System Software

Unit I:

Introduction- System Software and Machine Architecture-Simplified Instructional Computer (SIC)-SIC Machine Architecture-SIC/XE Machine Architecture-Traditional (CISC) Machines-VAX Architecture-Pentium Pro Architecture-RISC Machines-UltraSPARC Architecture-PowerPC Architecture-Cray T3E Architecture

Unit II:

Assemblers-Basic Assembler Functions-A Simple SIC Assembler-Assembler Algorithm and Data Structures-One pass Assemblers-Multi-pass Assemblers.

Unit III:

Loaders & Linkers: Basic Loader Functions-Design of Absolute Loader-Simple Bootstrap Loader-Machine Dependent Loader Features.

Unit IV:

Compilers-Basic Compiler Functions-Grammars-Lexical Analysis-Syntactic Analysis-Code Generation

Unit V:

Other System Software: Text Editors-Interactive Debugging Systems.

Text Book:

Leland L.Beck,"System Software-An Introduction to Systems Programming",3rd Edition, Pearson Education Asia, 2000

Unit I:	**Chapter 1**
Unit II:	**Chapter 2(2.1 & 2.4)**
Unit III:	**Chapter 3(3.1 & 3.2)**
Unit IV:	**Chapter 5(5.1)**
Unit V:	**Chapter 7(7.1 & 7.2)**

References:

1.D.M. Dhamdhere,"Systems Programming and Operating System",Second Revised Edition,Tata McGraw-Hill, 1999
2.John J.Donovan "Systems Programming",Tata McGraw-Hill Edition,1972.

UNIT I

Introduction:

System Software consists of programs that supports the operation of a computer. This software makes its possible for the user to focus on an application without needing to know the detail of how the machine works internally.

Compiler:

Compiler is system software it converts higher level language into Machine language.

Assembler:

Assembler is a system software it converts low level language into Machine language.

Loaded or linker:

The machine language instructions are loaded into the memory for execution loaded or linker.

System software and Machine architecture

1. Most system software difference from the application software is one characteristic called Machine dependency.
2. An application program is primarily consened with the solution of some problems using the computer as a tool.
3. The focuses on the application not on the computing system.
4. System programs are intendened to supports the operation and use of the computer itself rather than any particular application.
5. They are usually related to the architecture of the machine on which they are to run.
6. To avoid this problem the fundamental functions of each piece of software are presented through SIC (Simplified Instructional Computer).
7. SIC is the hypothetical computer that has been designed to include the hardware features most of an found on real machine.

The Simplified instructional computer:

They are two versions

1.SIC - Stanadard Model

2.SIC/XE – Extra equipment or extra expensive.

The two versions are designed to be upward compatible. An object program for the standard SIC machine will also executed properly on a SIC/XE machine.

SIC Machine Architecture

Memory:

1. Memory consists of 8 bit bytes.
2. Three consecutive bytes form a word.
3. All addresses on SIC are byte addresses.
4. These are a total of 32,768(2 the power 15 bytes).

Register:

There are 5 Register, each register is 24 bites in length.

Mnemonic	Number	Special use
A	0	Accumulator, used for arithmetic operation.
X	1	Index register, used for addressing.
L	2	Linkage register, the jump to Subroutine (JSUB) instruction stores the return Address in this register.
PC	8	Program Counter, contains the address of The next instruction to be fetched for execution.
SW	9	Status word, contains a variety of information Including a Condition Code (CC).

Data Formats:

1. Integer are stored as 24 bit binary number.
2. 2's Complement representations is used for negative values.
3. Characters are stored as 8 bit ASCII Code.
4. There is a no floating point hardware.

Instruction Formats:

All machine instruction have 24 bit formats.

8	1	15
Opcode	X	Address

The flag bit X is used to indicate indexed addressing mode.

Addressing Mode:

1. There are 2 types of addressing mode.
2. It indicate by the setting of the x bit in the instruction.

Mode	Indication	Target address calculation
Direct	X=0	TA= address
Indexed	X=1	TA=address+(x)

Direct Addressing mode:

Example:

LDA TEN

8	1	15 bit
0000 0000	0	001 0000 0000 0000
0 0		1 0 0 0
Opcode	x	TEN

Effective address (EA)=100

Content of the address 1000 is loaded to accumulator.

Indexed Addressing Mode:

Example:

STCH BUFFER,X

8 bit	1	15 bit
0101 0100	1	001 0000 0000 0000
5 4		1 0 0 0
Opcode	x	BUFFER

Accumulator contains the content of a calculated address.

Instruction Set:

1. SIC provides a basic set of instruction.

2. Instruction to load and store register (LDA,LDX,STA,STX).
3. It includes integer arithmetic operation (ADD,SUB,MUL,DIV).
4. There is an instruction COMP it compares the value in register A with a word in memory.
5. This instruction sets a Condition Code(CC) to indicate the result(>,=,<).
6. Conditional jumps instructional are used JLT,JET,JGT are used.
7. JSUB jumps to the subroutine placing the return address in Register "L".
8. RSUB returns by jumping to the address contained in register L.

Input and Output:

1 Input and output are performed by transferring one byte at a time.
2. Each device is assigned a unique eight bite code.
3. There are 3 IO instructions that uses device code as an operand.

1. Text device(TD)
   Instruction test whether the addressed device is ready to send or receive a byte of data.
   This can be done by setting a Condition Code(CC).
   < means the device is ready to send or receive.

= means the device is not ready.
A program needing to transfer data must wait until the device is ready, then execute.
2. Read Data(RD) or
3.Write Data(WD).

SIC/XE Machine Architecture

Memory:

1. Memory consists of 8 bit.
2. Three consecutive byte form a word.
3. All address on SIC/XE are byte address.
4. The maximum memory available on SIC/XE is one mega bite (2 the power 20) byte.

Register:

Same register of SIC is also available in SIC/XE and additional 4 registers.

Mnemonic	Number	Special use
B	3	Base register, used for addressing.

S	4	General working register- no special use
T	5	General working register-no special use
F	6	Floating-point accumulator (48 bits)

Data Formats:

1. Integers are stored as 24 bit binary number.
2. 2's Complement representation is used for negative value.
3. Characters are stored as ASCII code.
4. There is a 48 floating point data types it will be following format.

1	11	36
S	Exponent	fraction

S=0 – positive
S=1 - negative

Instruction formats:

Format 1 and format 2- that do not reference memory.
Format 3 and format 4- that reference memory.

Format 1: (1 byte)

Opcode – 8 bit

Eg: RSUB (retain to subroutine)

```
        Opcode
     0100    1100
       4       C
```

Format 2: (2 byte)

8	4	4
opcode	r 1	r 2

Eg: COMPR A, S (Compare the contents of register A,S)

Opcode	A	S
1010 0000	0000	0100
8 bit	4 bit	4 bit
A 0	0	4

Format 3: (3 byte)

6	1	1	1	1	1	1	12
op	n	i	x	b	p	e	Dis

Eg: LDA #3

Format 4: (4 byte)

6	1	1	1	1	1	1	20
op	n	i	x	b	p	e	add

Addressing mode:

1. Base relative
2. Program Counter relative

Base relative	b=1, p=0	TA=(B)+ disp
Program Counter	b=0, p=1	TA=(PC)+disp

Base relative:
Eg: STX LENGTH

6 op	n	i	x	b	p	e	12
0001 00	1	1	0	1	0	0	0000 0000 0000
1 3		4				0	0 0

The content of the address 0033 is loaded into indexed register x.

TA = (B) + disp
= 0033 + 0
=0033

Program Counter relative:

Eg: STL RETADR

6 Op	n	i	x	b	p	e	12
0100 01	1	1	1	0	1	1	0000 0010 1101
1 7			2				0 2 D

Format 3:
b=0 p=0 TA = Disp

Format 4:
B=0 p=0 TA = Disp

If format 3 and 4 the bits i, and n is used to specify how target address is used.
Target address is used as operand i=1,n=0, Immediate addressing mode.
Eg: LDA #9 – Directly store 9 in address
i=0,n=1 - Indirect addressing memory location(or) register.
i=0,n=0 – simply addressing.

Instruction set:

1. There are instruction to load and store the new register (LDB,STB)
2. Floating point arithmetic operation instruction are also available.
   ADDF,SUBF,MULF,DIVF.
3. Register to register arithmetic operation are also available.
   ADDR,SUBR,MULR,DIVR
4. supervisor cal instruction are also available.
5. This instruction used to generate an interrupt that can be used for communicating with OS.

Input and Output:

1. There are IO channels that can be used to perform input and output while the CPU is executing other instructions.

2. The instructions SIO , PIO , HIO are used to Start, Test and Halt the operation of IO channels

Traditional CISC Machine

VAX Architecture:

It was introduced by DEC (Digital Equipment Corporation) in 1978. The VAX programs to share the same machine in a multi user environment.

Memory:

1. It consists of 8 bit byte
2. 2 consecutive byte form a word.
3. 4 byte form a long word.
4. 8 byte form a quadword.
5. 16 byte form a octaword.
6. All the VAX program operate in a Virtual address.
7. Routine in the os take care of the details of memory management.
8. one half of the VAX virtual address space is called "System space"

System Space:

1. It contains the OS and it is shared by all the program.

Process Space:

1. The other half of the address space is called "process space", and it defined separately for each program.

Register:

1. There are 16 general register designated to R0 through R15.
2. Each register is 32 bites in length.

R15 = Program Counter (PC)
It contains the next instruction address to be fetched (or) executed.
R14 = Stack Pointer (SP)
It points to the current top of the Stack in the program process space.
R13 = Front Pointer (FP)
The VAX procedure called convensions built a data structure called a Stack Front and its address in FP.
R12 = Argument Pointer (AP)
It is used to pass a list of arguments associated with each procedure called Argument Pointer.
R6 through R11 = Have no special functions and
R0 through R15 = For general use.

Processor Status Long word (PSL)
It contains state variable and flag associated with the processes.
There are also number of Control registers to support varies OS functions.

Data formats:

1. Integers are stored as binary numbers in a byte,word,longword,quadword or octaword.
2. Negative values are represented by 2's complement.
3. Characters are stored as 8 bit ASCII Code.

4. Floating point data formats are available, 4 different floating point are available ranging from 4-16 bytes.
5. Two of these are compactative and standard on all VAX processor.
6. The other 2 are available for options and provide an extended range of value in exponent field.
7. VAX processor provide a

(1)Packed decimal data format:
Each byte represent 2 decimal digit (each contain 4 bit for digit).The sign is encoded in the last 4 bites.

