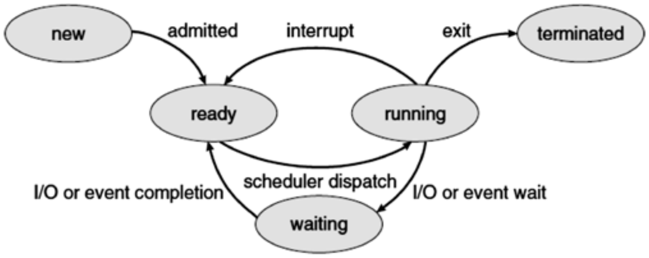
Q.2 a. Explain the following systems: (9)
i. Batch processing systems
ii. Time sharing systems
iii. Real-time operating systems

Answer:

b. Draw the process state diagram. (3)

Answer:



c. What resources are used when a thread is created? How do they differ from those used when a process is created? (4)

Answer: Because a thread is smaller than a process, thread creation typically uses fewer resources than process creation. Creating a process requires allocating a process control block (PCB), a rather large data structure. The PCB includes a memory map, list of open files, and environment variables. Allocating and managing the memory map is typically the most time-consuming activity.

Q.3 a. Most round-robin schedulers use a fixed size quantum. Give an argument in favour of small quantum and large quantum. Compare and contrast the types of systems and jobs to which both the arguments apply. (4)

Answer:
Small quantum: Using a small quantum will enhance responsiveness by frequently running all processes for a short time slice. When the ready queue has many processes that are interactive, responsiveness is very important e.g. general-purpose computer system.

Large quantum: Using a large quantum will enhance the throughput, and the CPU utilization measured with respect to real work, because there is less context switching and therefore less overhead. e.g. batch jobs.

b. Consider the following set of processes: (6)

Process Name	Arrival Time	Processing Time
A	0	7
B	1	5
C	2	2
D	3	4

Find the average turn round time for the FCFS, SJF and RR (time quantum = 4) non-preemptive CPU scheduling methods.

Answer:

	Turn Around Time (ms) (TAT)		
Process	**FCFS**	**SJF**	**RR (4 Quantum)**
A	7	7	17
B	11	17	17
C	12	7	8
D	15	10	11
Avg. TAT	**45/4= 11.25**	**41/4 = 10.25**	**53/4 =13.25**

c. Mention any three measures for Deadlock detection and avoidance. (6)

Answer:

Deadlock Detection Algorithm

A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur. Checking at each resource request has two advantages: It leads to early detection, and the algorithm is relatively simple because it is based on incremental changes to the state of the system. On the other hand, such frequent checks consume considerable processor time.

Deadlock avoidance

This method on the other hand, allows the three necessary conditions (mutual exclusion, hold and wait, no preemption) but makes judicious choices to assure that the deadlock point is never reached. As such, avoidance allows more concurrency than prevention. With deadlock avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock.

Q.4 a. Explain the working of bounded-buffer problem in synchronization. (6)

Answer: The bounded-buffer problem was commonly used to illustrate the power of synchronization primitives. We present here a general structure of this scheme, without committing ourselves to any particular implementation. We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers, respectively. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

b. Explain boot control block and volume control block used in file systems. (4)

Answer:

A bootcontrol block (per volume) can contain information needed by the system to boot an operating system from that volume. If the disk does not contain an operating system, this block can be empty. It is typically the first block of a volume. In UFS, it is called the boot block. In NTFS, it is the partition boot sector.

A volume control block (per volume) contains volume (or partition) details, such as the number of blocks in the partition, the size of the blocks, a free-block count and free-block pointers, and a free-FCB count and FCB pointers. In UFS, this is called a superblock. In NTFS, it is stored in the master file table.

c. Explain any two file sharing techniques. (6)

Answer:

DFS - Through the evolution of network and file technology, remote file-sharing methods have changed. The first implemented method involves manually transferring files between machines via programs like ftp. The second major method uses a distributed file system (DFS) in which remote directories are visible from a local machine. In some ways, the third method, the WorldWideWeb, is a reversion to the first. A browser is needed to gain access to the remote files, and separate operations (essentially a wrapper for ftp) are used to transfer files.

Clinet/server file systems allow a computer to mount one or more file systems from one or more remote machines. In this case, the machine containing the files is the server, and the machine seeking access to the files is the client. The client–server relationship is common with networked machines. Generally, the server declares that a resource is available to clients and specifies exactly which resource (in this case, which files) and exactly which clients.

Q.5 a. Consider a paged virtual memory system with 32-bit virtual addresses and 1K-byte pages. Each page table entry requires 32 bits. It is desired to limit the page table size to one page. (3+3+4)

(i) How many levels of page tables are required?

(ii) What is the size of the page table at each level?

(iii) The smaller page size could be used at the top level or the bottom level of the page table hierarchy. Which strategy consumes the least number of pages?

Answer:

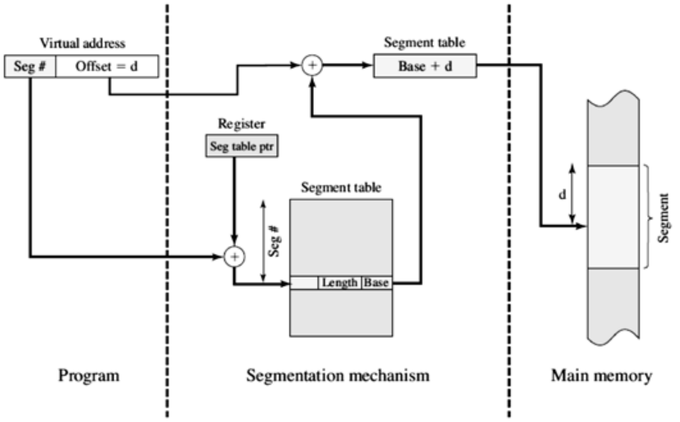
(i) Virtual memory can hold (232 bytes of main memory)/(210 bytes/page) = 222
so 22 bits are needed to specify a page in virtual memory. Each page table contains (210 bytes per page table)/(4 bytes/entry) = 28 entries. Thus, each page table can handle 8 of the required 22 bits. Therefore, 3 levels of page tables are needed.

(ii) Tables at two of the levels have 28 entries; tables at one level have 26 entries. (8 +8 + 6 = 22).

(iii) Less space is consumed if the top level has 26 entries. In that case, the second level has 26 pages with 28 entries each, and the bottom level has 214 pages with 28 entries each, for a total of 1 + 26 + 214 pages = 16,449 pages. If the middle level has 26 entries, then the number of pages is 1 + 28 + 214 pages = 16,641 pages. If the bottom level has 26 entries, then the number of tables is 1 + 28 + 216 pages = 65,973 pages.

b. Explain with a diagram how addresses are translated in a segmentation system. (6)

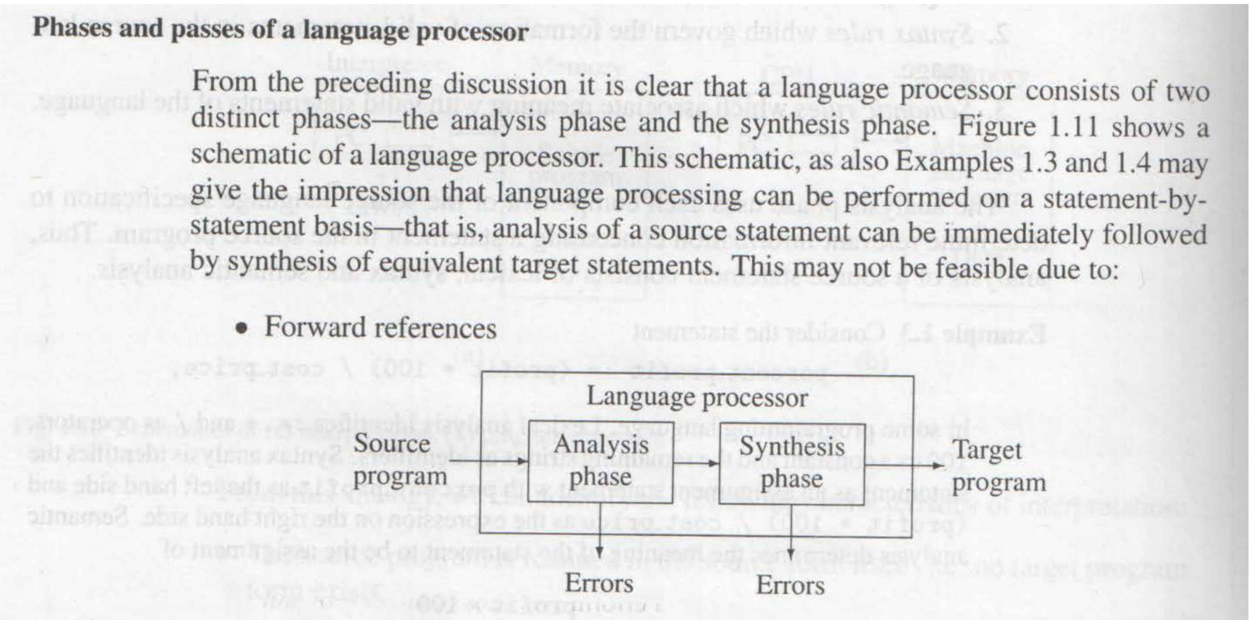
Answer:



Address Translation in a Segmentation System

Q.6 a. Explain phases and passes of language processor. (6)

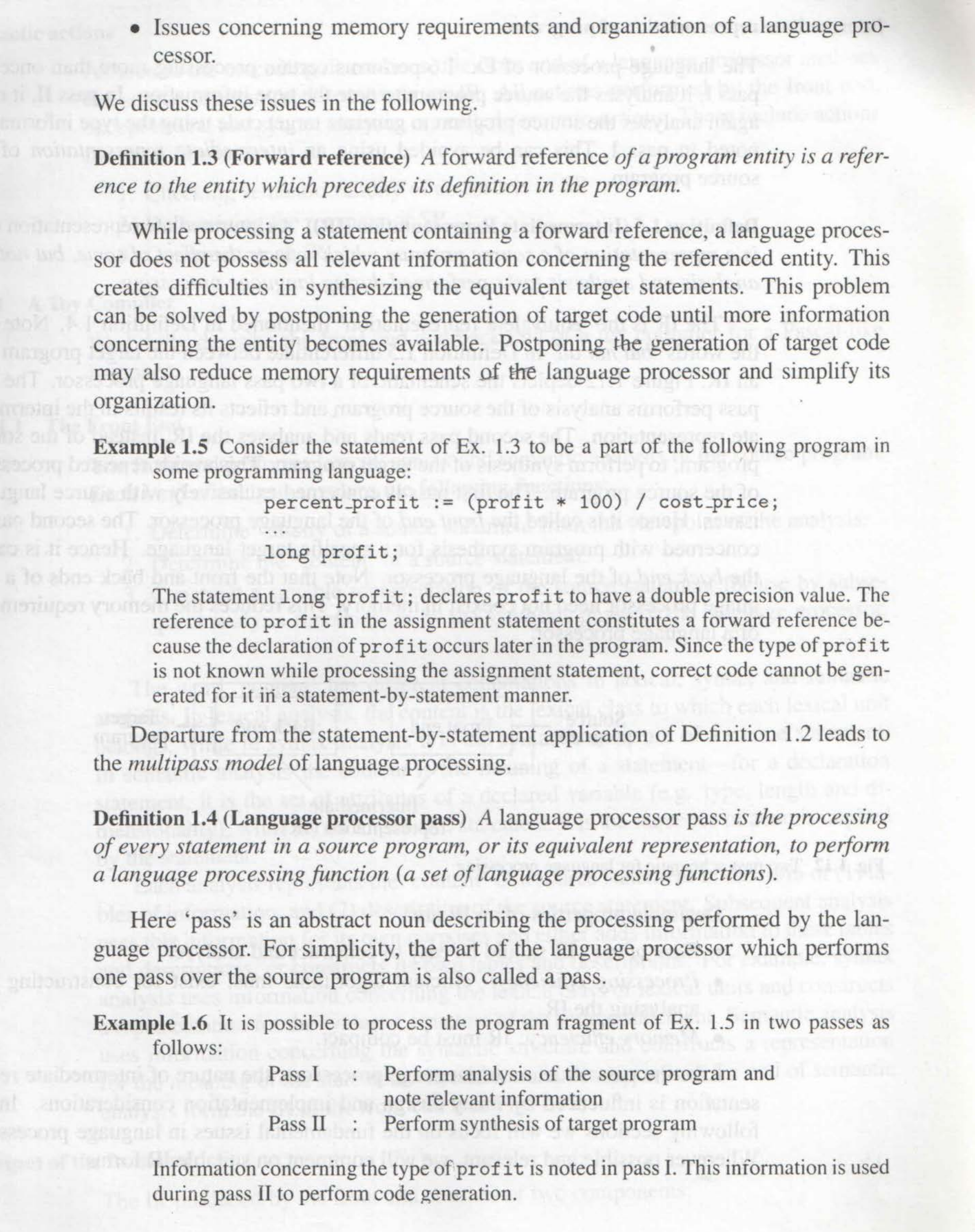
Answer:



Phases and passes of a language processor

From the preceding discussion it is clear that a language processor consists of two distinct phases—the analysis phase and the synthesis phase. Figure 1.11 shows a schematic of a language processor. This schematic, as also Examples 1.3 and 1.4 may give the impression that language processing can be performed on a statement-by-statement basis—that is, analysis of a source statement can be immediately followed by synthesis of equivalent target statements. This may not be feasible due to:

- Forward references



- Issues concerning memory requirements and organization of a language processor.

We discuss these issues in the following.

Definition 1.3 (Forward reference) *A forward reference of a program entity is a reference to the entity which precedes its definition in the program.*

While processing a statement containing a forward reference, a language processor does not possess all relevant information concerning the referenced entity. This creates difficulties in synthesizing the equivalent target statements. This problem can be solved by postponing the generation of target code until more information concerning the entity becomes available. Postponing the generation of target code may also reduce memory requirements of the language processor and simplify its organization.

