.
Formal parameter can also appear in the label and opcodefields of model statements.

```
      MACRO
      CALC        &X, &Y, &OP= MULT, &LAB=

&LAB      MOVER       AREG, &X
          &OP         AREG, &Y
          MOVEM       AREG, &X
          MEND
```

Expansion of the call `CALC A, B, LAB=LOOP` leads to the following code:

```
+ LOOP    MOVER       AREG, A
+         MULT        AREG, B
+         MOVEM       AREG, A
```

Advantage of interpreter.

## Advantages:

–Meaning of source statement is implemented using interpretation routine which results to simplified implementation.

–Avoid generation of machine language instruction.

–Helps make interpretation portable.

–Interpreter itself is coded in high level language.

mIMP

1. Given input mate op table,symbol table,mnemonic table generate karo.

2. Variant 1 and variant 2 with example and difference.

3. Diff bw macro and function.

4. Type of statement in macro definition.

5. Block and nested macro nu extented stack data structure with example.

6.diff bw compiler and interpreter.

7. Few terminology in linker.

8.exaplain relocation and linker algo.

9.major design issue of dd

10. Strategy suedo code

11.