(2) Numeric format:
1 digit by byte. The sign may appears either in the last byte or as a separate byte preceding the first digit.

These variations are called Trailing Numeric and Leading Separate Numeric.

Instruction formats:

1 VAX machine instruction use the various length instruction formats.
2. Each instruction consists of an operation code (opcode) 1 or 2 bytes.
followed by upto 6 operand specifiers depending on the type of instruction.
3. The operand specifiers denote the type of addressing mode used.

Addressing mode:

1 There are large number of addressing mode.
1. Register mode = The operand itself may be in a register or its address may be specified by a register (register deferred mode)
2. If the operand address is in a register, the register contents may be automatically incremented or decremented by the operand length (auto increment, and auto decrement modes).

3. All of these addressing modes may also include an index register, and many of them are available in a form that specifiers indirect addressing called deferred mode on VAX.
4. In addition there are immediate operands and several special purpose addressing modes

Instruction set:

One of the goals of the VAX designers was to produce an instructions set that is symmetric with respect to data type.

Many instructions Mnemonics are formed by combining the following elements.

1. A prefix that specifiers the type of operation.
2. A suffix that specifiers the data type of the operands.
3. A modifiers that gives the number of operands involved.

Input and Output:

1. Input and output on the VAX are accomplished by I/O device controllers.
2. Each controller has a set of Control /Status and data registers which are assigned location in the physical address space.
3. The portion of the address space into which the device controller registers are mapped is called I/O space.

Pentium Pro Architecture

Memory:

1. Memory consists of 8 bit bytes
2. All addresses used are byte address.
3. Two consecutive byte form a word.
4. Four byte form a double-word.(dword)

Register:

1. These are eight general-purpose register,which are named EAX,EBX,ECX,EDX,ESI,EDI,EBP and ESP.
2. Each general purpose register is 32 bits long(One double word)
3. Register EAX,EBX,ECX, and EDX are generally used for data manipulation.
4. It is possible to access individual words or byte these registers.

5. Floating point computations are performed using a special floating point unit (FPU). This unit contains eight 80 bit data registers and several other control and status registers.

Data formats:

1. Integers are normally stored as 8,16 or 32 bits binary numbers.
2. 2’s complement is used for negative values.
3. Integers can also be stored in binary coded decimal(BCD)
4. There are three different floating point data formats. The single-precision format is 32 bit long.

Instruction formats:

1. This format begins with optional containing flags that modify the operation of the instructions.
2. The opcode is the only elements that is always present in every instructions.

3. Other elements may or may not be present, and may be of different lengths,depending on the operations and the operands involved.
4. In length form 1 byte to 10 byte or more.

Addressing mode:

1. An operand value may be specified as part of the instruction itself (immediate mode), or it may be in a register (register mode).
2. Operands stored in memory are often specified using variations of the general target address calculations.

   TA = (base register) + (index register) * (scale factor) + displacement
3. Various combinations of these items may be omitted resulting in eight addressing mode.
4. The address of an operand in memory may also be specified as an absolute location (direct mod) or as a location relative to the EIP register (relative mode).

Instruction set:

1. An instruction may have zero, one,two,or three operands.
2. There are registers-to-memory instructions, and a few memory-to-memory instructions.

3. In some cases, operands may also be specified in the instructions as immediate values.
4. Most data movements and integers arithmetic instructions can use operands that are 1,2, or 4 byte.
5. These are many instructions that perform logical and bit manipulations and support control of the processor and memory- management systems.

Input and Output:

1. Input is performed by instructions that transfer one byte,word of doubleword at a time form an I/O part into register EAX.
2. Output instructions transfer one byte word,or double word from EAX to an I/O part

RISC MACHINE

It consists of 3 Machine

1. UltraSPARC Architeture
2. PowerPC Architecture
3. CrayT3E Architecture

UltraSPARC Architeture:

1. Announced by Sun Microsystems in 1995.
2. Others members of family SPARC and superSPARC
3. SPARC – Scalable Processor Architeture.
4. SPARC,UltraSPARC,and superSPARC are upward compatible.

Memory:

1. 8-bit bytes
2. 2 bytes form Halfword
3. 4 bytes form Word.
4. 8 bytes form Double Word.
5. Programs written using Virtual Address Space of 2 power 24 bytes.
6. Address space is divided into pages.
7. some pages are stored in the physical memory and others on disk.
8. when an instruction is executed, the hardward and OS should make sure that that the needed page is loaded into the memory.
9. Virtual address is translated into physical memory by “MMU”.

Register:

1. Larger register file (more than 100 general- purpose register.
2. Any procedure can access only 32 registers (r0 through r31)

3. First 8 register (r0 through r7) are global register accessed by all procedures.
4. Other 24 registers available to a procedure can be Visualized as a “Window” through which part of the register file can be seen.
5. If a set of concurrently running procedures needs more window than are physically available, a ‘window overflow’ interrupt occurs.
6. General purpose registers 24 bits long.
7. Floating point computations are performed using Floating Point Unit (FPU).
8. PC, Condition Code registers and a no.of other control registers are also available.

Data Formats:

1. Provides storage of integers, floating point values and characters.
2. Integers 8,16,32,64 bit binary numbers.
3. Signed and unsigned supported.
4. Negative values 2’s Complement
5. Supports big-endian, little-endian. Most significant part of numeric value is stored at the lowest numbered address.
6. Floating point format (3 types)

	Length	Floating point Value	Exponent	Sign
Single precision	32 bits	23 bits	8 bits	1 bit
Doubleprecision	64 bits	52 bits	11 bits	1 bit
Quad precision		63 bits	15 bits	1 bit

7. Characters 8 bit ASCII Code.

Instruction Formats:

1. 3 basic formats each 32 bits long.
2. First 2 bit identifies the type of format
3. Format 1 used for call instruction.
4. Format 2 used for branch instruction.
5. Format 3 used for register loads and stores, and 3 operand Arithmetic Operations.

Addressing Mode:

1. Immediate mode (operand value directly in instruction).
2. Register - Direct mode
   PC – relative TA= (PC) + disp
3. Register- Indirect with disp
   TA= (Reg) + disp
4. Register indirect indexed TA= (reg-1) + (reg-2)

Instruction Set:

1. 100 machine instruction.
2. Only instruction [load and store] access memory.
3. Other instruction are register-to-register operations
4. Instruction execution is pipelined [while one instruction is executed, the next one fetched and decoded.
5. This technique speeds execution
6. It also includes special purpose instructions to support os and compiler.

Input and Output:

1. Communication with I/O device is done through memory.
2. Large of memory locations is replaced by device register.
3. Each I/O device has unique address.
4. When load and store instruction refers this area, the corresponding device is activated.
5. Thus i/p & o/p can be performed with instruction.

POWER PC ARCHITETURE

1. Introduced by IBM in 1990.
2. POWER [Performance Optimization With Enhanced RISC]
3. IBM,APPLE,Motorola combined to develop and market such up named Power PC.

Memory:

1. 8 bit bytes.
2. 2 bytes form ½ Word .
3. 4 bytes form Word.
4. 8 bytes from double word.
5. 16 bytes form quad word.
6. Programs written using Virtual Address Space of 2 power 64 bytes.
7. Address space is divided into fixed- length segments (256 MB)
8. Segments is divided into "Pages" (4096 bytes)
9. MMU (same point as in UltraSPARC).

Registers:

1. 32 general purpose registers GPR0 – GPR31
2. Each registers is 64 bits long.
3. FPU (Floating Point Unit) for floating point computations,
4. This unit contains 64 bit floating point register and status and control registers.
5. 32 bit condition register are used for testing and branching.
6. This register is divided into 8 subfields (4 bit).
7. name CRO – CR7

8. Link Register (LR) and Count Register (CR) used for branch instruction.
9. Machine Status Register (MSR) available.

Data Formats:

1. Provides storage of integers, floating point values and characters.
2. Instruction 8.16,32,64 bit binary number.
3. Both Sign and Unsigned allowed
4. Negative values 2'Complement
5. big-endian byte Ordering is used
6. 2 Floating point formats.

	Length	Floating point Value	Exponent	Sign
Single-precision	32 bits	23 bits	8 bits	1 bit
Double precision	64 bits	52 bits	11 bits	1 bit

7. Characters 8 bit ASCII Code.

Instruction Formats:

1. 7 basic formats and has subforms
2. All formats 32 bits long
3. First 6 bits specify opcode.

Addressing Modes:

1. Immediate mode.
2. Registers direct mode.
3. Only instruction (load and store) refer memory.
4. Register indirect TA = (reg)
5. Register indirect with index   TA = (reg-1)+(reg-2)
6. Register indirect
   with immediate mode   TA = (reg) + disp.
7. Branch instruction use the following address mode.

Absolute	TA = actual address
Relative	TA = current instruction address + disp
Link register	TA = (LR)
Count register	TA = (CR)

Instruction Set:

1. 200 machine instruction.
2. Some instruction more complex

Eg: Floating point multiply & address” instruction take 3 l/p operands and performs multiply and addition in one instruction.

3. Instruction execution in pipelined.

Input and Output:

1. 2 different method for performing I/O operations
2. Segment in Virtual address space are mapped on to an external address space (I/O bus)
3. This way is called “direct-store” segments.

Cray T3E Architeture

Memory:

1. It consists of 8 bit bytes
2. 2 consecutive bytes form a Word.
3. 4 bytes form a long word.
4. 8 bytes form a quad word.
5. It supports 64 bit Virtual addresses.

Register:

1. 32 general purpose registers designated R0- R31.
2. R31 always contains the value zero.
3. There are also 32 Floating point registers designed F0 through F31.
4. F31 always contains the value zero.
5. There is a 64 bit Program Counter (PC).

Data Formats:

1. Integers are stored as longword or quadword.
2. 2’s Complement is used for Negative values.
3. There are two different types of floating point.
4. Characters may be stored one per byte using their 8 bit ASCII Code.

Instruction Formats:

1. There are 5 basic instruction formats.
2. All of these formats are 32 bits long
3. The first 6 bit of the instruction word always specify the opcode.

Addressing Mode:

An operand values may be specified as part of the instruction itself (immediate mode) or it may be in a register (register direct mode).