Example 1.5 Consider the statement of Ex. 1.3 to be a part of the following program in some programming language:

```
percent_profit := (profit * 100) / cost_price;
...
long profit;
```

The statement `long profit;` declares `profit` to have a double precision value. The reference to `profit` in the assignment statement constitutes a forward reference because the declaration of `profit` occurs later in the program. Since the type of `profit` is not known while processing the assignment statement, correct code cannot be generated for it in a statement-by-statement manner.

Departure from the statement-by-statement application of Definition 1.2 leads to the *multipass model* of language processing.

Definition 1.4 (Language processor pass) *A language processor pass is the processing of every statement in a source program, or its equivalent representation, to perform a language processing function (a set of language processing functions).*

Here 'pass' is an abstract noun describing the processing performed by the language processor. For simplicity, the part of the language processor which performs one pass over the source program is also called a pass.

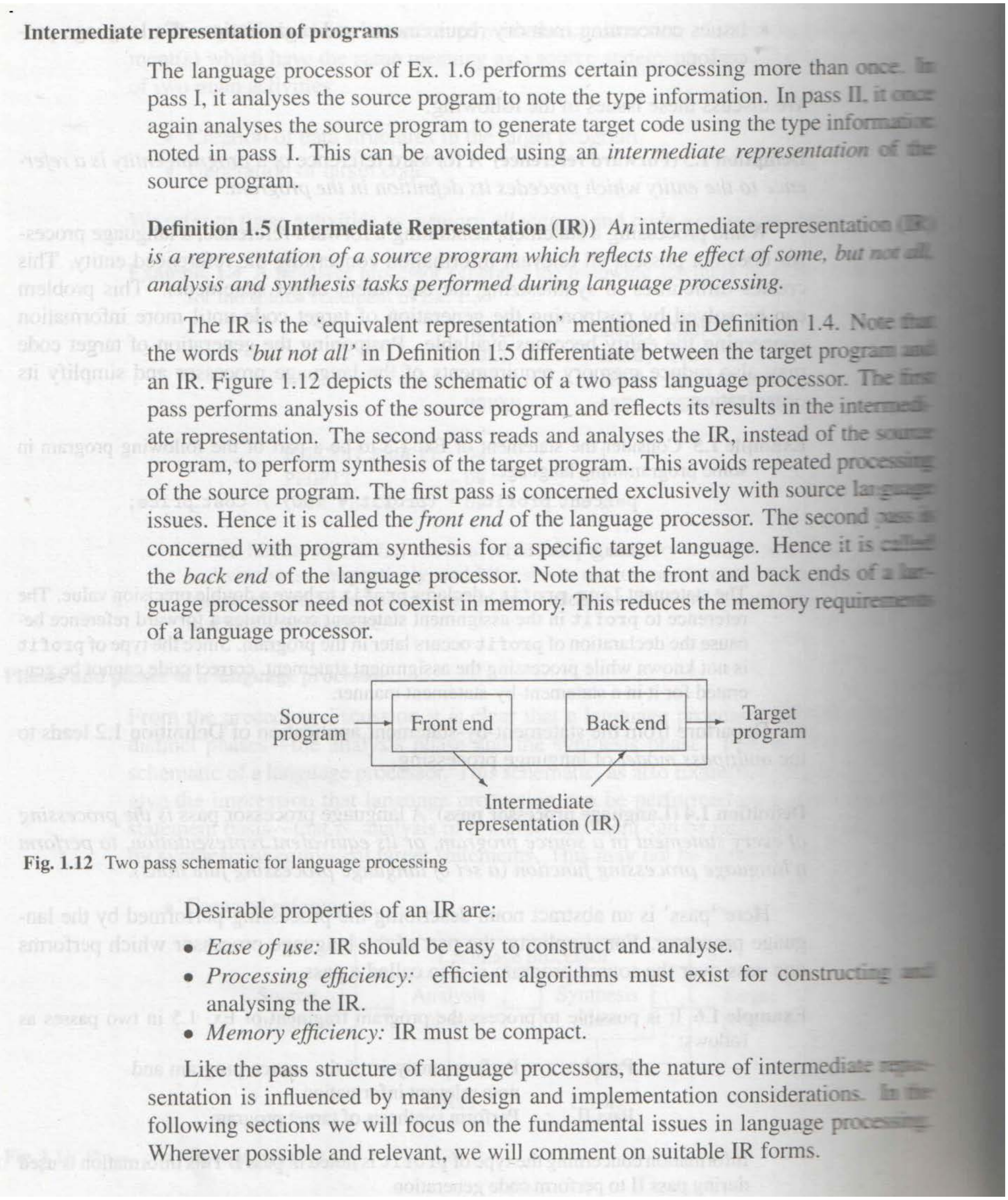
Example 1.6 It is possible to process the program fragment of Ex. 1.5 in two passes as follows:

Pass I	:	Perform analysis of the source program and note relevant information
Pass II	:	Perform synthesis of target program

Information concerning the type of `profit` is noted in pass I. This information is used during pass II to perform code generation.

b. What is Intermediate Representation (IR)? What are the desirable properties of an IR? (4)

Answer:



Intermediate representation of programs

The language processor of Ex. 1.6 performs certain processing more than once. In pass I, it analyses the source program to note the type information. In pass II, it once again analyses the source program to generate target code using the type information noted in pass I. This can be avoided using an *intermediate representation* of the source program.

Definition 1.5 (Intermediate Representation (IR)) *An intermediate representation (IR) is a representation of a source program which reflects the effect of some, but not all, analysis and synthesis tasks performed during language processing.*

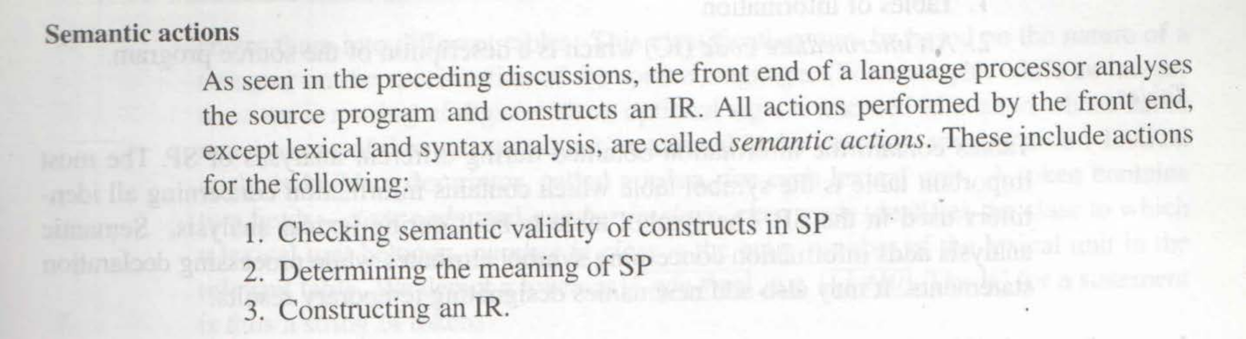
The IR is the 'equivalent representation' mentioned in Definition 1.4. Note that the words *'but not all'* in Definition 1.5 differentiate between the target program and an IR. Figure 1.12 depicts the schematic of a two pass language processor. The first pass performs analysis of the source program and reflects its results in the intermediate representation. The second pass reads and analyses the IR, instead of the source program, to perform synthesis of the target program. This avoids repeated processing of the source program. The first pass is concerned exclusively with source language issues. Hence it is called the *front end* of the language processor. The second pass is concerned with program synthesis for a specific target language. Hence it is called the *back end* of the language processor. Note that the front and back ends of a language processor need not coexist in memory. This reduces the memory requirements of a language processor.

Fig. 1.12 Two pass schematic for language processing

Desirable properties of an IR are:

- *Ease of use:* IR should be easy to construct and analyse.
- *Processing efficiency:* efficient algorithms must exist for constructing and analysing the IR.
- *Memory efficiency:* IR must be compact.

Like the pass structure of language processors, the nature of intermediate representation is influenced by many design and implementation considerations. In the following sections we will focus on the fundamental issues in language processing. Wherever possible and relevant, we will comment on suitable IR forms.



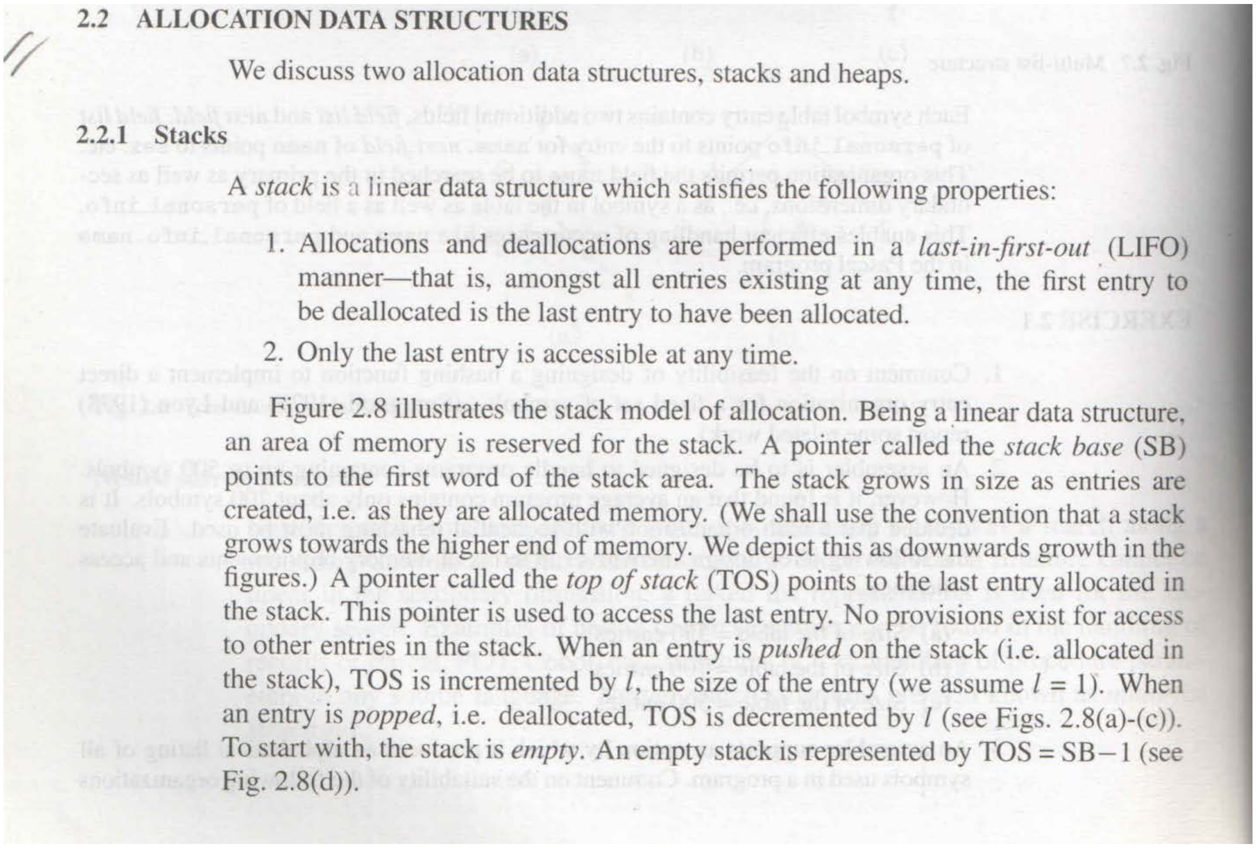
Semantic actions

As seen in the preceding discussions, the front end of a language processor analyses the source program and constructs an IR. All actions performed by the front end, except lexical and syntax analysis, are called *semantic actions*. These include actions for the following:

1. Checking semantic validity of constructs in SP
2. Determining the meaning of SP
3. Constructing an IR.

c. Explain the allocation data structures: stacks and heaps used in language processing. (6)

Answer:



2.2 ALLOCATION DATA STRUCTURES

We discuss two allocation data structures, stacks and heaps.

2.2.1 Stacks

A *stack* is a linear data structure which satisfies the following properties:

1. Allocations and deallocations are performed in a *last-in-first-out* (LIFO) manner—that is, amongst all entries existing at any time, the first entry to be deallocated is the last entry to have been allocated.
2. Only the last entry is accessible at any time.

Figure 2.8 illustrates the stack model of allocation. Being a linear data structure, an area of memory is reserved for the stack. A pointer called the *stack base* (SB) points to the first word of the stack area. The stack grows in size as entries are created, i.e. as they are allocated memory. (We shall use the convention that a stack grows towards the higher end of memory. We depict this as downwards growth in the figures.) A pointer called the *top of stack* (TOS) points to the last entry allocated in the stack. This pointer is used to access the last entry. No provisions exist for access to other entries in the stack. When an entry is *pushed* on the stack (i.e. allocated in the stack), TOS is incremented by l, the size of the entry (we assume $l = 1$). When an entry is *popped*, i.e. deallocated, TOS is decremented by l (see Figs. 2.8(a)-(c)). To start with, the stack is *empty*. An empty stack is represented by TOS = SB−1 (see Fig. 2.8(d)).

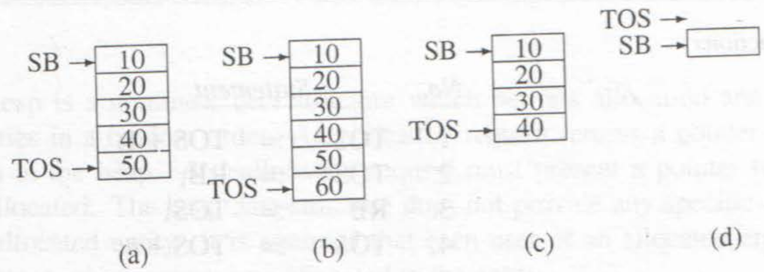


Fig. 2.8 Stack model of allocation

Extended stack model

The LIFO nature of stacks is useful when the lifetimes of the allocated entities follow the LIFO order. However, some extensions are needed in the simple stack model because all entities may not be of the same size. The size of an entity is assumed to be an integral multiple of the size of a stack entry. To allocate an entity, a *record* is created in the stack, where the record consists of a set of consecutive stack entries. For simplicity, the size of a stack entry, i.e. *l*, is assumed to be one word. Figure 2.9(a) shows the extended stack model. In addition to SB and TOS, two new pointers exist in the model:

1. A *record base pointer* (RB) pointing to the first word of the last record in stack.
2. The first word of each record is a *reserved pointer*. This pointer is used for housekeeping purposes as explained below.