Mode	Target address calculation
PC – relative	TA = (PC) + displacement {23 bits,signed}
Register indirect with	TA = (register) + dis {16 bits, signed}

Instruction Set:

The instruction set is designed so that an implementation of the architecture can be as fast as possible
Ex: There are no byte or word load and store instructions.

Input and Output:

1. The T3E System performs I/O through multiple ports into one or more I/O channels.
2. These channels are integrated into the networks that inter connects the processing mode.
3. All channels are accessible and controllable from all PES.

UNIT II

ASSEMBLERS

Fundamental functions translating mnemonic operation codes to their machine language equivalents and assigning machine addresses to symbolic labels used by the programmer.
BASIC ASSEMBLER FUNCTIONS
The line numbers are for reference only and are not part of the program.
These numbers also help to relate corresponding parts to different versions of the program.
Indexed addressing is indicated by adding the modifier ",X" following the operand.
Lines beginning with "." Contain comments only.
ASSEMBLER DIRECTIVES

START	: Specify name and starting address for the program.
END	: Indicate the end of the source program and specify the first executable instruction in the program.

BYTE : Generate character or hexadecimal constant, occupying as many bytes as needed to represent the constant.

RESB : Reserve the indicated number of bytes for a data area.

RESW : Reserve the indicated number of words for a data area.

The program contains a main routine that reads records from an input device and copies them to an output device.

This main routine calls subroutine RDREC to read a record into a buffer and subroutine WRREC to write the record from the buffer to the output device.

Each subroutine must transfer the record one character at a time because the only I/O instructions available are RD and WD.

A record is longer than the length of the buffer, only the first 4096 bytes are copied.

The end of the file to be copied to is indicated by a zero –length record.

When the end of the file is detected, the program writes EOF on the output device and terminates by executing an RSUB instruction.

This program was called by the operating system using a JSUB instruction.

```
LINE                    SOURCE STATEMENT
5       COPY     START    1000        COPY  FILE FROM INPUT TO
OUTPUT
10      FIRST    STL      RETADR      SAVE  RETURN ADDRESS
15      CLOOP    JSUB     RDREC       READ THE INPUT RECORD
20               LDA      LENGTH      TEST FOR THE EOF
25               COMP     ZERO
30               JEQ      ENDFIL      EXIT IF EOF FOUND
35               JSUB     WRREC       WRITE THE OUTPUT RECORD
115     BUFFER         SUBROUTINE TO READ RECORD INTO BUFFER
120     BUFFER



125     RDREC    LDX      ZERO        CLEAR LOOP COUNTER
200     MAXLEN   SUBROUINE TO WRITE THE RECORD FROM BUFFER
205     MAXLEN
210     WRREC    LDX      ZERO        CLEAR THE LOOP COUNTER
```

A SIMPLE SIC ASSEMBLER

The column headed loc gives the machine address for each part of the assembled program.

It is assumed that the program starts at address 1000.

Translation of source program to object codes required us to accomplish the following functions:

1. Convent mnemonic operation codes to their machine language equivalents e.g., translate STL to 14.
2. Convert symbolic operands to their equivalents machine addresses e.g., translate RETADR to 1033.
3. Build the machine instructions in the proper format.

4. Convert the data constant specified in the source program into their internal machine representations-e.g., translate EOF to 454F46.
5. Write the object program and the assembly listing.

Consider the statement

```
10       1000    FIRST     STL       RETADR         141033
```

```
LINE      LOC        SOURCE STATEMENT      OBJECT CODE
5               1000     COPY        START       1000
10              1000     FIRST       STL           RETADR           141033
15             1003     CLOOP     JSUB         RDREC              482039
20             1006                    LDA          LENGTH           001036
25             1009                    COMP       ZERO               281030
30             100C                    JEQ          ENDFIL             301015
35             100F                    JSUB         WRREC             482061
115                          BUFFER    SUBROUTINE TO READ THE RECORD INTO
BUFFER
120                          BUFFER
125           2039     RDREC      LDX            ZERO                041030
200                          MAXLEN   SUBROUTINE TO WRITE RECORD FROM THE
BUFFER
205                          MAXLEN
210           2061     WRREC     LDX            ZERO                041030
```

We attempt to translate the program line by line, we will be unable to process this statement because we do not know the address that will be assigned to RETADR.
In addition to translating the instructions of the source program, the assembler must process statements called assembler directives.
Assembler directives in our simple sample program are START, which specifies the starting memory address for the object program, and END, which marks the end of the program.
This object program will later be loaded into the memory for execution.
The simple object program format we use contains three types of records: Header, Text, End.
The Header contains the program name, starting address and length.
Text record contains the translated instructions and data of the program, together with an indication of the addresses where these are loaded.
The End record marks the end of the object program.

Header record:

Col. 1	H
Col. 2-7	program name
Col. 8-13	starting address of object program

Col. 14-19 length of object program in bytes
Text record:
Col. 1 T
Col. 2-7 starting address for object code in this record
Col. 8-9 Length of object code in this record in bytes
C0l. 10-69 object code, represented in hexadecimal
End record:
Col.1 E
Col. 2-7 address of first executable instruction in object program
General description of the functions of the two passes of our simple assembler
Pass 1

1. Assign addresses to all statements in the program.
2. Save the values assigned to all labels for use in pass 2.
3. Perform some processing of assembler directives.

Pass 2

1. Assemble instructions.
2. Generate the data values defined by BYTE, WORD, etc.
3. Perform processing of assembler directives not done during pass 1.
4. Write the object program and the assembly listing.

Assembler algorithm and Data structures
Assembler uses two major internal data structures: the Operation Code Table(OPTAB) and the Symbol Table(SYMTAB).
OPTAB is used to look up mnemonic operation codes and translate them to their machine language equivalents.
SYMTAB is used to store values assigned to labels.
Location Counter LOCCTR is a variable that is used to help in the assignment of addresses.
LOCCTR is initialized to the beginning address specified in the START statement.
After each source statement is processed, the length of the assembled instruction or data area to be generated is added to LOCCTR.
The operation code table must contain the mnemonic operation code and its machine language equivalent.
During pass 1, OPTAB is used to look up and validate operation codes in the source program.
In pass 2, it is used to translate the operation codes to machine language.
OPTAB is usually organized as a hash table, with mnemonic operation code as a key.
The hash table organization is particularly appropriate, since it provides fast retrieval with a minimum of searching.
Pass 1 usually writes an intermediate file that contains each source statement together with its assigned address, error indicators, etc.
This working copy to the source file program also be used to retain the results of certain operation that may be performed during pass 1.

One-Pass Assembler

The main problem in trying to assemble a program in one pass involves forward references.
Instruction operands often are symbols that have not yet been defined in the source program.
Thus the assembler does not know what address to insert in the translated instruction.
The logic of the program often requires a forward jump for example, in escaping from a loop after testing some condition.
Requiring that he programmer eliminate all such forward jumps would be much too restrictive and inconvenient.
There are two main types of one-pass assembler.
One type produces object code directly in memory for immediate execution; the other type produces the usual kind of object program for later execution.
No object program is written out and no loader is needed.
This kind of load-and-go assembler is useful in a system that is oriented forward program development and testing.
The assembler simply generates object code instruction as it scans the source program.
The symbol used an operand is entered into the symbol table.
The address of the operand field of the instruction that refers to undefined symbol is added to a list of forward references associated with the symbol is table entry.
When the definition for a symbol is encountered the forward reference list for that symbol is scanned and the proper address is inserted into any instructions previously generated.
The first forward reference occurred on line, since the operand was not defined, the instruction assembled with no value assigned as the operand address.
When the symbol ENDFIL was defined the assembler placed its value in the SYMTAB entry; it then inserted this value into the instruction operand field as directed by the forward reference list.

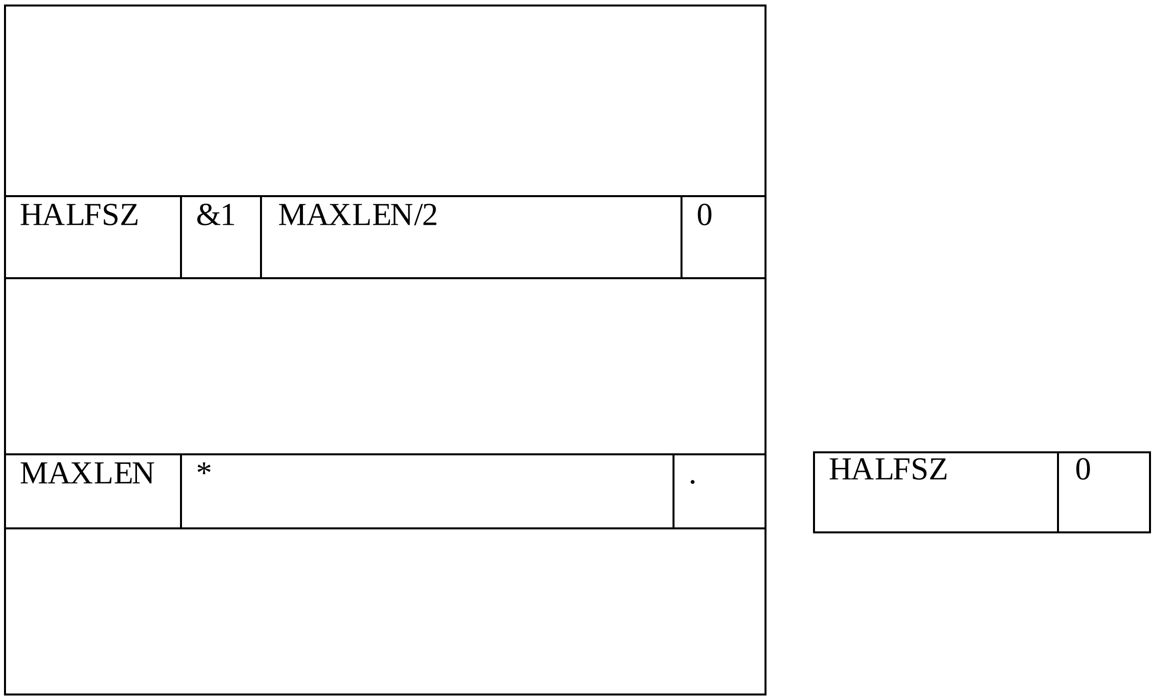
```
begin
read first input line
if OPCODE='START' then
begin
save #[OPERAND] as starting address
initialize LOCCTR as starting address
read next input line
end (if start)
else
initialize LOCCTR to 0
```

Multi-Pass Assembler

```
ALPHA      EQU       BETA
BETA       EQU       DELTA
DELTA      RESW      1
```

The symbol BETA cannot be assigned a value it is encountered during the first pass because DELTA has not yet been defined.
As a result, ALPHA cannot be evaluated during the second pass.

This means that any assembler that makes only two sequential passes over the source program cannot resolve such a sequence of definitions.
As a matter of fact, such forward references tend to create difficulty for a person reading the program as well as for the assembler.
It is not necessary for such an assembler to make more than two passes over the entire program.
There are several ways of accomplishing the task outlined above method we describe involves storing those symbol definitions are saved during pass 1.
Additional passes through these stored definitions are made as assembly progresses.
This process is followed by a normal pass 2.
There are several ways of accomplishing the task outlined above method we describe involves storing those symbol definitions that involve forward references in the symbol table.
SYMTAB would then simply contain pointer to the defining expression.
The symbol MAXLEN is also entered in symbol table, with the flag * identifying it as undefined.
Associated with this entry is a list of the symbols whose values depend on MAXLEN.
In this case there are two undefined symbols involved in the definition: BUFEND and BUFFER.
Both of these are entered into SYMTAB with lists indicating the dependence of MAXLEN upon them.



BUFEND	*		.
HALFSZ	&1	MAXLEN/2	0
MAXLEN	&2	BUFEND-BUFFER	.
BUFFER	*		.

MAXLEN	0

HALFSZ	0

MAXLEN	0

UNIT III

BASIC LOADER FUNCTIONS

The most fundamental functions of a loader –bringing an object program into memory and starting its execution.

Design of an Absolute Loader:

All functions are accomplished in a single pass. As each Text record is read, the object code it contains is moved to the indicated address in memory. When the End record is encountered, the loader jumps to the specified address to begin execution of the loaded program

Object program

```
HCOPY        ^ 001000^00107A
 T^002073^07382064^4C0000^05
```

PROGRAM LOADING OF AN ABSOLUTE PROGRAM

```
Memory              contents
address
0000        xxxxxxxx     xxxxxxxx    xxxxxxx    xxxxxxx
..
.
0FF0         xxxxxxxx     xxxxxxx     xxxxxxx    xxxxxxxx
1000        14103348     20390010   36281030  30101548
…..         . ……………          ………..      ………..      …………….
```

The contents of memory locations for which there is no Text record. This indicates that the previous contents of these locations remain unchanged.

Each byte of assembled code is given using its hexadecimal representation in character from. For example, the machine

```
Begin
        read Header record
        verify program name and length
        read first Text record
        while record type ≠ 'E' do
        begin
            {if object code is in character from,
            convert  in internal
            representation}

            move  object code to specified
            location in memory read    next
            object program record
            end
            jump to address specified in end
            record
            end
```

Loading an absolute loader.

Operation code for an STL instruction would be represented by the pair of characters "1" and "4".when these are read by the loader they will occupy two bytes of memory. In the instruction as loaded for execution, however, this operation code must be stored in a single byte with hexadecimal value 14.Thus each pair of bytes from the object program record must be packed together into one byte during loading.

A Simple Bootstrap Loader:

When a computer is first turned on or restarted, a special type of absolute loader, called a **bootstrap loader**, is executed. This bootstrap loads the first program to be run by the computer –usually an operating system. In this section, we examine a Very simple bootstrap loader for SIC/XE. In spite of its simplicity, this program illustrates almost all of the logic and coding techniques that are used in an absolute order.

```
                CLEAR          A            CLEAR REGISTER A TO ZERO
LOOP             LDX          #128        INITIALIZE REGISTER X TO HEX 80
                 JSUB         GETC          READ HEX  DIGIT FROM PROGRAM
BGING
                 RMO          A,S             SAVE IN REGISTER S
                   SHIFTL       S,4             MOVE TO HIGH-ORDER 4 BITS OF
BYTE
                   JSUB         GETC            GET NEXT HEX DIGIT
                   ADDR         S,A             COMBINE DIGITS TO FORM ONE
BYTE
                   STCH         O,X             STORE AT ADDRESS IN REFISTER X
                   TIXR          X,X              ADD 1 TO MEMORY ADDRESS
BEING                                                    LOADED
                   J              LOOP          LOOP UNTIL  END OF INOPPUT IS
REACHED
```

MACHINE-DEPENDENT LOADER FEATURES:

One of the most obvious is the need for the programmer to specify the actual address at which it will be loaded into memory. There is only room to run

one program at a time, and the starting address for this single user program is known in advance. On a larger and more advanced machine the situation is not quite as easy. Often like to run several independent programs together, sharing memory between them. This means that we do not know in advance where a program will be loaded.

Most such libraries contain many more subroutines than will be used by any one program. To make efficient use of memory, it is important to be able select and load exactly those routines that are needed. This could not be done effectively if all of the subroutines had preassignd absolute addresses.

Relocation:

Loaders that allow for program relocation are called **relocating** or **relative loaders.** In this section we discuss two methods for specifying relocation as part of the object program.

A Modification record is used to describe each part of the object code that must be changed when the program is relocation.

Most of the instructions in this program use relative or immediate addressing. The only portions of the assembled program that contain actual addresses are the extended format instruction on lines 15,35, and 65.Thus these are the only items whose values are affected by relocation.

Notice that there is one Modification record for each value that must be changed during relocation. Each Modification record specifies the starting address and length of the field whose value is to be altered. In this example, all modifications add the value of

```
Line    Loc          Source statement                 Object code
 5      0000    COPY     START    0
10      0000    FIRST    STL      RETADR              17202D
12      0003             LDB     #LENGTH              69202D
13      0006             BASE     LENGTH
15      000A    CLOOP   +JSUB     RDREC               4B101036
20      000D             LDA      LENGTH              032026
25      0010             COMP    #0                   290000
30      0013             JEQ      ENDFIL              332007
35      0017            +JSUB     WRREC               4B10105D
```

the symbol COPY, which represents the starting address of the program.

The Modification record scheme is a convenient means for specifying program relocation; however, it is not well suited for use with all machine architectures. This is a reloadable program written for the standard version of SIC. The important difference between this example and the one in that the standard SIC machine does not use relative addressing. In this program the addresses in all the instructions except RSUB must be modified when the program is relocated.

This would require 31 Modification records, which results in an object program more than twice as large as the one.

On a machine that primarily uses direct addressing and has a fixed instruction format, it is often more efficient to specify relocation using a different technique. The method applied to our SIC program. There are no Modification records. The Text records are the same as before except that there is a **relocation bit** associated with each word of object code. Since all SIC instructions are occupy one word, this means that there is one relocation big for each possible instruction.

The relocation bits are gathered together into a big mask following the length indicator in each text record.

Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	140033
15	0003	CLOOP	JSUB	RDREC	481039
20	0006		LDA	LENGTH	000036
25	0009		COMP	ZERO	280030
30	000C		JEQ	ENDFIL	300015
35	000F		JSUB	WRREC	481061

It eliminates some of the need for the loader to perform relocation.
For e.g. some such machines consider all memory references to relatives to

```
Begin
   Get PROGRAM from operating system
   While not end of input do
        Begin

      Read next record
           While record type ≠ 'E' do
           While record type ='T'
            Begin
```

```
                    Get length=second data
                    Mask bits(M) as third data
                           For (i=0,i<length,i++)
                                 If Mi=1 then
                                    Add PROGRAM at the location PROGADDR
                                       +specified address
                            Else
                              Move object code from record to location
                               PROGADDR + specified address
                Read next record
             end
      end

end
```

The beginning of the users assigned area of memory. The conversion of these relative addresses to actual addresses is performed as the program is executed.

Programming Linking:

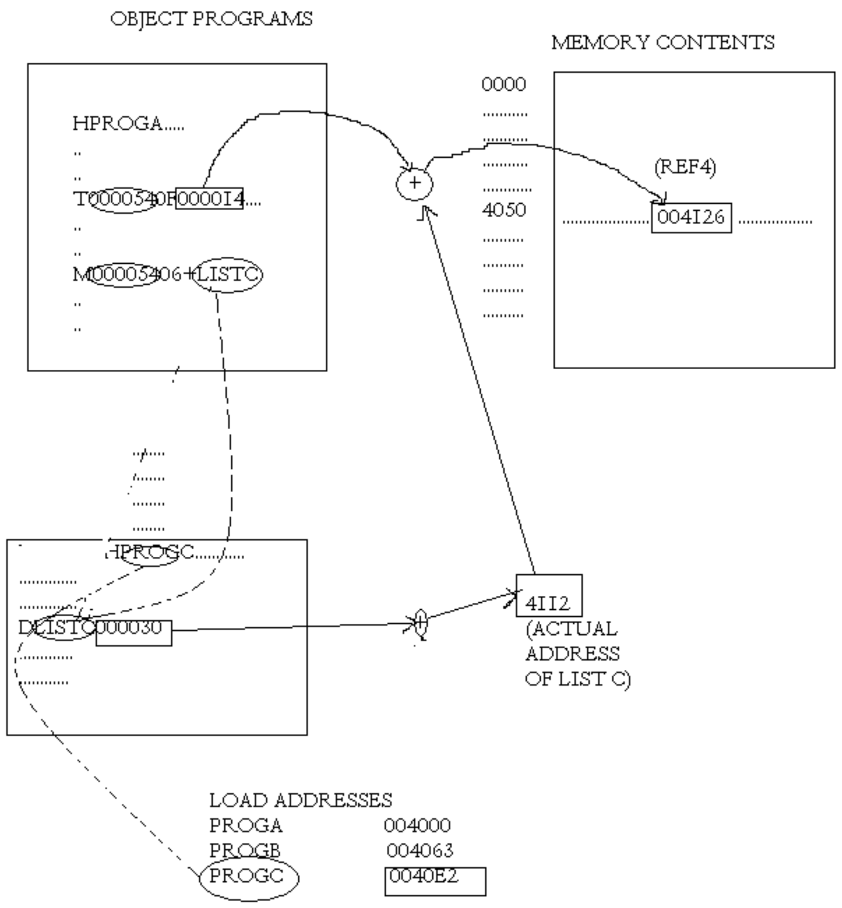
A program made up of three control sections. These control sections could be assembled together (or), they could be assembled independently of one another. However, there is no such thing as a program in this sense-there are only control sections that are to be linked, relocated, and loaded.