The allocation and deallocation time actions in the extended stack model are described in the following paragraphs (see Fig. 2.9(b)–(c)).

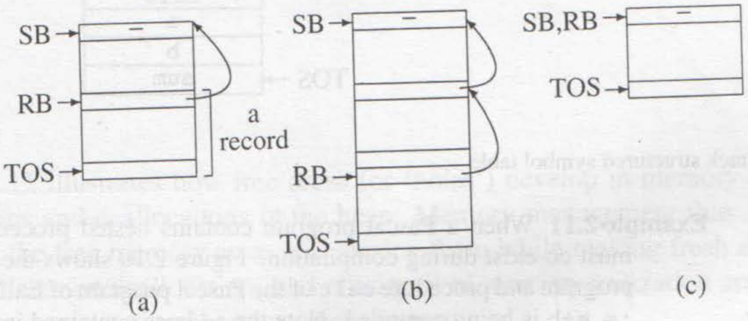
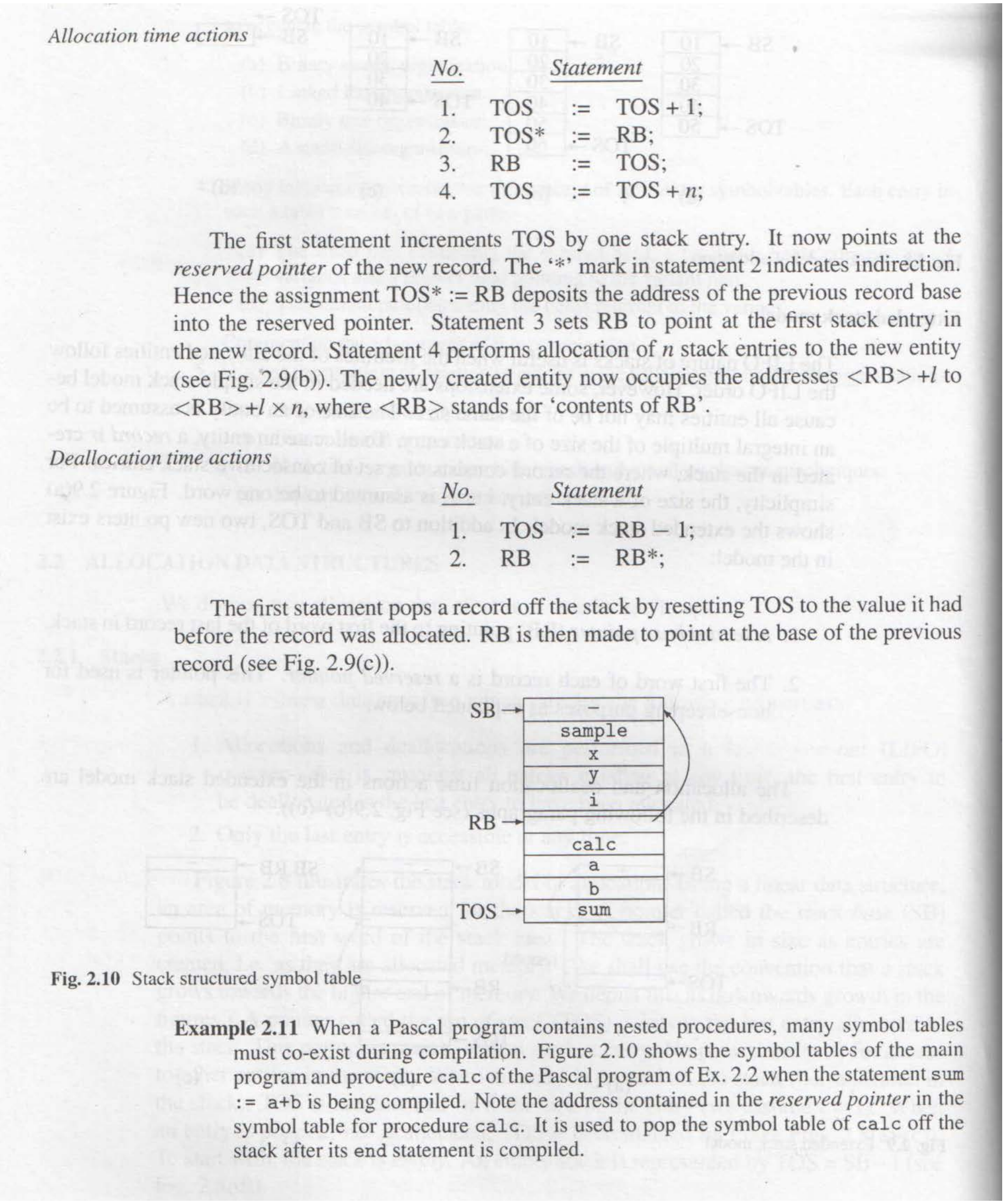


Fig. 2.9 Extended stack model



Allocation time actions

No.	*Statement*		
1.	TOS	:=	TOS + 1;
2.	TOS*	:=	RB;
3.	RB	:=	TOS;
4.	TOS	:=	TOS + *n*;

The first statement increments TOS by one stack entry. It now points at the *reserved pointer* of the new record. The '*' mark in statement 2 indicates indirection. Hence the assignment TOS* := RB deposits the address of the previous record base into the reserved pointer. Statement 3 sets RB to point at the first stack entry in the new record. Statement 4 performs allocation of *n* stack entries to the new entity (see Fig. 2.9(b)). The newly created entity now occupies the addresses $<$RB$>$ $+l$ to $<$RB$>$ $+l \times n$, where $<$RB$>$ stands for 'contents of RB'.

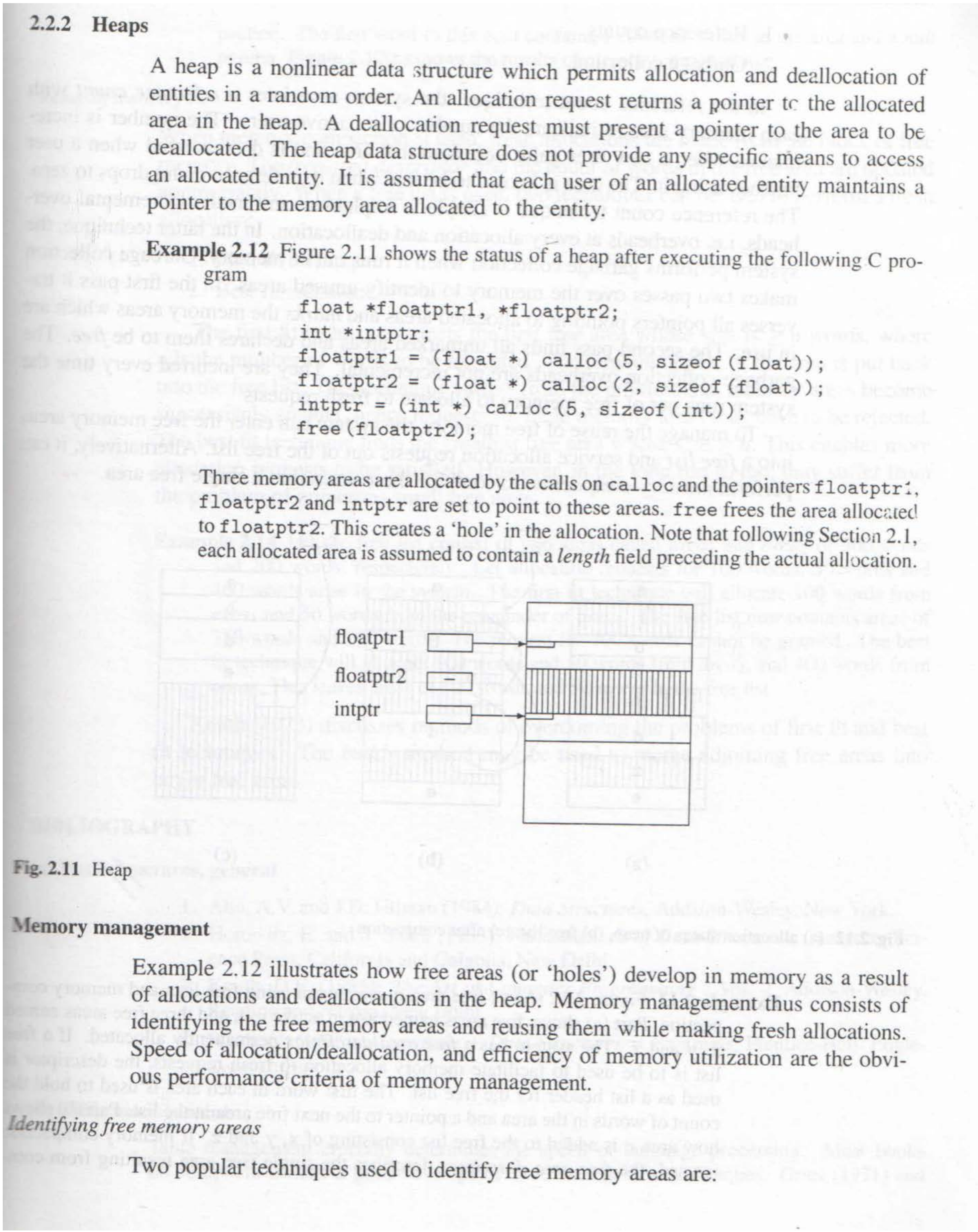
Deallocation time actions

No.	*Statement*		
1.	TOS	:=	RB − 1;
2.	RB	:=	RB*;

The first statement pops a record off the stack by resetting TOS to the value it had before the record was allocated. RB is then made to point at the base of the previous record (see Fig. 2.9(c)).

Fig. 2.10 Stack structured symbol table

Example 2.11 When a Pascal program contains nested procedures, many symbol tables must co-exist during compilation. Figure 2.10 shows the symbol tables of the main program and procedure `calc` of the Pascal program of Ex. 2.2 when the statement `sum := a+b` is being compiled. Note the address contained in the *reserved pointer* in the symbol table for procedure `calc`. It is used to pop the symbol table of `calc` off the stack after its `end` statement is compiled.



2.2.2 Heaps

A heap is a nonlinear data structure which permits allocation and deallocation of entities in a random order. An allocation request returns a pointer tc the allocated area in the heap. A deallocation request must present a pointer to the area to be deallocated. The heap data structure does not provide any specific means to access an allocated entity. It is assumed that each user of an allocated entity maintains a pointer to the memory area allocated to the entity.

Example 2.12 Figure 2.11 shows the status of a heap after executing the following C program

```
float *floatptr1, *floatptr2;
int *intptr;
floatptr1 = (float *) calloc(5, sizeof(float));
floatptr2 = (float *) calloc(2, sizeof(float));
intptr = (int *) calloc(5, sizeof(int));
free(floatptr2);
```

Three memory areas are allocated by the calls on `calloc` and the pointers `floatptr1`, `floatptr2` and `intptr` are set to point to these areas. `free` frees the area allocated to `floatptr2`. This creates a 'hole' in the allocation. Note that following Section 2.1, each allocated area is assumed to contain a *length* field preceding the actual allocation.

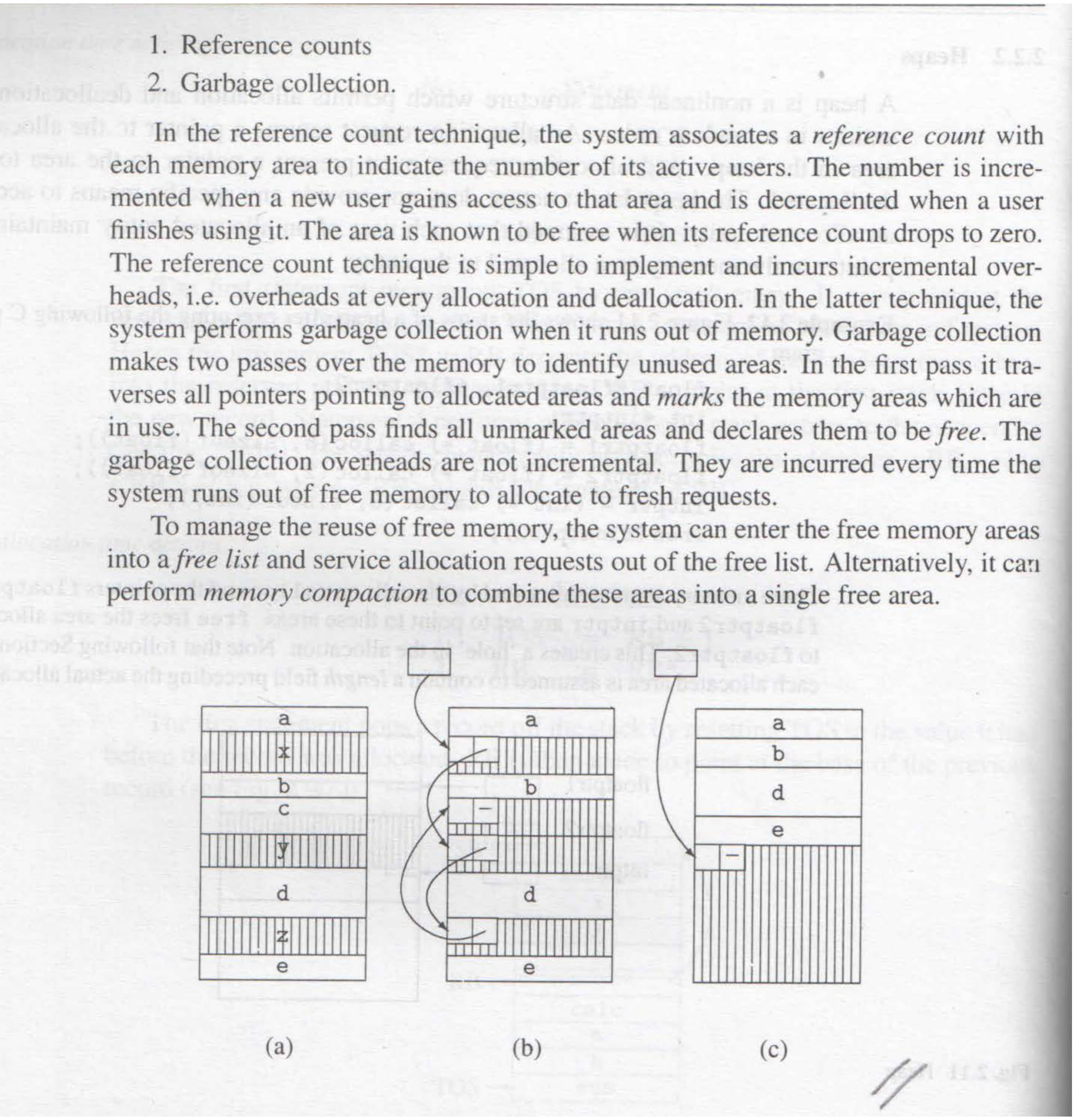
Fig. 2.11 Heap

Memory management

Example 2.12 illustrates how free areas (or 'holes') develop in memory as a result of allocations and deallocations in the heap. Memory management thus consists of identifying the free memory areas and reusing them while making fresh allocations. Speed of allocation/deallocation, and efficiency of memory utilization are the obvious performance criteria of memory management.