Consider the three programs which consist of single control section. Each program contains a list of items (LISTA, LISTA, LISTC); the ends of these lists are marked by the labels ENDA, ENDB, ENDC. Three of these are instruction operands (REF1 through REF3), and others are the values of data words (REF4 through REF8). This emphasizes the relationship between the relocation and linking processes.

Consider first the reference marked REF1. For the first program (PROGA), REF1 is simply a reference to label within the program. In the PROGB, on the other hand, the same operand refers to an external symbol. The assembler use an extended-format instruction with address field set to 00000. For PROGA, the operand expression consists of external reference plus a constant. In PROGB, the same expression is simply a local reference and is assembled using a program-counter relative instruction with no relocation or linking required.

To see this, consider REEF4.The assembler fir PROGA can evaluate all of the expression in REF4 exceot for the value of LISTC.This results in an initial value of (hexadecimal) 000014 PROGB contains no terms than can be evaluated by the assembler.

The object code therefore contains an initial value of 000000 and three Modification records PROGC, the assembler can supply the value of LISTC relative to the beginning of the program. The initial value of this data word contains the relative address of LISTC. Thus the expression in REF4 represents a simple external reference for PROGA, a more complicated external reference for PROGB, and a combination of relocation and external references for PROGC. These three programs as they might appear in memory after loading and linking. PROGA has been loaded starting at address 4000, with PROGB and PROGC immediately. Note that each of RWD8 has resulted in the same value in each of the three programs. For example, the value for reference REF4 in PROGA is located at address 4054.The initial value (from the Text record) is 000014. To this is added the address assigned to LISTC, which is 4112. In PROGB, the value for REF4 is located at relative address 70. To the initial value (000000), the value of LISTA(4040). The result, 004126, is the same as was obtained in PROGA. Similarly, the computation for REF4 in PROGC results in the same value. The same is also true for each of the other references REF5 through REF8.



(Relaxation and linking operations performed on REF4
From PROGA)
For the references that are instruction operands, the calculated values after loading do not always appear to be equal. This is because there is an additional address calculation step involved for program-counter relative (or base relative) instructions. In these cases it is the target addresses that are the same. When this instruction is executed, the program counter contains the value 4023. The resulting target address is 4040 this process as automatically providing the

needed relocation at execution time through the target address calculation. In PTOGB, on the other hand, reference REF1 is an extended format instruction that contains a direct (actual) address. This address, after linking is 4040-the same as the target address for the same reference in PROGA.

Algorithm and Data Structures for a Linking Loader

We use Modification records for relocation so that the linking and relocation function are performed using the same mechanism. The algorithm for a linking loader is considerably more complicated than the absolute loader algorithm. The input to such a loader consists of a set object programs that are to be linked together. In such a case the required linking operation cannot be performed until an address is assigned to the external symbol involved. Thus a linking loader usually makes two passes over its input, just as an assembler does. In terms of general function, the two passes of a linking loader are quite similar to the two passes of an assembler. Pass 1 assigns addresses to all external symbols, and pass 2 performs the actual loading, relocation, and linking.

The main data structure needed for our linking loader is an external symbol table ESTAB. This table, which is analogous to SYMTAB in our assembler algorithm, is used to store the name and address of each external symbol in the set of control section being loaded. The table also often indicates in which control section the symbol is defined. Two other important variables are PROGADDR   and CSADDR. PROGADDR is to be loaded. Its value is supplied to the loader by the operating system. CASDDR contains the starting address assigned to the control section currently being scanned by the loader. This value is added to all relative addresses within the control section to convert them to actual addresses.

During the first pass the loader is concerned only with Header and Define record types in the control section. The control section name from the Header record is entered into ESTAB. All external symbols appearing in the Define record for the control section are also entered into ESTAB. Their addresses are obtained by adding the value specified in the define record to CSADDR. When the End record is read, the control section length CSLTH is added to CSADDR. This calculation gives the starting address for the next control section in sequence.

At the end of pass 1, ESTAB contains all external symbols defined in the set of control sections together with the address assigned to each. This information is often useful in program debugging.

Control section	**symbol name**	**address**	**Length**
PROGA	LISTA ENDA	4000	0063
		4040	
		4054	
PROGB	LISTBENDA	40C3	007F
		40C3	
		40C3	
PROGC	LISTC ENDC	40E2	0051
		4112	
		4124	

Pass 1:

```
    being
    get PROGADDR from operating system
```

```
set CSADDR to control section length
while not end of input do
being
read next input record {Header record for control section}
set CSLTH to control section length
search ESGAB for control section name
if found then
set error flag {duplicate external symbol}
else
          enter control section name into ESTAB with value CSADDR while record
type ≠ ‘E’ do
begin
read next input record
if record type = ‘D’ then
for each symbol in the record do
begin
search ESTAB for symbol name


if  found than
set error flag (duplicate external symbol)
else
enter symbol into ESTAB with value (CSADDR + indicated address)
end {for}
end  {while ≠ ‘E’}
add CSLTH to CSADDR {starting address for next control section end {while
not EOF }
end {pass 1}
pass 2:
begin
set CSADDR  to PROGADDR
set  ESECADDR  to PROGADDR
while  not end of input do
begin
read next  input record {Header record}
set CSLTH to control  section length
if  record type ≠ ‘E’ do
begin
read next input record


if record type = ‘T’  then
begin
{if object code is in character form, convert into internal representation}
Move  object code from record to location
(CSADDR + specified address)
```

```
End {if ‘T’}
Else if record type = ‘M’ then
begin
search ESTAB for modifying symbol name
if found  then
add or subtract symbol value at location
(CSADDR + specified address)
Else
Set error flag (undefined external symbol )
End {if ‘M’}
If an address is specified {in end record } then
Set EXECADDR to (CSADDR + specified address )
else {while ≠ ‘E’}
if an address is specified {in End record} then

set EXECADDR to (CSADDR + specified address)
add CSLTH to CSADDR
end {while not  EOF}
        jump to location given by EXECADDR {to start execution on loaded
program
end {pass 2}
```

Pass 2 of our loader performs the actual loading, relocation, and linking of the program. CSADDR is used in the same way it was in Pass 1- it always contains the actual starting address of the control section currently being loaded. When a modification record is encountered, the symbol whose value is to be used for modification is looked up in ESTAB. This value is then added to or subtracted from the indicated location in memory.

The last step performed by the loader is usually the transferring of control to the loaded program to being execution. The End record for each control section may contain the address of the first instruction in that control section to be executed. If more than one control section specifies a transfer address, the loader arbitrarily uses the last one encountered. If no control section contains a transfer address, the loader uses the beginning of the linked program as the transfer point. If PROGADDR is taken to be 4000, the result should b the same

This algorithm can be made more efficient if a slight change is made in the object program format. This reference number is used in Modification records. Suppose we always assign the reference number 01 to the control section name. The other external reference symbols may be assigned numbers as part of the Refer record for the control section. The reference numbers are underlined in the Refer and Modification records for easier reading. The main advantage of this reference –number mechanism is that it avoids multiple searches of ESTAB for

the same symbol can be during the loading of a control section. An external reference symbol can be looked up in ESTAB once for each control section that uses it. The values for code modification can then be obtained by simply indexing into an array of these values. `

UNIT IV

BASIC COMPILER FUNCTIONS

- For the purpose of compiler construction, a high-level programming language is usually described in terms of a grammar.
- This grammar specifies the form, or syntax, of legal statements in the language. For example, an assignment

```
PROGRAM  STATS
VAR
      SUM,SUMSQ,I,VALUE,MEAN,VARIENCE : INTEGER
BEGIN
     SUM   :=0;
     SUMSQ  :=0;
     FOR   I  :=1 TO 100 DO
          BEGIN
             READ(VALUE);
             SUM  :=SUM+VALUE;
            SUMSQ  := SUMSQ+VALUE+VALUE
         END;
      MEAN  :=SUM DIV 100;
      VARIENCE  :=SUMSQ DIV 100 – MEAN * MEAN;
      WRITE(MEAN,VARIENCE)
END
```

- Statement might be defined by a grammar as a variable name, followed by an assignment operator (:=), followed by an expression.
- The problem of compilation then becomes one of matching statements written by the programmer to structures defined by the grammar, and generating the appropriate object code for each statement.
- It is convenient to regard a source program statement as a sequence of tokens rather than simply as a string of characters.
- Tokens may be thought of as the fundamental building blocks of the language. For example, a token might be a keyword, a variable name, an integer, an arithmetic operator, etc.

- The task of scanning the source statement, recognizing and classifying the various tokens, is known as lexical analysis.
- The part of the compiler that performs this analytic function is commonly called the scanner.
- After the tokens scan, each statement in the program must be recognized as some language construct, such as a declaration or an assignment statement, described by the grammar.
- This process, which is called syntactic analysis or parsing, is performed by a part of the compiler that is usually called the parser.
- The last step in the basic translation process is the generation of object code.
- Most compilers create machine-language programs directly instead of producing a symbolic program for later translation by an assembler.

GRAMMAR

- A grammar for a programming language is a formal description of the syntax of form, of programs and individual statements written language.
- The grammar does not describe the semantics, or meaning, of the various statements; such knowledge must be supplied in the code generation routines.
- Difference between syntax and semantics, consider the two statements.

```
                I: =J+K
        And
                X: = Y+I
```

- Where X and Y are REAL variables and I, J, K are INTEGER variables.
- These two statements have identical syntax.
- Each is an assignment statement; the value to be assigned is given by an expression that consists of two variables names separated by the operator +.
- The semantics of the two statements are quite different.
- The first statement specifies that the variables in the expression are to be added using integer arithmetic operations.
- The second statement specifies a floating point addition, with the integer operand I being converted to floating point before adding.
- A number of different notations can be used for writing grammars.
- The one we describe is called BNF (for Backus-Naur Form).
- BNF is not the most powerful syntax description tool.