Identifying free memory areas

Two popular techniques used to identify free memory areas are:



1. Reference counts
2. Garbage collection.

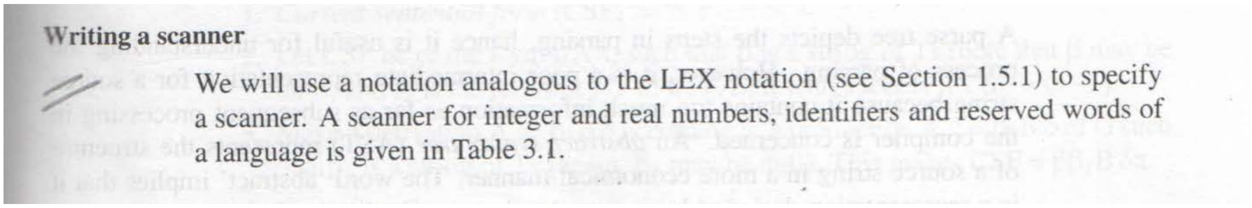
In the reference count technique, the system associates a *reference count* with each memory area to indicate the number of its active users. The number is incremented when a new user gains access to that area and is decremented when a user finishes using it. The area is known to be free when its reference count drops to zero. The reference count technique is simple to implement and incurs incremental overheads, i.e. overheads at every allocation and deallocation. In the latter technique, the system performs garbage collection when it runs out of memory. Garbage collection makes two passes over the memory to identify unused areas. In the first pass it traverses all pointers pointing to allocated areas and *marks* the memory areas which are in use. The second pass finds all unmarked areas and declares them to be *free*. The garbage collection overheads are not incremental. They are incurred every time the system runs out of free memory to allocate to fresh requests.

To manage the reuse of free memory, the system can enter the free memory areas into a *free list* and service allocation requests out of the free list. Alternatively, it can perform *memory compaction* to combine these areas into a single free area.

(a) (b) (c)

Q.7 a. Give the specifications of scanner with regular expression and respective semantic actions. (6)

Answer:



Writing a scanner

We will use a notation analogous to the LEX notation (see Section 1.5.1) to specify a scanner. A scanner for integer and real numbers, identifiers and reserved words of a language is given in Table 3.1.

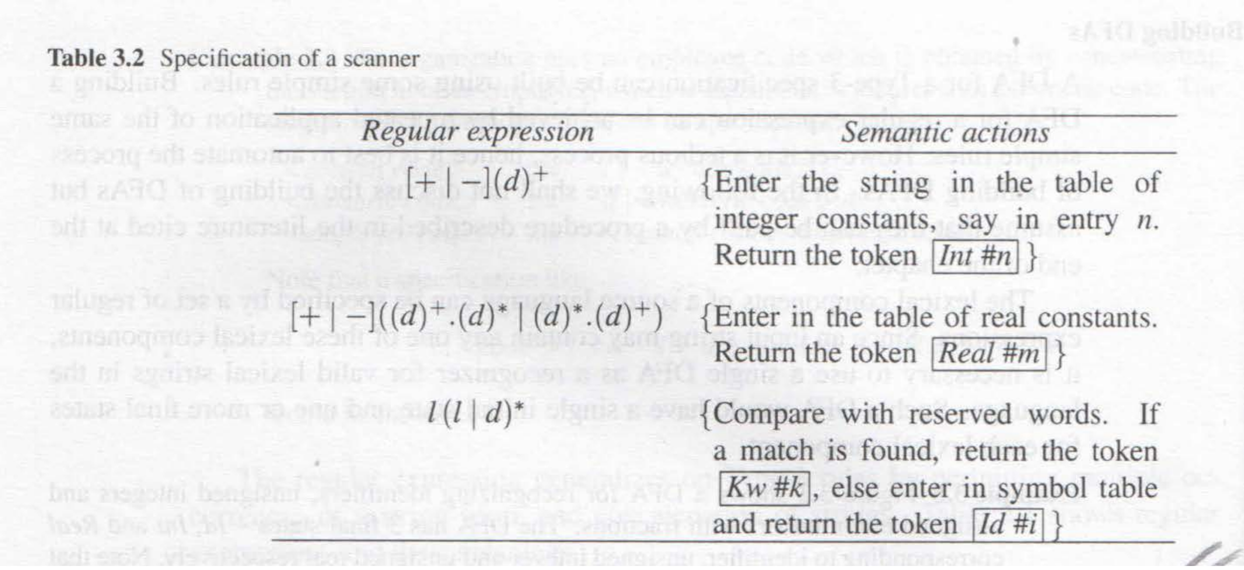
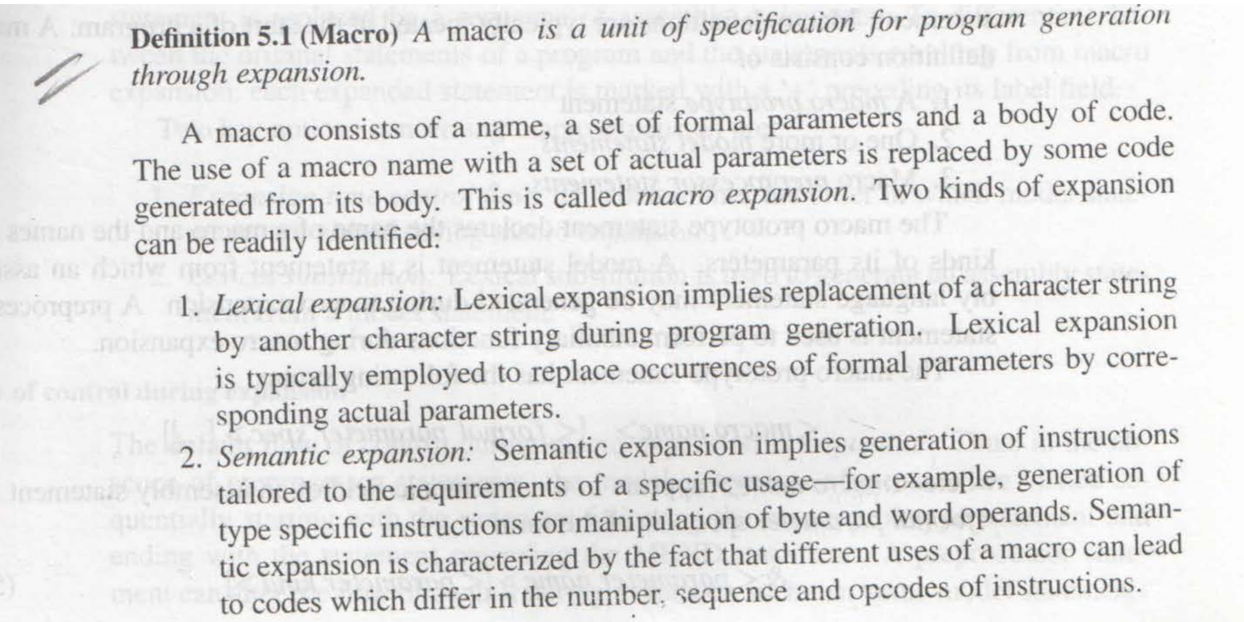


Table 3.2 Specification of a scanner

Regular expression	*Semantic actions*
$[+ \mid -](d)^+$	{Enter the string in the table of integer constants, say in entry *n*. Return the token $\boxed{Int\ \#n}$}
$[+ \mid -]((d)^+.(d)^* \mid (d)^*.(d)^+)$	{Enter in the table of real constants. Return the token $\boxed{Real\ \#m}$}
$l\,(l \mid d)^*$	{Compare with reserved words. If a match is found, return the token $\boxed{Kw\ \#k}$, else enter in symbol table and return the token $\boxed{Id\ \#i}$}

b. What is macro? Identify and explain the different kinds of macro expansion. (4)

Answer:



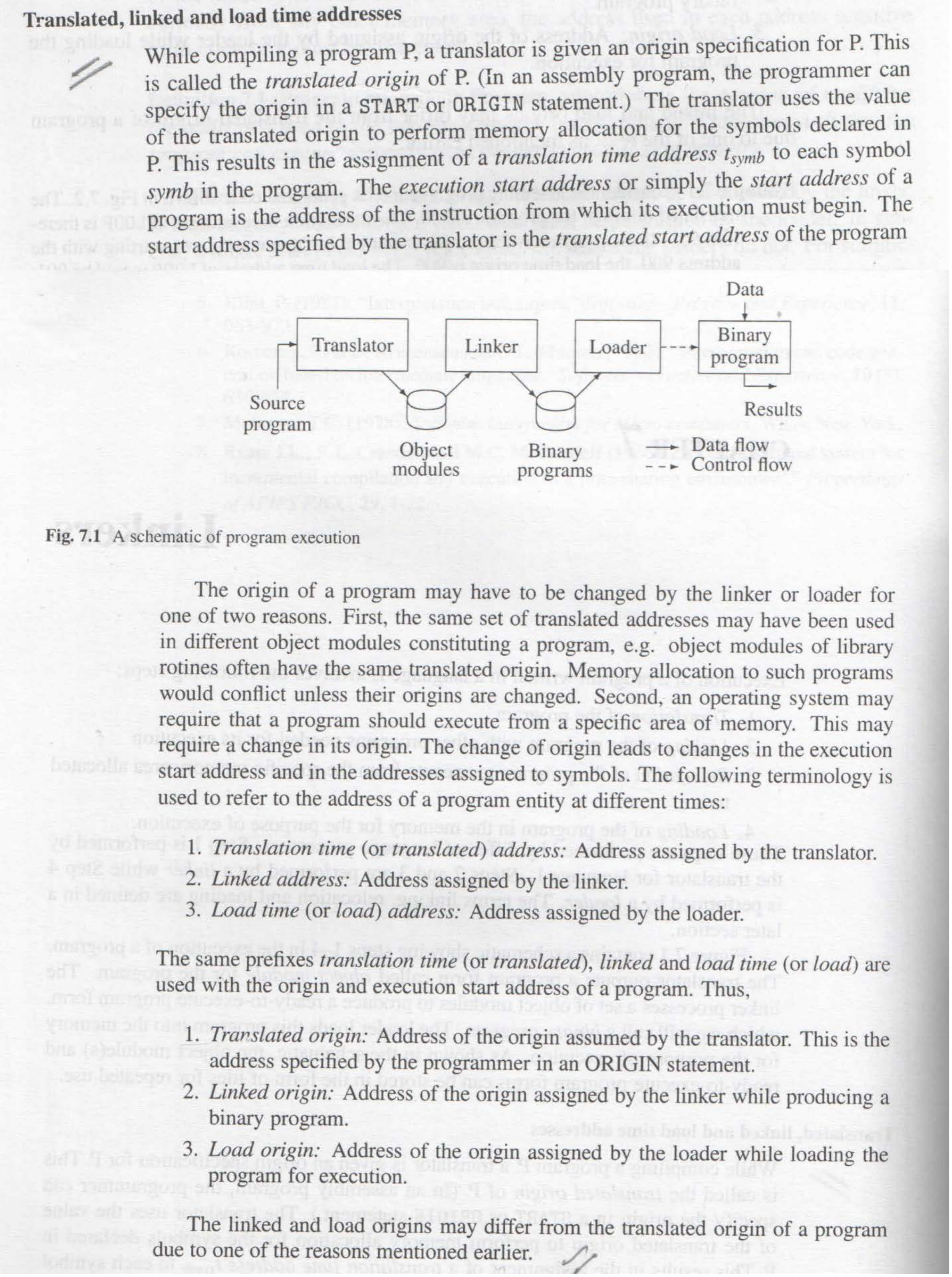
Definition 5.1 (Macro) *A* macro *is a unit of specification for program generation through expansion.*

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 readily identified:

1. *Lexical expansion:* Lexical expansion implies replacement of a character string by another character string during program generation. Lexical expansion is typically employed to replace occurrences of formal parameters by corresponding actual parameters.
2. *Semantic expansion:* Semantic expansion implies generation of instructions tailored to the requirements of a specific usage—for example, generation of type specific instructions for manipulation of byte and word operands. 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.

c. What are the different steps in execution of a program? Explain with the help of a diagram. (6)

Answer:



Translated, linked and load time addresses

While compiling a program P, a translator is given an origin specification for P. This is called the *translated origin* of P. (In an assembly program, the programmer can specify the origin in a `START` or `ORIGIN` statement.) The translator uses the value of the translated origin to perform memory allocation for the symbols declared in P. This results in the assignment of a *translation time address* t_{symb} to each symbol *symb* in the program. The *execution start address* or simply the *start address* of a program is the address of the instruction from which its execution must begin. The start address specified by the translator is the *translated start address* of the program

Fig. 7.1 A schematic of program execution

The origin of a program may have to be changed by the linker or loader for one of two reasons. First, the same set of translated addresses may have been used in different object modules constituting a program, e.g. object modules of library rotines often have the same translated origin. Memory allocation to such programs would conflict unless their origins are changed. Second, an operating system may require that a program should execute from a specific area of memory. This may require a change in its origin. The change of origin leads to changes in the execution start address and in the addresses assigned to symbols. The following terminology is used to refer to the address of a program entity at different times:

1. *Translation time* (or *translated*) *address:* Address assigned by the translator.
2. *Linked address:* Address assigned by the linker.
3. *Load time* (or *load*) *address:* Address assigned by the loader.