- A BNF grammar consists of a set of rules, each of which defines the syntax of some construct in the programming language. Consider the example

<Read>:=READ (<id-list>)

- This is the definition of the syntax of a Pascal READ statement that is denoted in the grammar as <read>.
- The symbol: = can be read “is defined to be.” On the left of this symbol is the language construct being defined, <read>, and on the right is a description of the syntax being defined for it.
- Characters strings enclosed between the angle brackets < and > are called nonterminal symbols.
- These are the names of constructs defined in the grammar.
- Entries not enclosed in angle brackets are terminal symbols of the grammar.
- In this rule, the nonterminal symbols are <read> and <id-list>, and the terminal symbols are the tokens READ.
- This rule specifies that a <read> consists of the token READ, followed by the token construct<id-list>.
- The blanks spaces in the grammar rules are not significant.
- They have been included only to improve readability.

```
1. <prog>              ::= PROGRAM <prog-name> VAR <dec-list> BEGIN
<stmt-list> END.
2. <prog-name>         ::=id
3. <dec-list>          ::=<dec> | <dec-list> ; <dec>
4. <dec>               ::=<id-list> : <type>
5. <type>              ::=INTEGER
6. <ID-LIST>           ::=id | <id-list> , id
7. <stmt-list>         ::=<stmt> | <stmt-list> | <stmt>
8. <stmt>              ::=<assign> | <read> | <write> | <for>
9. <assign>            ::=id := <expr>
10. <exp>              ::= <term> | <expr> + <term> | <exp> - <term>
11. <term>             ::=<factor> | <term> * <factor> | <term> div <factor>
12. <factor>           ::=id | int | (<expr>)
13. <read>             ::=READ ( <id-list>)
14. <write>            ::=WRITE ( <id-list> )
15. <for>              ::=FOR <index-exp> DO <body>
 16. <index-exp>       ::=id := <exp> TO <exp>
 17. <body>            ::= <start> | BEGIN <stmt-list> END
```

<id-list>:= id | <id-list>, id

- This rule offers two possibilities, separated by the | symbol, for the syntax of an <id-list>
- The first alternative specifies that an <id-list> may consist simply of a token id.
- The second syntax alternative is an <id-list>, followed by the token "," (comma) followed by a token id. Which means the construct <id-list>is defined partially in terms of itself.
- By trying a few example

ALPHA

- Is an <id-list> that consists of a single id ALPHA;

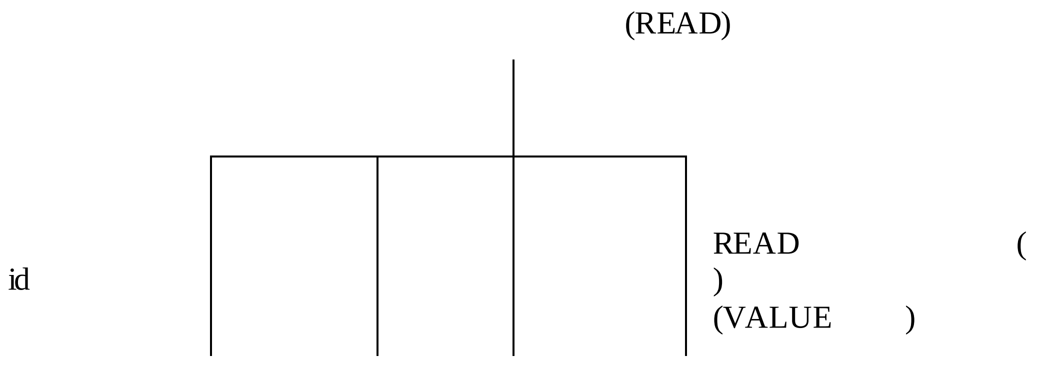
ALPHA BETA

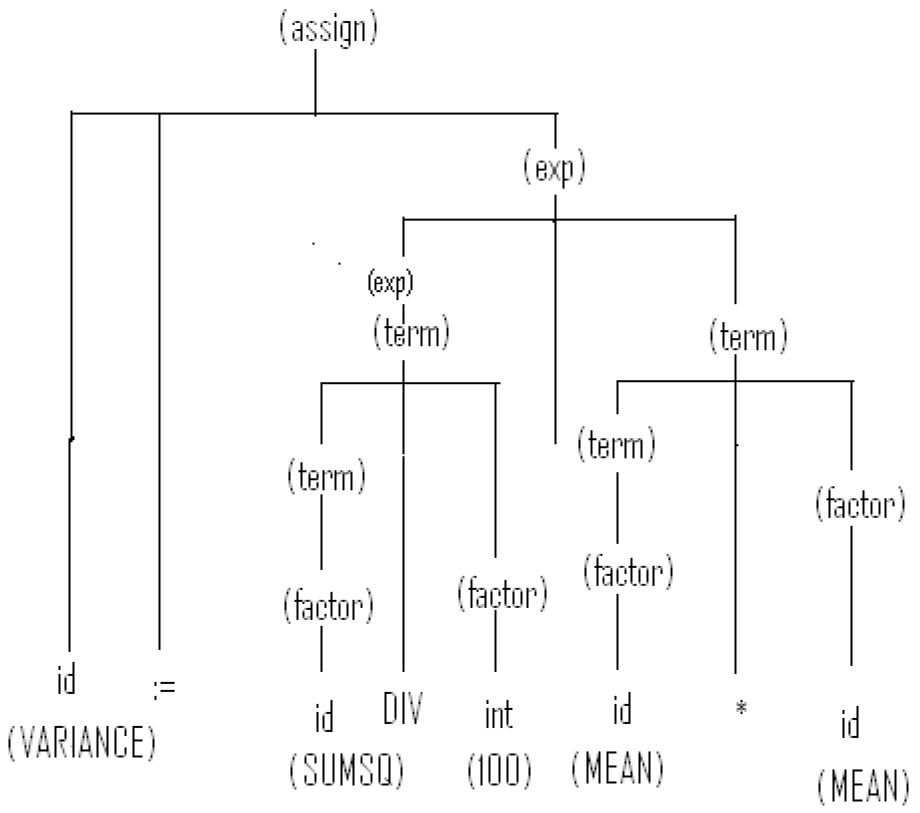
- Is an <id-list> that consists of anther <id-list> ALPHA, followed by a comma, followed by an id BETA, and so forth.
- It is often convenient to display the analysis of a source statement in terms of a grammar as a tree.
- This tree is usually called the parse tree, or syntax tree, for the statement.

READ (VALUE)

- In terms of the two rules just discussed.
- <assign> ::= id ::= <exp>
- That is, an <assign> consists of an id, followed by the token:=, followed by an expression <expr>.Rule 10 gives a definition of an <exp>:

<exp> ::= <term> | <exp> + <term> | <exp> - <term>





LEXICAL ANALYSIS

- Lexical analysis involves scanning the program to be complied and recognizing the tokens that make up the source statements.
- Scanners are usually designed to recognize keywords, operators, and identifiers, as well as integer, floating point numbers, character strings, and other similar items that are written as part of the source program.
- Items such as identifiers and integers are usually recognized directly as single tokens.

- As an alternative, these tokens could be defined as a part of the grammar.

```
<ident>  ::= <letter> | <ident> <letter> | <ident>  <digit>
 <letter>  ::= A | B | C | D | ….| Z
  <digit>  ::= 1 | 2 | 3 | 4 |….| 9
```

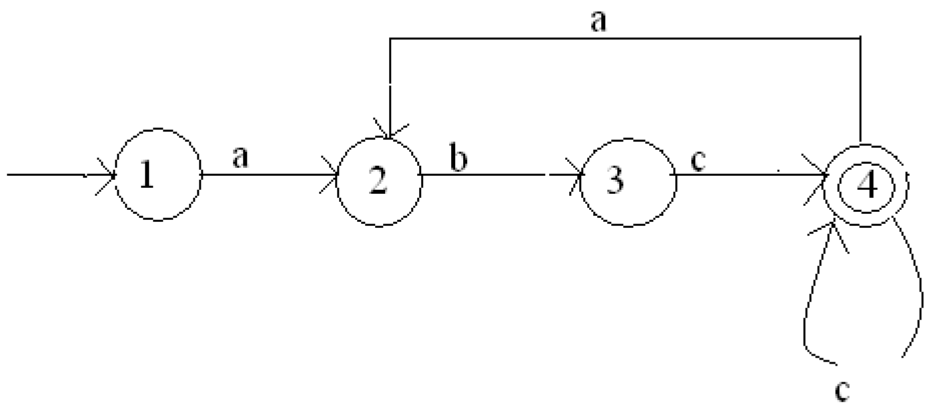
- In such a case the scanner would recognize as tokens the single character A, B, 0, 1, and so on.
- The parser would interpret a sequence of such characters as the language construct <ident>.
- A special – purpose mutine such as the scanner can perform this same function much more efficiently.
- Since, a large part of the source program consists of such multiple-character identifiers; this saving compilations time can be highly significant.
- Restrictions such as a limitation on the length of identifiers are easier to include in a scanner than in a general-purpose parsing routine.
- The scanners generally recognize both single and multiple characters tokens directly.
- The characters string READ would be integer string: = would be recognized as single assignment operators, not as: followed by =.
- The output of the scanner consists of a sequence of tokens.
- For efficiency of later use, each token is usually represented by some fixed-length code, such as integer, rather than as a variable-length characters string.

Token	code
PROGRAM	1
VAR	2
BEGIN	3
END	4
END.	5
INTEGER	6
FOR	7
READ	8
WRITE	9
TO	10
DO	11
;	12
:	13
,	14

:=	15
+	16
-	17
*	18
DIV	19
(	20
)	21
id	22
int	23

MODELING SCANNERS AS FINITE AUTOMATS

- The tokens of most programming languages can be recognized by a finite automation.
- A finite automation consists of a finite set of states and a set of transitions form one state to another.
- One of the states is designed as the starting state, and one or more states are designed as final states.
- Finite automats are often represented graphically.
- States are represented by circles and transition by arrows from one state to another.
- Each arrow is labeled with a character or set of characters that cause the specified transition to occur.
- The starting state has an arrow entering it that is not connected to anything else final states are identified by double circles.



Adc (recognized)

Abccabc (recognized)

Ac (not recognized)

- The automation starts in state 1 and examines the first character of the input string.
- The character a cause's automation to move form state 1 to state 2.
- The b causes a transition from state 2 to state 3, and the c causes a transition from state 3to state 4.
- At this point, all characters have been scanned, so the finite automation stops in state 4.
- The scanning of the first three characters happens exactly as described above.
- This, time however, there are still characters left in the input string.
- The fourth character of the string (the second c) causes the automation to remain in state 4(note the arrow labeled with c that loops back to state 4).
- The following a takes the automation back to state.
- At the end of the input string, the finite automation is again in state 4, so it recognizes the string abccabc.
- On the other hand the third input string, the finite automation begins in state 1, as before, and the causes a transition from state 1 to state 2.

- Now the next character to be scanned c.there is no transition from a state2 that is labeled with c.
- The automation must stop in state 2.
- Because this not a final state, the finite automation fails to recognizes the input string.
- If you try some other examples, you will discover that the finite automation recognizes tokens of the form abc…abc… where the grouping abc is repeated one or more times, and the c within each grouping may also be repeated.

SYNTACTIC ANALYSIS

- During syntactic analysis, the source statements written by the programming are recognized as language constructs described by the grammar being used.
- Parsing techniques are divided into two general classes bottom-up and top-down – according to the way in which the parse tree is constructed.
- Top-down methods begin with rule of the grammar that specifies the goal of the analysis, and attempt to construct the tree so that the terminal nodes match the statements being analyzed.
- Bottom-up methods begin with the terminal nodes of the tree, and attempt to combine these into successively higher-level nodes until the root is reached.

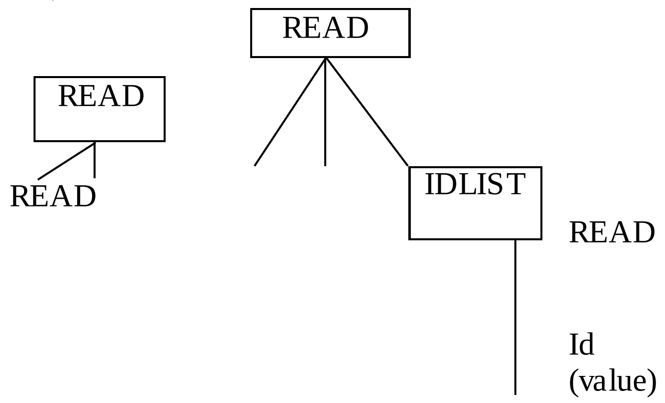
RECURSIVE-DESCENT PARSING

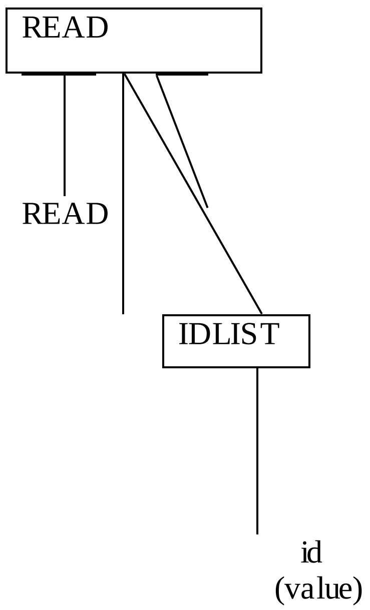
- A top-down method which is known as recursive descent is made up of a procedure for each nonterminal symbol in the grammar.
- When a procedure is called, it attempts to find a substring of the input, beginning with the current token, that can be interpreted as the nonterminal with which the procedure, or even for other nonterminals.
- If a procedure finds the nonterminal that is its goal, it returns an indication of success to its caller.
- If the procedure is unable to find a substring that can be interpreted as the desired nonterminal, it returns an indication of failure, or invokes an error diagnosis and recovery routine.
- The procedure for <read> in a recursive-descent parser first examines the next two input tokens, looking for READ and.

- If that procedure succeeds, the <read>procedure examines the next input token, looking for).
- If all these tests are successful, the <read>procedure returns an indication of success to its caller and advances to the next token following).

```
  Procedure READ
         Begin
            FOUND: = FALSE
            If TOKEN = 8 (READ)*then
              Begin
                 Advance to next token
                 If TOKEN = 20 { { } then
                   Begin
                       Advance to next token
                       If IDLIST returns success then
                         If TOKEN = 21 { { } then
                           Begin
                              FOUND: = TRUE
                              Advance to next token
                           End (if  }  }
                      End  (if  {  }
                If FOUND =TRUE then
                      Return success
                    Else
                      Return failure
                End (READ)
Procedure IDLIST
       Begin
          FOUND: = FALSE
          If TOKEN = 22 (id) then
            Begin
               FOUND: = TRUE
               Advance to next token
               While (TOKEN) = 14 {,}) and (FOUND = TRUE) do
                   Begin
                      Advance to next token
                      If TOKEN = 22 (id) then
                          Advance to next token
                        Else
                          FOUND: = FALSE
                    End (while)
               End (if id)
          If FOUND = TRUE then
```

```
                Return success
        Else
               Return failure
End (IDLIST
```



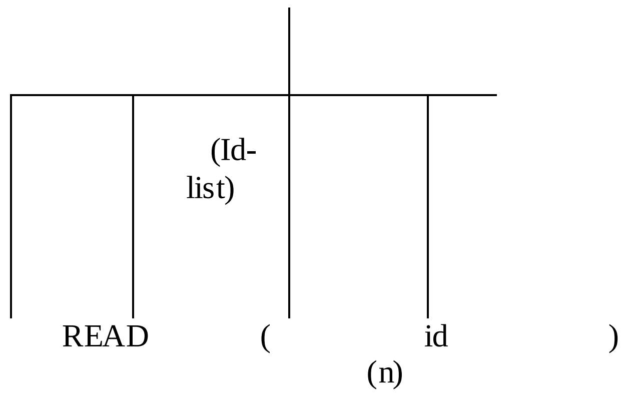


CODE GENERATION

- The code-generation techniques we describe involves a set of routines, one for each rule or alternative rule in the grammar.
- When the parser recognizes a portion of the source program according to some rule of the grammar, the corresponding routine is executed.

- Such routines are often called semantic routines because the processing performed is related to the meaning we associated with the corresponding construct in the language.
- In our simple scheme, these semantic routines generated object code directly, so we refer to them as code-generation routines.
- In more, complex compilers, the semantic routines might-generated an intermediate form of the program that would be analyzed further in an attempt to generate more efficient object code.
- As we have seen, neither of the parsing techniques discussed the constructs specified by this grammar.
- The operator –precedence method ignores certain nonterminals, and the recursive-descent method must use a slightly modified grammar.
- We choose to use this grammar in our discussion of code generation to emphasize the point that code-generation techniques need not be associated with any particular parsing method.
- The specific code to be generated clearly depends upon the computer for which the program is being complied.
- Our code-generation routines make use of two data structures for working storage a list and a stack.
- Items inserted into the list are removed in the order of their insertion, first in first out.
- Items pushed onto the stack are removed in the opposite order, last in first out.
- The variable LISTCOUNT is used to keep a count of the number of items currently in the list.
- The code-generation routines also make use of the token specifies.
- For a token id,s(id) is the name of the identifiers, or a pointer to the symbol-table entry for it.
- For a token int, s(int) is the value of the integer, such as #100.

(Read)



```
<id-list>:= id
          Add s (id) to list
          Add 1 to LISTCOUNT
 <id-list> ::= <id-list>  , Id
          Add s (id) to list
          Add 1 to LISTCOUNT
<read>:: = READ ( <id-list> )
           Generate (+JSUB XREAD)
            Record external reference to XREAD
           Generate (WORD LISTCOUNT)
           For each item on list do
               Begin
                   Removes s (ITEM) from list
                   Generate (WORD S (ITEM))
               End
            LISTCOUNT: =0
                        (B)
                     +JSUB       XREAD
                     WORD        1
                     WORD         VALUE
                          (C)
```

UNIT V:

BASIC COMPILER FUNCTIONS

- For the purpose of compiler construction, a high-level programming language is usually described in terms of a grammar.
- This grammar specifies the form, or syntax, of legal statements in the language. For example, an assignment

```
17  PROGRAM  STATS
18  VAR
19          SUM,SUMSQ,I,VALUE,MEAN,VARIENCE : INTEGER
```

```
BEGIN
        SUM   :=0;
        SUMSQ  :=0;
        FOR   I  :=1 TO 100 DO
                BEGIN
                     READ(VALUE);
                     SUM  :=SUM+VALUE;
                    SUMSQ  := SUMSQ+VALUE+VALUE
               END;
          MEAN  :=SUM DIV 100;
          VARIENCE  :=SUMSQ DIV 100 – MEAN * MEAN;
          WRITE(MEAN,VARIENCE)
END
```

- Statement might be defined by a grammar as a variable name, followed by an assignment operator (:=), followed by an expression.
- The problem of compilation then becomes one of matching statements written by the programmer to structures defined by the grammar, and generating the appropriate object code for each statement.
- It is convenient to regard a source program statement as a sequence of tokens rather than simply as a string of characters.
- Tokens may be thought of as the fundamental building blocks of the language. For example, a token might be a keyword, a variable name, an integer, an arithmetic operator, etc.
- The task of scanning the source statement, recognizing and classifying the various tokens, is known as lexical analysis.
- The part of the compiler that performs this analytic function is commonly called the scanner.
- After the tokens scan, each statement in the program must be recognized as some language construct, such as a declaration or an assignment statement, described by the grammar.
- This process, which is called syntactic analysis or parsing, is performed by a part of the compiler that is usually called the parser.
- The last step in the basic translation process is the generation of object code.
- Most compilers create machine-language programs directly instead of producing a symbolic program for later translation by an assembler.

GRAMMAR

- A grammar for a programming language is a formal description of the syntax of form, of programs and individual statements written language.
- The grammar does not describe the semantics, or meaning, of the various statements; such knowledge must be supplied in the code generation routines.
- Difference between syntax and semantics, consider the two statements.

I: =J+K

And

X: = Y+I

- Where X and Y are REAL variables and I, J, K are INTEGER variables.
- These two statements have identical syntax.
- Each is an assignment statement; the value to be assigned is given by an expression that consists of two variables names separated by the operator +.
- The semantics of the two statements are quite different.
- The first statement specifies that the variables in the expression are to be added using integer arithmetic operations.
- The second statement specifies a floating point addition, with the integer operand I being converted to floating point before adding.
- A number of different notations can be used for writing grammars.
- The one we describe is called BNF (for Backus-Naur Form).
- BNF is not the most powerful syntax description tool.
- A BNF grammar consists of a set of rules, each of which defines the syntax of some construct in the programming language. Consider the example

<Read>:=READ (<id-list>)

- This is the definition of the syntax of a Pascal READ statement that is denoted in the grammar as <read>.
- The symbol: = can be read “is defined to be.” On the left of this symbol is the language construct being defined, <read>, and on the right is a description of the syntax being defined for it.
- Characters strings enclosed between the angle brackets < and > are called nonterminal symbols.
- These are the names of constructs defined in the grammar.
- Entries not enclosed in angle brackets are terminal symbols of the grammar.