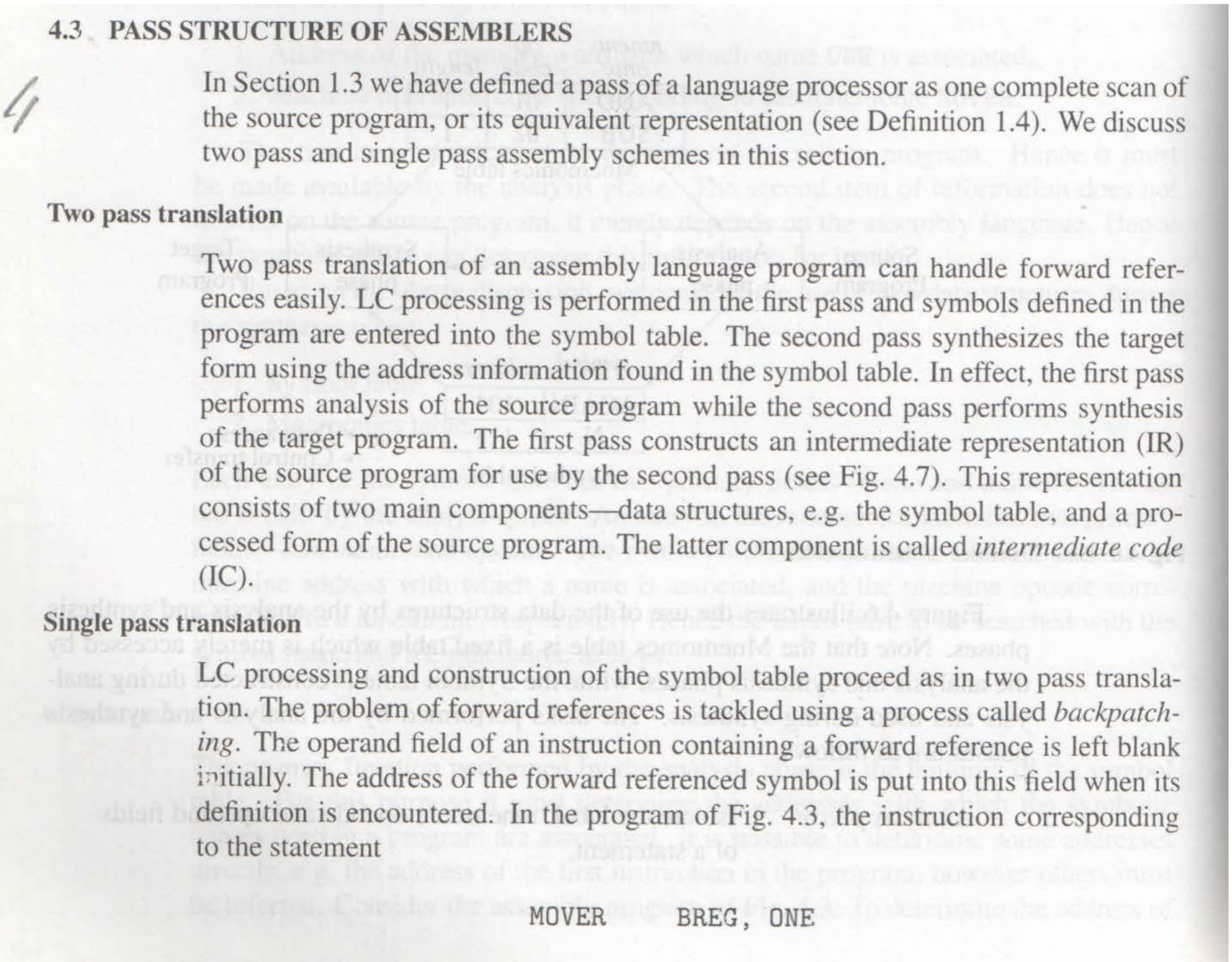
The same prefixes *translation time* (or *translated*), *linked* and *load time* (or *load*) are used with the origin and execution start address of a program. Thus,

1. *Translated origin:* Address of the origin assumed by the translator. This is the address specified by the programmer in an ORIGIN statement.
2. *Linked origin:* Address of the origin assigned by the linker while producing a binary program.
3. *Load origin:* Address of the origin assigned by the loader while loading the program for execution.

The linked and load origins may differ from the translated origin of a program due to one of the reasons mentioned earlier.

Q.8 a. Explain the pass structures of assemblers. (6)

Answer:



4.3 PASS STRUCTURE OF ASSEMBLERS

4

In Section 1.3 we have defined a pass of a language processor as one complete scan of the source program, or its equivalent representation (see Definition 1.4). We discuss two pass and single pass assembly schemes in this section.

Two pass translation

Two pass translation of an assembly language program can handle forward references easily. LC processing is performed in the first pass and symbols defined in the program are entered into the symbol table. The second pass synthesizes the target form using the address information found in the symbol table. In effect, the first pass performs analysis of the source program while the second pass performs synthesis of the target program. The first pass constructs an intermediate representation (IR) of the source program for use by the second pass (see Fig. 4.7). This representation consists of two main components—data structures, e.g. the symbol table, and a processed form of the source program. The latter component is called *intermediate code* (IC).

Single pass translation

LC processing and construction of the symbol table proceed as in two pass translation. The problem of forward references is tackled using a process called *backpatching*. The operand field of an instruction containing a forward reference is left blank initially. The address of the forward referenced symbol is put into this field when its definition is encountered. In the program of Fig. 4.3, the instruction corresponding to the statement

```
MOVER     BREG, ONE
```

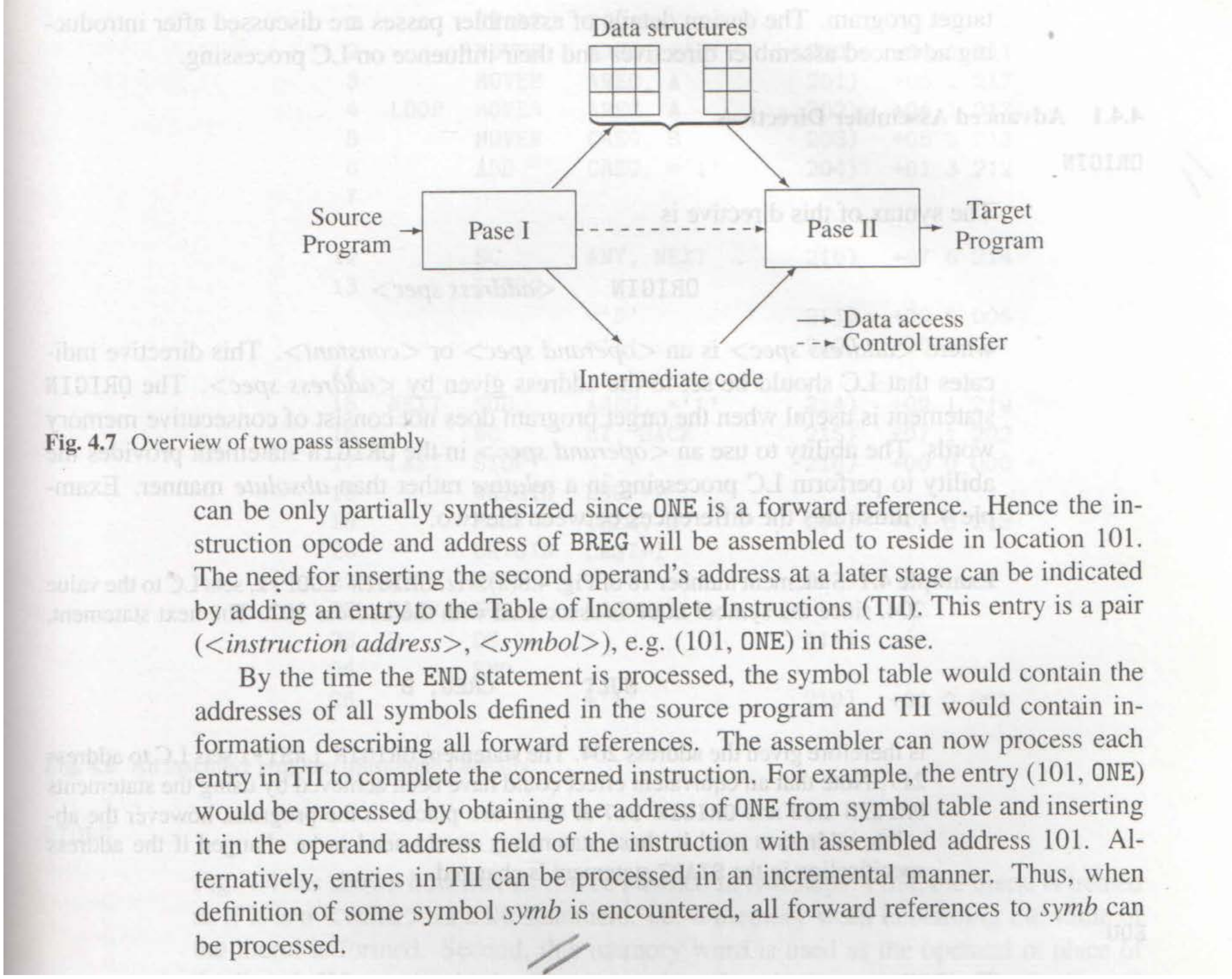

Data structures
Source Program
Pase I
Pase II
Target Program
Data access
Control transfer
Intermediate code

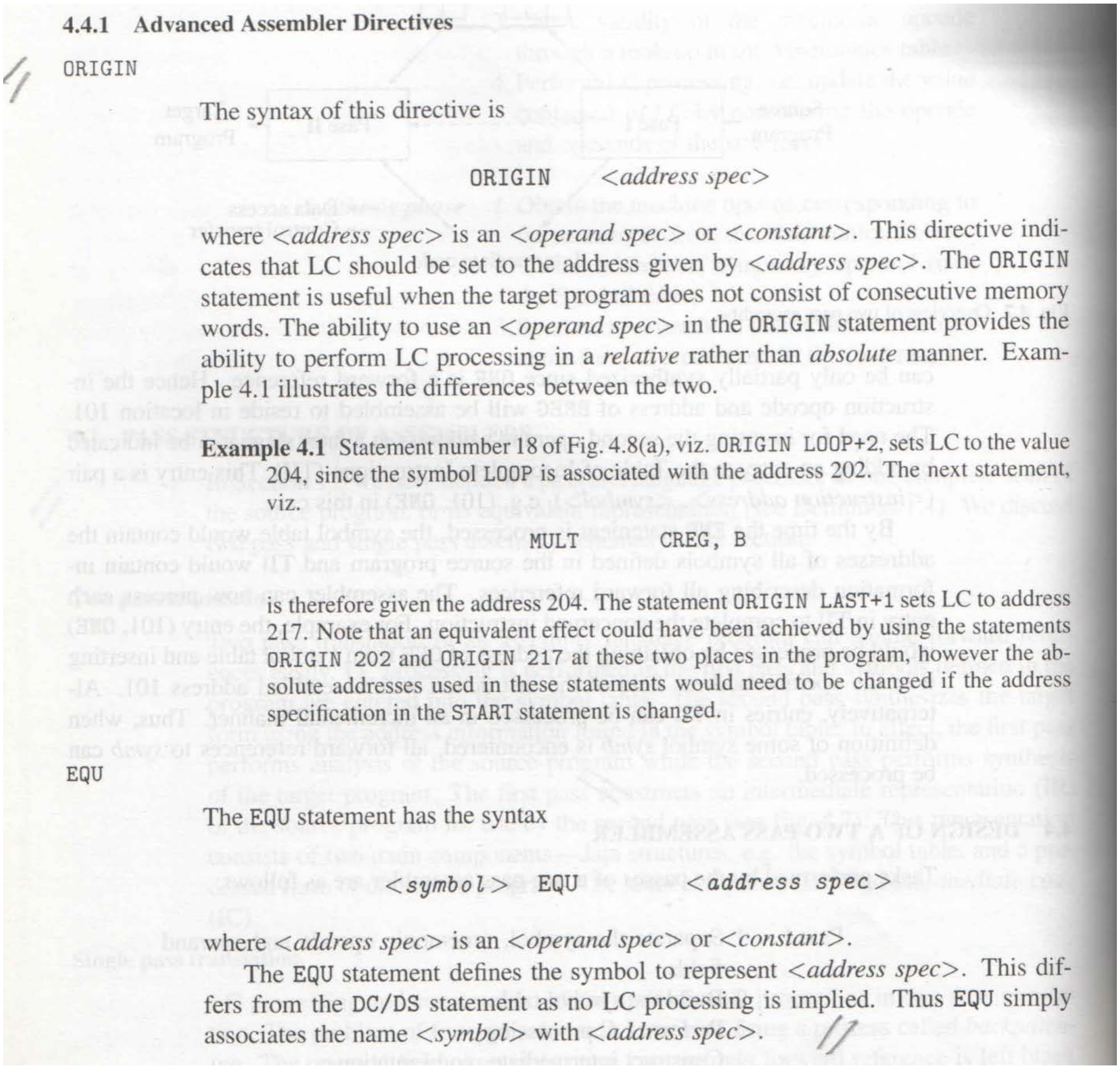
Fig. 4.7 Overview of two pass assembly

can be only partially synthesized since ONE is a forward reference. Hence the instruction opcode and address of BREG will be assembled to reside in location 101. The need for inserting the second operand's address at a later stage can be indicated by adding an entry to the Table of Incomplete Instructions (TII). This entry is a pair (<*instruction address*>, <*symbol*>), e.g. (101, ONE) in this case.

By the time the END statement is processed, the symbol table would contain the addresses of all symbols defined in the source program and TII would contain information describing all forward references. The assembler can now process each entry in TII to complete the concerned instruction. For example, the entry (101, ONE) would be processed by obtaining the address of ONE from symbol table and inserting it in the operand address field of the instruction with assembled address 101. Alternatively, entries in TII can be processed in an incremental manner. Thus, when definition of some symbol *symb* is encountered, all forward references to *symb* can be processed.

b. What are the advantages of assembler directives. (4)

Answer:



4.4.1 Advanced Assembler Directives

`ORIGIN`

The syntax of this directive is

```
ORIGIN    <address spec>
```

where *<address spec>* is an *<operand spec>* or *<constant>*. This directive indicates that LC should be set to the address given by *<address spec>*. The `ORIGIN` statement is useful when the target program does not consist of consecutive memory words. The ability to use an *<operand spec>* in the `ORIGIN` statement provides the ability to perform LC processing in a *relative* rather than *absolute* manner. Example 4.1 illustrates the differences between the two.

Example 4.1 Statement number 18 of Fig. 4.8(a), viz. `ORIGIN LOOP+2`, sets LC to the value 204, since the symbol `LOOP` is associated with the address 202. The next statement, viz.

```
MULT    CREG, B
```

is therefore given the address 204. The statement `ORIGIN LAST+1` sets LC to address 217. Note that an equivalent effect could have been achieved by using the statements `ORIGIN 202` and `ORIGIN 217` at these two places in the program, however the absolute addresses used in these statements would need to be changed if the address specification in the `START` statement is changed.

`EQU`

The `EQU` statement has the syntax

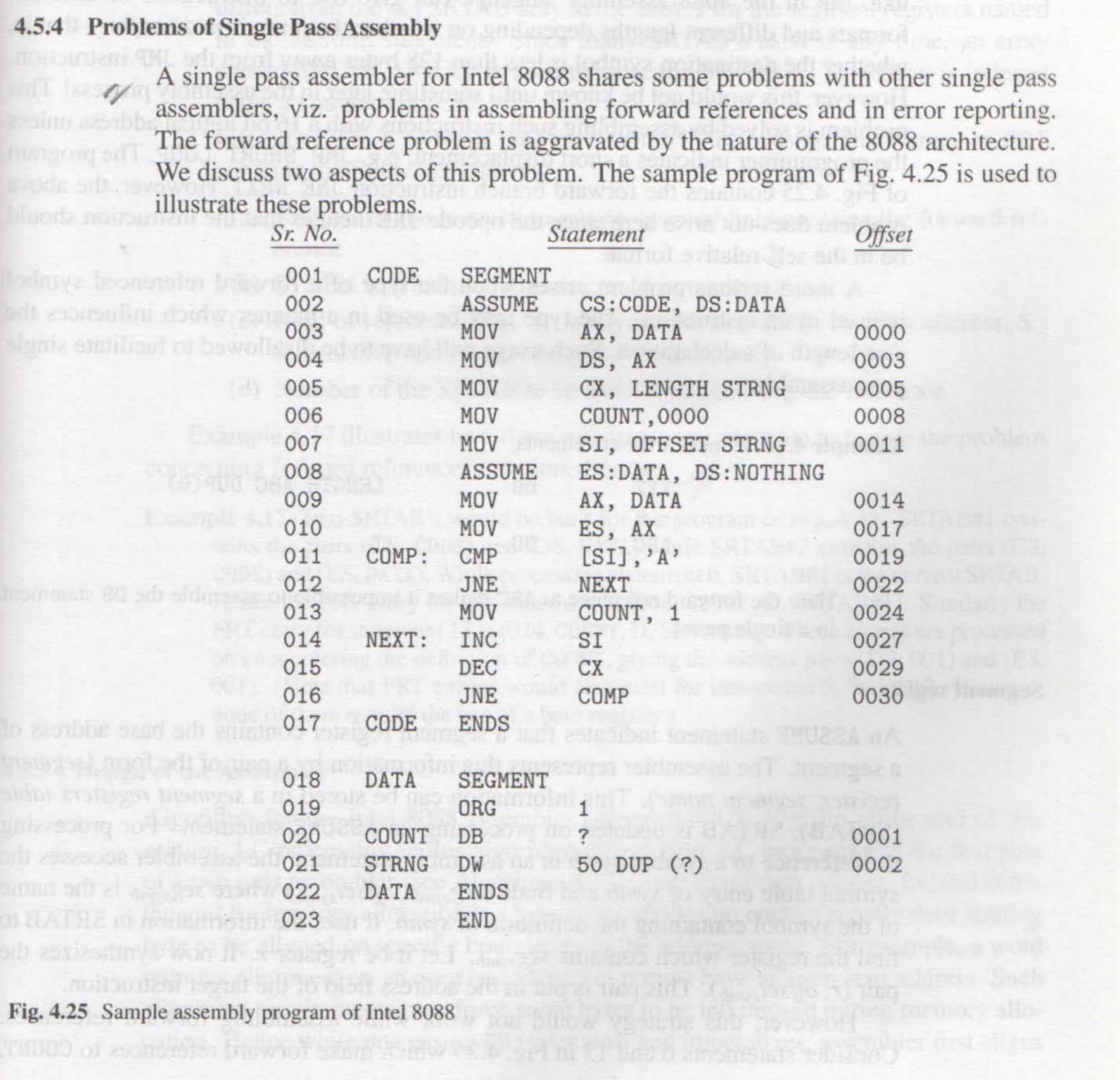
```
<symbol>  EQU    <address spec>
```

where *<address spec>* is an *<operand spec>* or *<constant>*.

The `EQU` statement defines the symbol to represent *<address spec>*. This differs from the `DC/DS` statement as no LC processing is implied. Thus `EQU` simply associates the name *<symbol>* with *<address spec>*.

c. **What are the problems of single pass assembler and their respective solutions? (3+3)**

Answer:



4.5.4 Problems of Single Pass Assembly

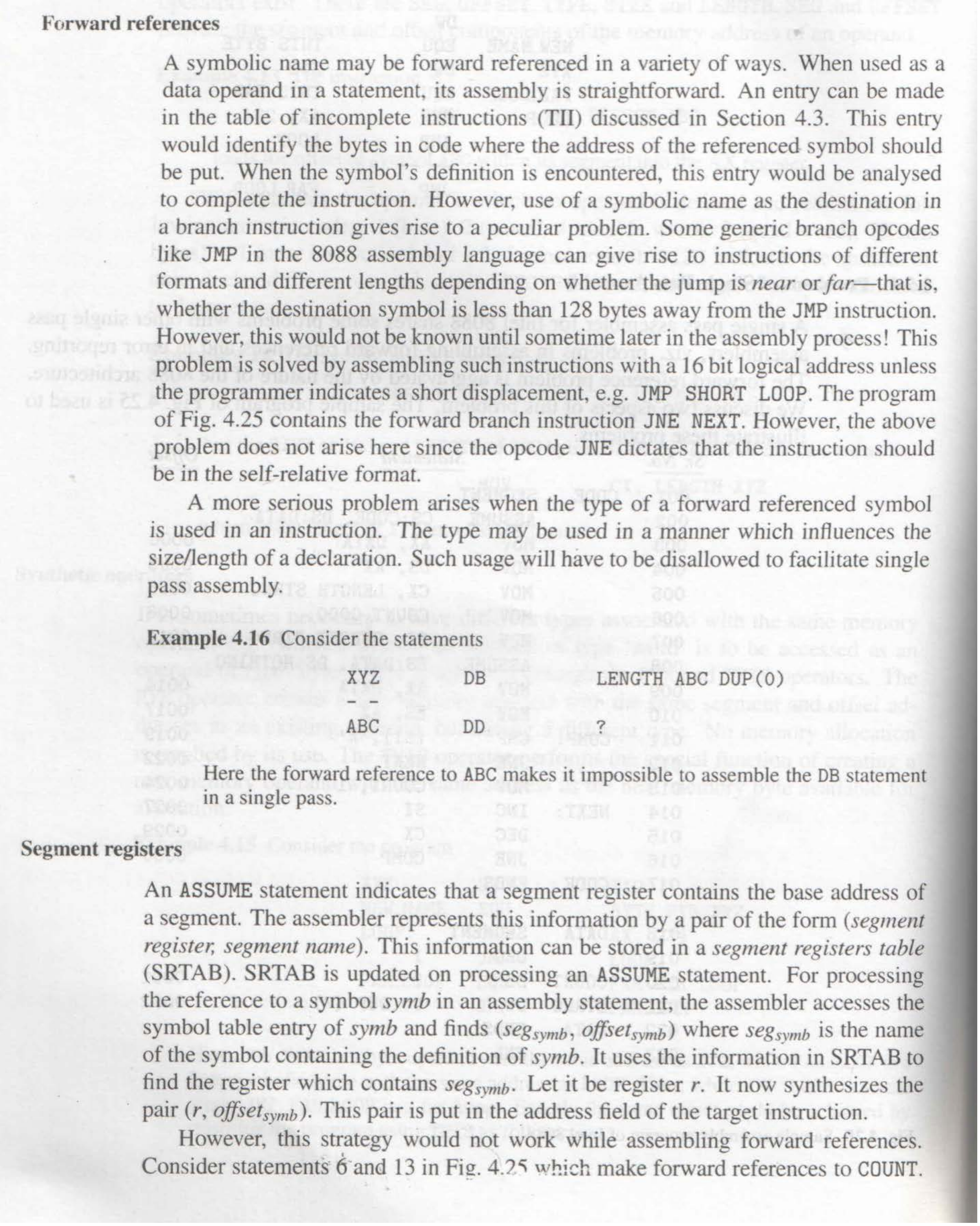
A single pass assembler for Intel 8088 shares some problems with other single pass assemblers, viz. problems in assembling forward references and in error reporting. The forward reference problem is aggravated by the nature of the 8088 architecture. We discuss two aspects of this problem. The sample program of Fig. 4.25 is used to illustrate these problems.

```
Sr. No.                Statement                   Offset

001     CODE     SEGMENT
002              ASSUME   CS:CODE, DS:DATA
003              MOV      AX, DATA                 0000
004              MOV      DS, AX                   0003
005              MOV      CX, LENGTH STRNG         0005
006              MOV      COUNT,0000               0008
007              MOV      SI, OFFSET STRNG         0011
008              ASSUME   ES:DATA, DS:NOTHING
009              MOV      AX, DATA                 0014
010              MOV      ES, AX                   0017
011     COMP:    CMP      [SI],'A'                 0019
012              JNE      NEXT                     0022
013              MOV      COUNT, 1                 0024
014     NEXT:    INC      SI                       0027
015              DEC      CX                       0029
016              JNE      COMP                     0030
017     CODE     ENDS

018     DATA     SEGMENT
019              ORG      1
020     COUNT    DB       ?                        0001
021     STRNG    DW       50 DUP (?)               0002
022     DATA     ENDS
023              END
```

Fig. 4.25 Sample assembly program of Intel 8088



Forward references

A symbolic name may be forward referenced in a variety of ways. When used as a data operand in a statement, its assembly is straightforward. An entry can be made in the table of incomplete instructions (TII) discussed in Section 4.3. This entry would identify the bytes in code where the address of the referenced symbol should be put. When the symbol's definition is encountered, this entry would be analysed to complete the instruction. However, use of a symbolic name as the destination in a branch instruction gives rise to a peculiar problem. Some generic branch opcodes like JMP in the 8088 assembly language can give rise to instructions of different formats and different lengths depending on whether the jump is *near* or *far*—that is, whether the destination symbol is less than 128 bytes away from the JMP instruction. However, this would not be known until sometime later in the assembly process! This problem is solved by assembling such instructions with a 16 bit logical address unless the programmer indicates a short displacement, e.g. JMP SHORT LOOP. The program of Fig. 4.25 contains the forward branch instruction JNE NEXT. However, the above problem does not arise here since the opcode JNE dictates that the instruction should be in the self-relative format.

A more serious problem arises when the type of a forward referenced symbol is used in an instruction. The type may be used in a manner which influences the size/length of a declaration. Such usage will have to be disallowed to facilitate single pass assembly.

Example 4.16 Consider the statements

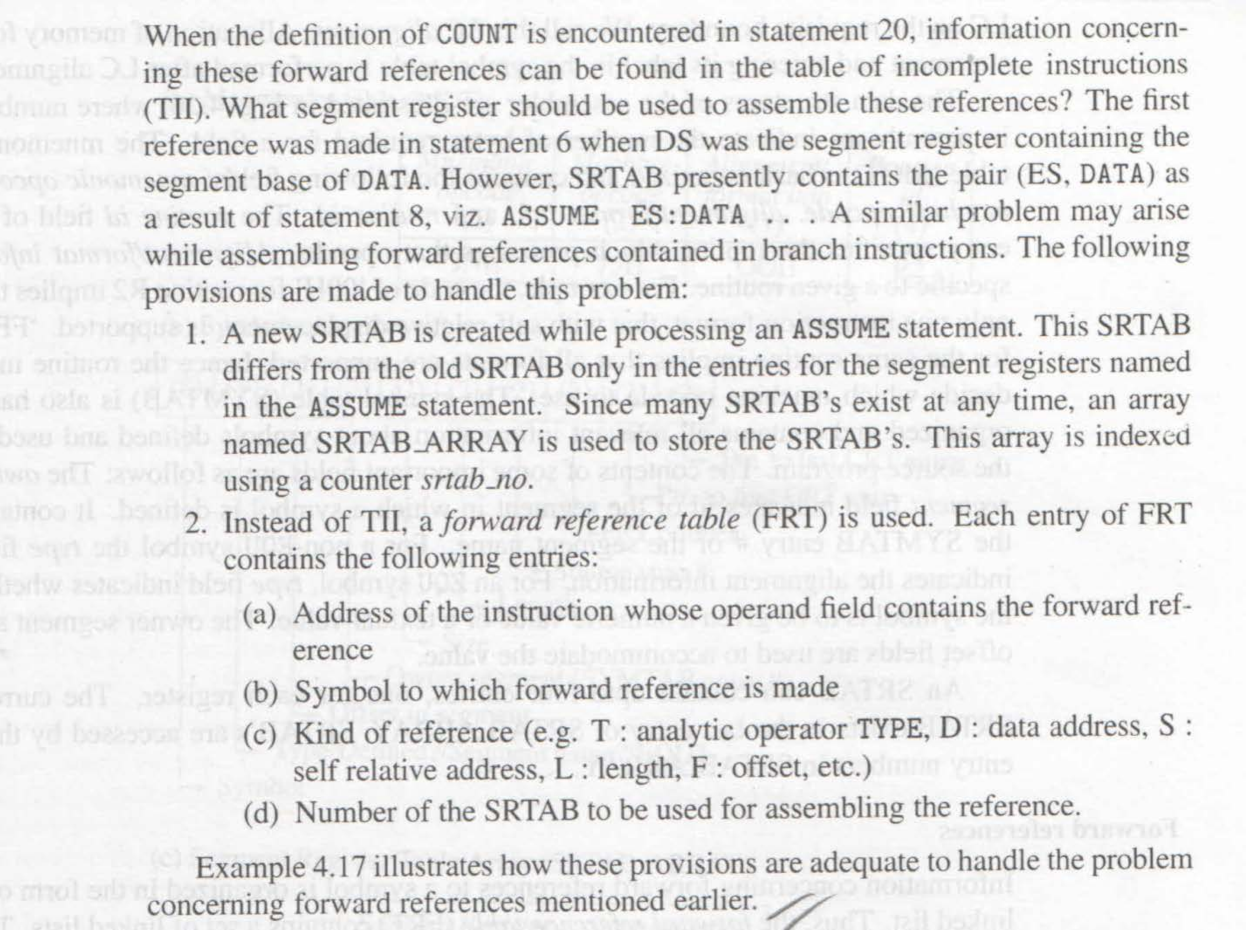
```
XYZ      DB       LENGTH ABC DUP(0)
– –
ABC      DD       ?
```

Here the forward reference to ABC makes it impossible to assemble the DB statement in a single pass.

Segment registers

An ASSUME statement indicates that a segment register contains the base address of a segment. The assembler represents this information by a pair of the form (*segment register, segment name*). This information can be stored in a *segment registers table* (SRTAB). SRTAB is updated on processing an ASSUME statement. For processing the reference to a symbol *symb* in an assembly statement, the assembler accesses the symbol table entry of *symb* and finds (seg_{symb}, $offset_{symb}$) where seg_{symb} is the name of the symbol containing the definition of *symb*. It uses the information in SRTAB to find the register which contains seg_{symb}. Let it be register *r*. It now synthesizes the pair (*r*, $offset_{symb}$). This pair is put in the address field of the target instruction.

However, this strategy would not work while assembling forward references. Consider statements 6 and 13 in Fig. 4.25 which make forward references to COUNT.



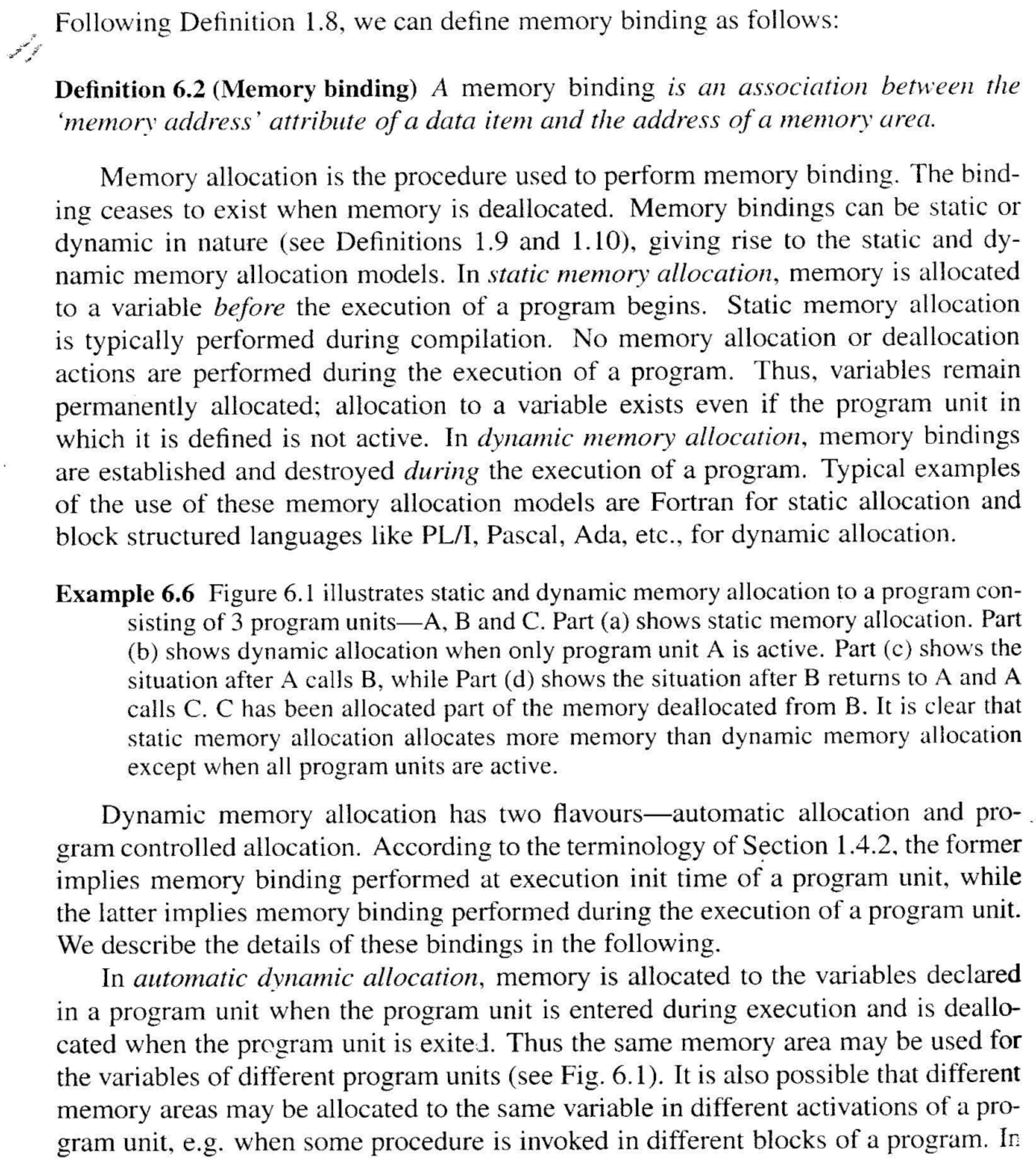
When the definition of COUNT is encountered in statement 20, information concerning these forward references can be found in the table of incomplete instructions (TII). What segment register should be used to assemble these references? The first reference was made in statement 6 when DS was the segment register containing the segment base of DATA. However, SRTAB presently contains the pair (ES, DATA) as a result of statement 8, viz. ASSUME ES:DATA . . A similar problem may arise while assembling forward references contained in branch instructions. The following provisions are made to handle this problem:

1. A new SRTAB is created while processing an ASSUME statement. This SRTAB differs from the old SRTAB only in the entries for the segment registers named in the ASSUME statement. Since many SRTAB's exist at any time, an array named SRTAB_ARRAY is used to store the SRTAB's. This array is indexed using a counter *srtab_no*.
2. Instead of TII, a *forward reference table* (FRT) is used. Each entry of FRT contains the following entries:
   (a) Address of the instruction whose operand field contains the forward reference
   (b) Symbol to which forward reference is made
   (c) Kind of reference (e.g. T : analytic operator TYPE, D : data address, S : self relative address, L : length, F : offset, etc.)
   (d) Number of the SRTAB to be used for assembling the reference.

Example 4.17 illustrates how these provisions are adequate to handle the problem concerning forward references mentioned earlier.

Q.9 a. Explain the role of static and dynamic memory allocation used in compilers. (5)

Answer:



Following Definition 1.8, we can define memory binding as follows:

Definition 6.2 (Memory binding) A memory binding *is an association between the 'memory address' attribute of a data item and the address of a memory area.*

Memory allocation is the procedure used to perform memory binding. The binding ceases to exist when memory is deallocated. Memory bindings can be static or dynamic in nature (see Definitions 1.9 and 1.10), giving rise to the static and dynamic memory allocation models. In *static memory allocation*, memory is allocated to a variable *before* the execution of a program begins. Static memory allocation is typically performed during compilation. No memory allocation or deallocation actions are performed during the execution of a program. Thus, variables remain permanently allocated; allocation to a variable exists even if the program unit in which it is defined is not active. In *dynamic memory allocation*, memory bindings are established and destroyed *during* the execution of a program. Typical examples of the use of these memory allocation models are Fortran for static allocation and block structured languages like PL/I, Pascal, Ada, etc., for dynamic allocation.

Example 6.6 Figure 6.1 illustrates static and dynamic memory allocation to a program consisting of 3 program units—A, B and C. Part (a) shows static memory allocation. Part (b) shows dynamic allocation when only program unit A is active. Part (c) shows the situation after A calls B, while Part (d) shows the situation after B returns to A and A calls C. C has been allocated part of the memory deallocated from B. It is clear that static memory allocation allocates more memory than dynamic memory allocation except when all program units are active.

Dynamic memory allocation has two flavours—automatic allocation and program controlled allocation. According to the terminology of Section 1.4.2, the former implies memory binding performed at execution init time of a program unit, while the latter implies memory binding performed during the execution of a program unit. We describe the details of these bindings in the following.

In *automatic dynamic allocation*, memory is allocated to the variables declared in a program unit when the program unit is entered during execution and is deallocated when the program unit is exited. Thus the same memory area may be used for the variables of different program units (see Fig. 6.1). It is also possible that different memory areas may be allocated to the same variable in different activations of a program unit, e.g. when some procedure is invoked in different blocks of a program. In

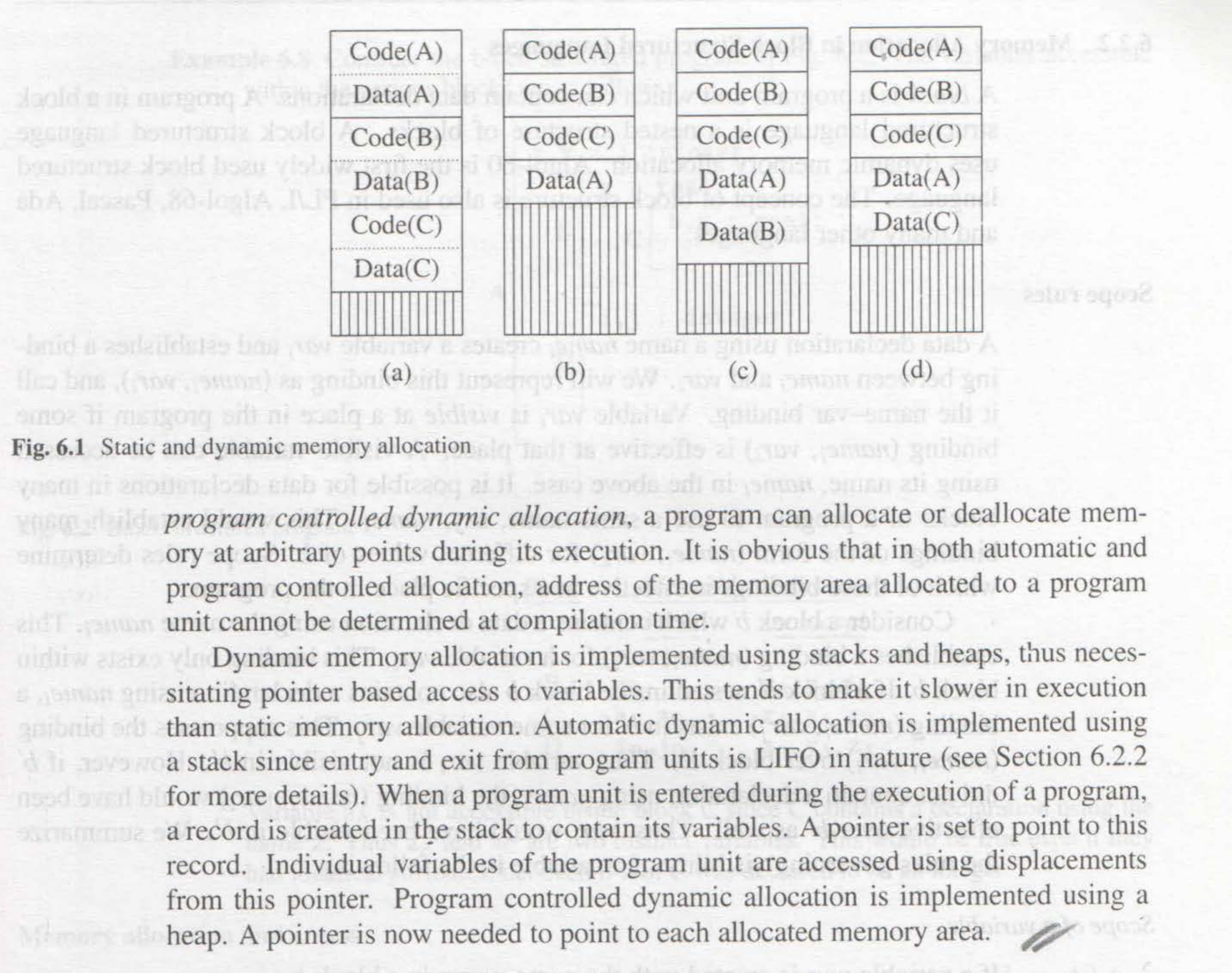

Code(A)
Data(A)
Code(B)
Data(B)
Code(C)
Data(C)
(a)
Code(A)
Code(B)
Code(C)
Data(A)
(b)
Code(A)
Code(B)
Code(C)
Data(A)
Data(B)
(c)
Code(A)
Code(B)
Code(C)
Data(A)
Data(C)
(d)

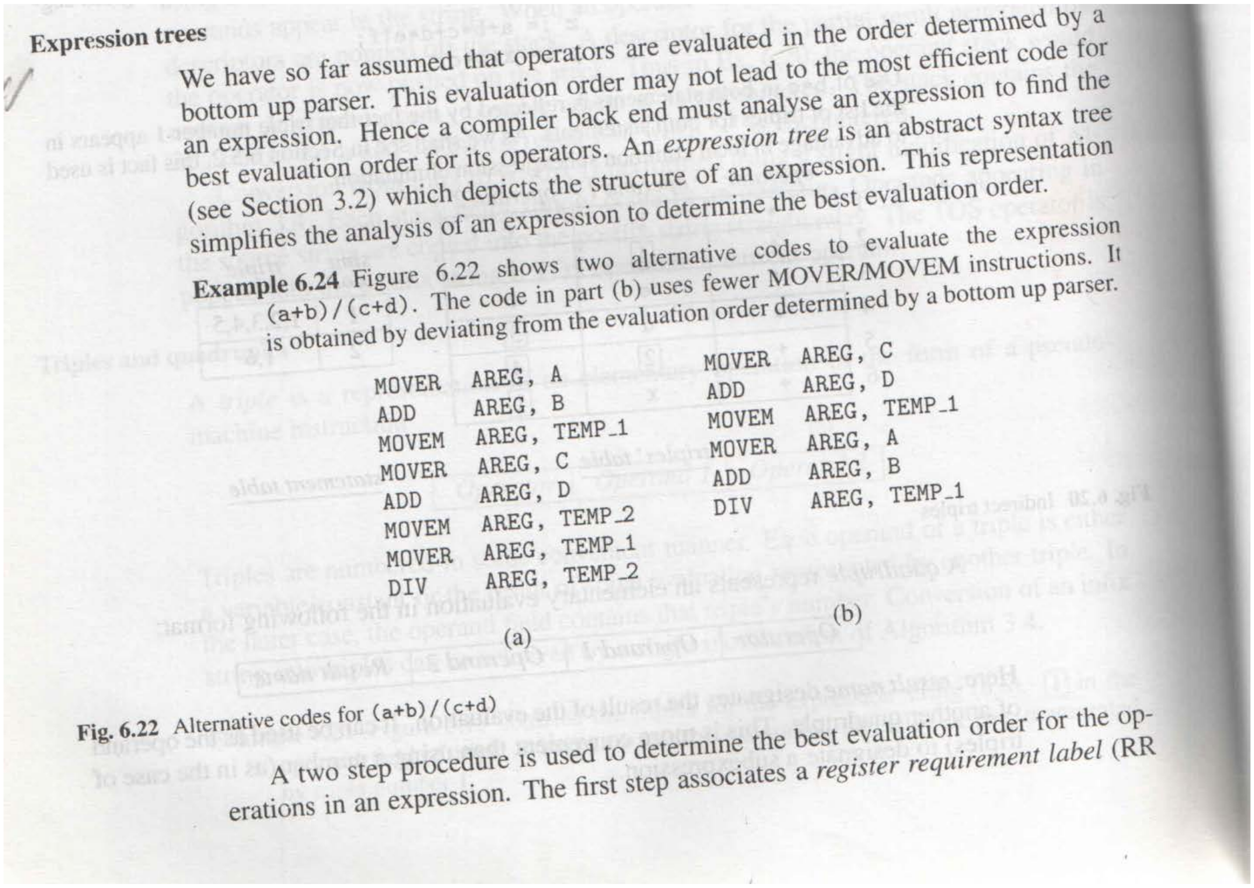
Fig. 6.1 Static and dynamic memory allocation

program controlled dynamic allocation, a program can allocate or deallocate memory at arbitrary points during its execution. It is obvious that in both automatic and program controlled allocation, address of the memory area allocated to a program unit cannot be determined at compilation time.

Dynamic memory allocation is implemented using stacks and heaps, thus necessitating pointer based access to variables. This tends to make it slower in execution than static memory allocation. Automatic dynamic allocation is implemented using a stack since entry and exit from program units is LIFO in nature (see Section 6.2.2 for more details). When a program unit is entered during the execution of a program, a record is created in the stack to contain its variables. A pointer is set to point to this record. Individual variables of the program unit are accessed using displacements from this pointer. Program controlled dynamic allocation is implemented using a heap. A pointer is now needed to point to each allocated memory area.

b. Define expression trees and give their applications. (5)

Answer:



Expression trees

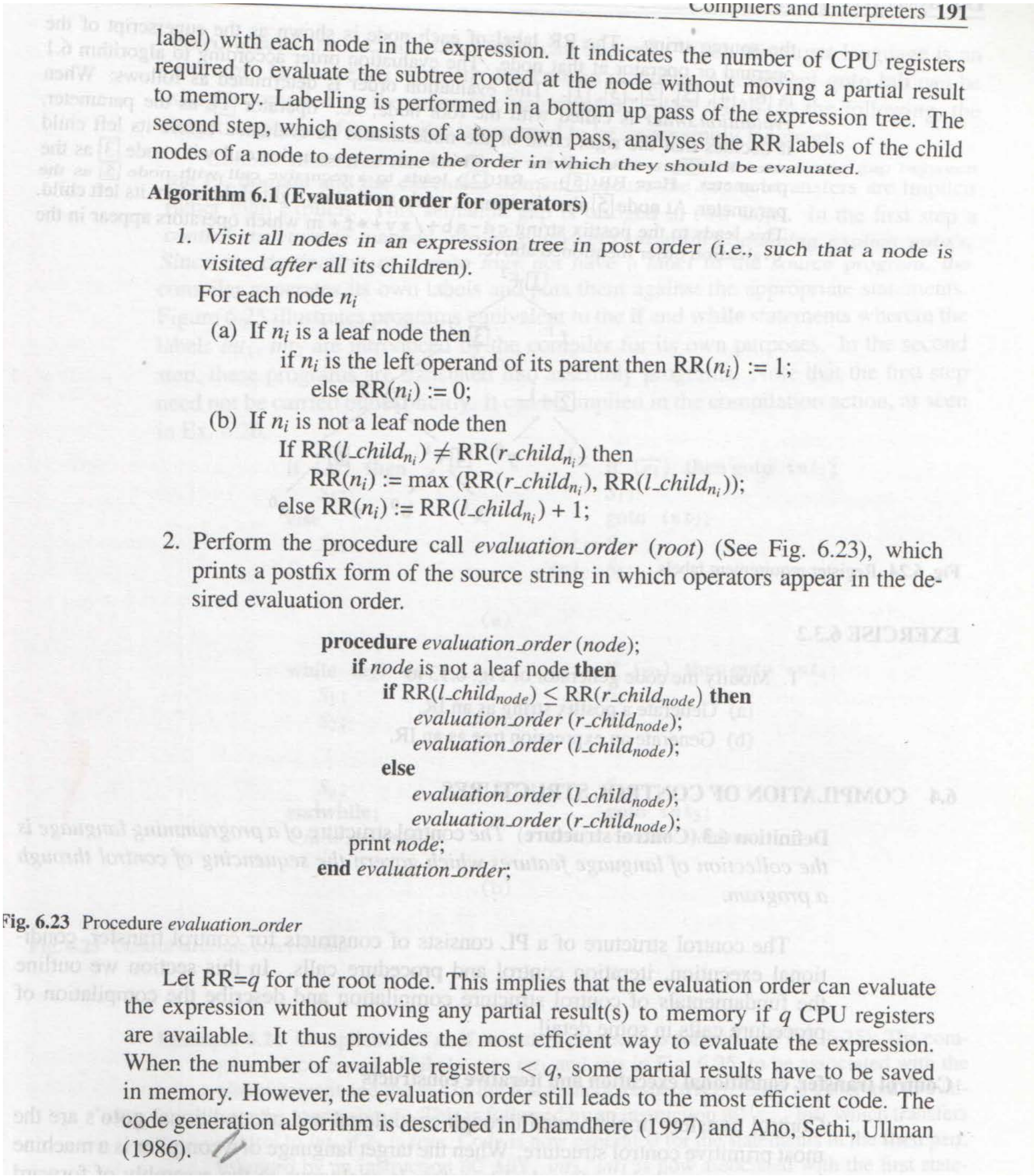
We have so far assumed that operators are evaluated in the order determined by a bottom up parser. This evaluation order may not lead to the most efficient code for an expression. Hence a compiler back end must analyse an expression to find the best evaluation order for its operators. An *expression tree* is an abstract syntax tree (see Section 3.2) which depicts the structure of an expression. This representation simplifies the analysis of an expression to determine the best evaluation order.

Example 6.24 Figure 6.22 shows two alternative codes to evaluate the expression (a+b)/(c+d). The code in part (b) uses fewer MOVER/MOVEM instructions. It is obtained by deviating from the evaluation order determined by a bottom up parser.

```
MOVER  AREG, A          MOVER  AREG, C
ADD    AREG, B          ADD    AREG, D
MOVEM  AREG, TEMP_1     MOVEM  AREG, TEMP_1
MOVER  AREG, C          MOVER  AREG, A
ADD    AREG, D          ADD    AREG, B
MOVEM  AREG, TEMP_2     DIV    AREG, TEMP_1
MOVER  AREG, TEMP_1
DIV    AREG, TEMP_2
                             (b)
          (a)
```

Fig. 6.22 Alternative codes for (a+b)/(c+d)

A two step procedure is used to determine the best evaluation order for the operations in an expression. The first step associates a *register requirement label* (RR



label) with each node in the expression. It indicates the number of CPU registers required to evaluate the subtree rooted at the node without moving a partial result to memory. Labelling is performed in a bottom up pass of the expression tree. The second step, which consists of a top down pass, analyses the RR labels of the child nodes of a node to *determine the order in which they should be evaluated.*

Algorithm 6.1 (Evaluation order for operators)

```
1. Visit all nodes in an expression tree in post order (i.e., such that a node is
   visited after all its children).
   For each node $n_i$
     (a) If $n_i$ is a leaf node then
           if $n_i$ is the left operand of its parent then $RR(n_i) := 1$;
             else $RR(n_i) := 0$;
     (b) If $n_i$ is not a leaf node then
           If $RR(l\_child_{n_i}) \neq RR(r\_child_{n_i})$ then
             $RR(n_i) := \max(RR(r\_child_{n_i}), RR(l\_child_{n_i}))$;
           else $RR(n_i) := RR(l\_child_{n_i}) + 1$;
2. Perform the procedure call evaluation_order (root) (See Fig. 6.23), which
   prints a postfix form of the source string in which operators appear in the de-
   sired evaluation order.
```

```
procedure evaluation_order (node);
  if node is not a leaf node then
    if $RR(l\_child_{node}) \le RR(r\_child_{node})$ then
      evaluation_order ($r\_child_{node}$);
      evaluation_order ($l\_child_{node}$);
    else
      evaluation_order ($l\_child_{node}$);
      evaluation_order ($r\_child_{node}$);
  print node;
end evaluation_order;
```

Fig. 6.23 Procedure *evaluation_order*

Let RR=*q* for the root node. This implies that the evaluation order can evaluate the expression without moving any partial result(s) to memory if *q* CPU registers are available. It thus provides the most efficient way to evaluate the expression. When the number of available registers < *q*, some partial results have to be saved in memory. However, the evaluation order still leads to the most efficient code. The code generation algorithm is described in Dhamdhere (1997) and Aho, Sethi, Ullman (1986).

c. Explain pure and impure interpreters. Give an illustration. (3+3)

Answer:

6.6.3 Pure and Impure Interpreters

The schematic of Fig. 6.34(a) is called a *pure* interpreter. The source program is retained in the source form all through its interpretation. This arrangement incurs substantial analysis overheads while interpreting a statement.

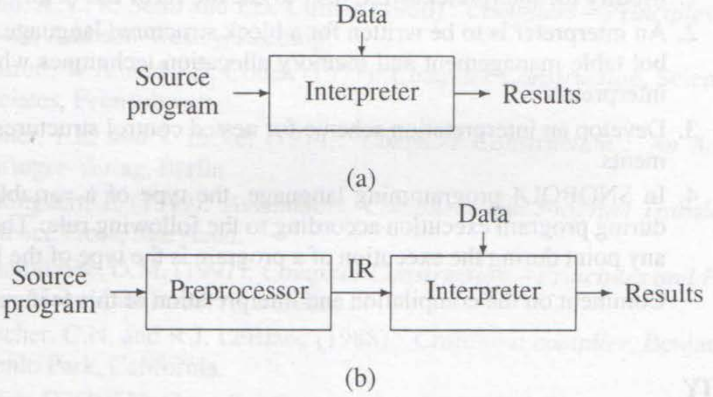


Fig. 6.34 Pure and impure interpreter

An *impure interpreter* performs some preliminary processing of the source program to reduce the analysis overheads during interpretation. Figure 6.34(b) contains a schematic of impure interpretation. The preprocessor converts the program to an intermediate representation (IR) which is used during interpretation. This speeds up interpretation as the code component of the IR, i.e. the IC, can be analysed more efficiently than the source form of the program. However, use of IR also implies that the entire program has to be preprocessed after any modification. This involves fixed overheads at the start of interpretation.

Example 6.42 Postfix notation is a popular intermediate code for interpreters. The intermediate code for a source string a+b*c could look like the following:

S #17	S #4	S #29	*	+

where each IC unit resembles a token (see Section 1.3.1.1).

IC of Ex. 6.42 eliminates most of the analysis during interpretation excepting type analysis to determine the need for type conversion. Even this can be eliminated if the preprocessor performs type analysis before generating IC.

Example 6.43 The preprocessor of an interpreter performs type analysis to generate following IC for the expression a+b*c, where a, b are of type real and c is of type integer

S #17	S #4	S #29	$t_{i\rightarrow r}$	$*_r$	$+_r$

where the unary operator $t_{i\rightarrow r}$ indicates type conversion of an operand (in this case, c) from 'integer' to 'real'. The arithmetic operators are also type specific now.Thus '$*_r$' indicates multiplication in the 'real' representation. This eliminates most analysis during interpretation.

TEXT-BOOK

I. Systems Programming and Operating Systems, D. M. Dhamdhere, Tata McGraw-Hill, Second Revised Edition, 2005