- In this rule, the nonterminal symbols are <read> and <id-list>, and the terminal symbols are the tokens READ.
- This rule specifies that a <read> consists of the token READ, followed by the token construct<id-list>.
- The blanks spaces in the grammar rules are not significant.
- They have been included only to improve readability.

```
1. <prog>              ::= PROGRAM <prog-name> VAR <dec-list> BEGIN
<stmt-list> END.
2. <prog-name>         ::=id
3. <dec-list>          ::=<dec> | <dec-list> ; <dec>
4. <dec>               ::=<id-list> : <type>
5. <type>              ::=INTEGER
6. <ID-LIST>           ::=id | <id-list> , id
7. <stmt-list>         ::=<stmt> | <stmt-list> | <stmt>
8. <stmt>              ::=<assign> | <read> | <write> | <for>
9. <assign>            ::=id := <expr>
10. <exp>              ::= <term> | <expr> + <term> | <exp> - <term>
11. <term>             ::=<factor> | <term> * <factor> | <term> div <factor>
12. <factor>           ::=id | int | (<expr>)
13. <read>             ::=READ ( <id-list>)
14. <write>            ::=WRITE ( <id-list> )
15. <for>              ::=FOR <index-exp> DO <body>
 16. <index-exp>       ::=id := <exp> TO <exp>
 17. <body>            ::= <start> | BEGIN <stmt-list> END
```

<id-list>:= id | <id-list>, id

- This rule offers two possibilities, separated by the | symbol, for the syntax of an <id-list>
- The first alternative specifies that an <id-list> may consist simply of a token id.
- The second syntax alternative is an <id-list>, followed by the token "," (comma) followed by a token id. Which means the construct <id-list>is defined partially in terms of itself.
- By trying a few example

ALPHA

- Is an <id-list> that consists of a single id ALPHA;

ALPHA BETA

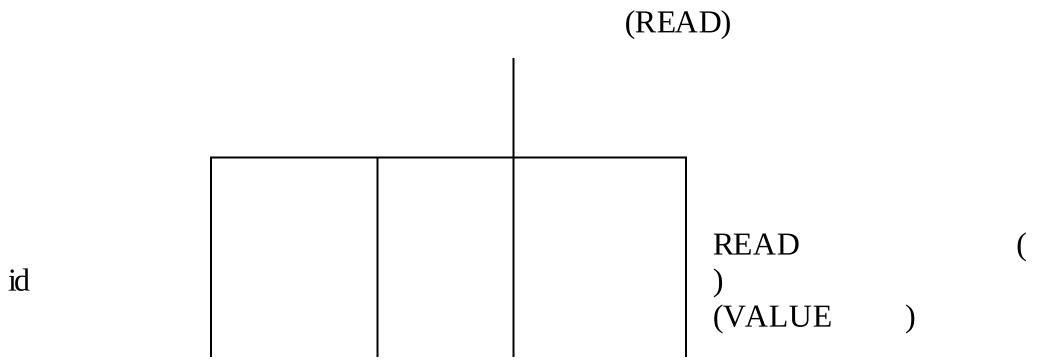
- Is an <id-list> that consists of anther <id-list> ALPHA, followed by a comma, followed by an id BETA, and so forth.

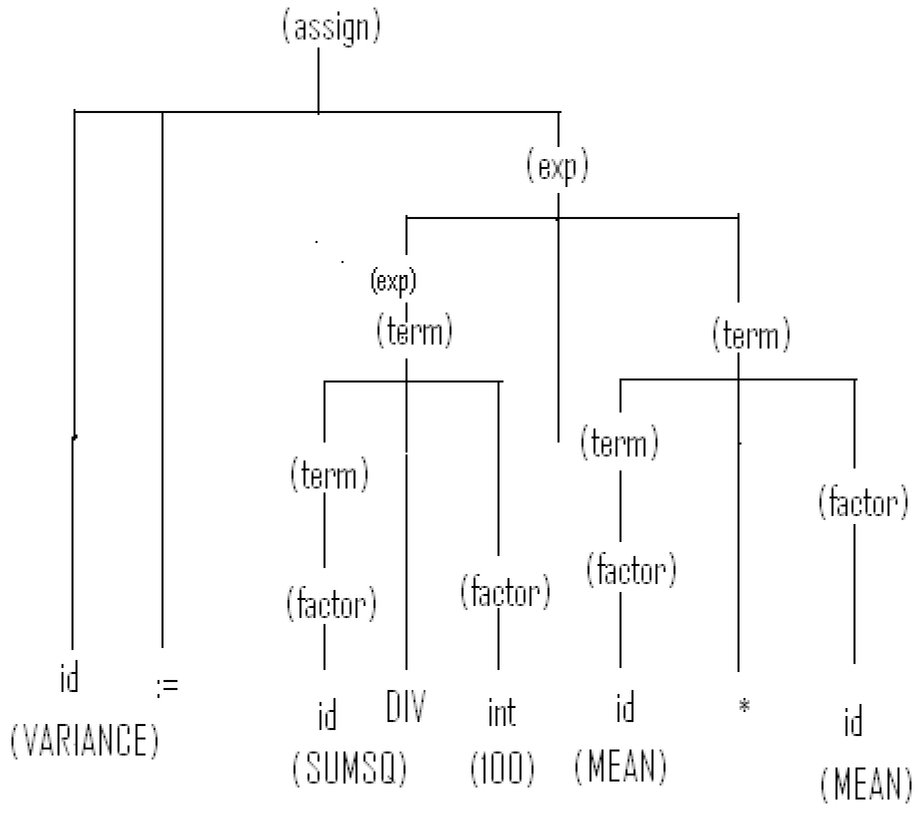
- It is often convenient to display the analysis of a source statement in terms of a grammar as a tree.
- This tree is usually called the parse tree, or syntax tree, for the statement.

READ (VALUE)

- In terms of the two rules just discussed.
- <assign> ::= id ::= <exp>
- That is, an <assign> consists of an id, followed by the token:=, followed by an expression <expr>.Rule 10 gives a definition of an <exp>:

<exp> ::= <term> | <exp> + <term> | <exp> - <term>





LEXICAL ANALYSIS

- Lexical analysis involves scanning the program to be complied and recognizing the tokens that make up the source statements.
- Scanners are usually designed to recognize keywords, operators, and identifiers, as well as integer, floating point numbers, character strings, and other similar items that are written as part of the source program.
- Items such as identifiers and integers are usually recognized directly as single tokens.
- As an alternative, these tokens could be defined as a part of the grammar.

<ident> ::= <letter> | <ident> <letter> | <ident> <digit>
<letter> ::= A | B | C | D | ….| Z
<digit> ::= 1 | 2 | 3 | 4 |….| 9

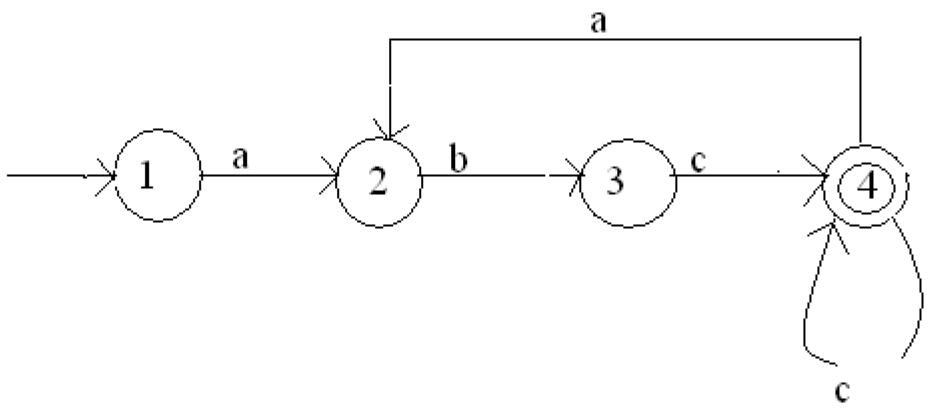
- In such a case the scanner would recognize as tokens the single character A, B, 0, 1, and so on.
- The parser would interpret a sequence of such characters as the language construct <ident>.
- A special – purpose mutine such as the scanner can perform this same function much more efficiently.
- Since, a large part of the source program consists of such multiple-character identifiers; this saving compilations time can be highly significant.
- Restrictions such as a limitation on the length of identifiers are easier to include in a scanner than in a general-purpose parsing routine.
- The scanners generally recognize both single and multiple characters tokens directly.
- The characters string READ would be integer string: = would be recognized as single assignment operators, not as: followed by =.
- The output of the scanner consists of a sequence of tokens.
- For efficiency of later use, each token is usually represented by some fixed-length code, such as integer, rather than as a variable-length characters string.

Token	code
PROGRAM	1
VAR	2
BEGIN	3
END	4
END.	5
INTEGER	6
FOR	7
READ	8
WRITE	9
TO	10
DO	11
;	12
:	13
,	14
: =	15
+	16
-	17

*	18
DIV	19
(	20
)	21
id	22
int	23

MODELING SCANNERS AS FINITE AUTOMATS

- The tokens of most programming languages can be recognized by a finite automation.
- A finite automation consists of a finite set of states and a set of transitions form one state to another.
- One of the states is designed as the starting state, and one or more states are designed as final states.
- Finite automats are often represented graphically.
- States are represented by circles and transition by arrows from one state to another.
- Each arrow is labeled with a character or set of characters that cause the specified transition to occur.
- The starting state has an arrow entering it that is not connected to anything else final states are identified by double circles.



Adc (recognized)

Abccabc	(recognized)
Ac	(not recognized)

- The automation starts in state 1 and examines the first character of the input string.
- The character a cause's automation to move form state 1 to state 2.
- The b causes a transition from state 2 to state 3, and the c causes a transition from state 3to state 4.
- At this point, all characters have been scanned, so the finite automation stops in state 4.
- The scanning of the first three characters happens exactly as described above.
- This, time however, there are still characters left in the input string.
- The fourth character of the string (the second c) causes the automation to remain in state 4(note the arrow labeled with c that loops back to state 4).
- The following a takes the automation back to state.
- At the end of the input string, the finite automation is again in state 4, so it recognizes the string abccabc.
- On the other hand the third input string, the finite automation begins in state 1, as before, and the causes a transition from state 1 to state 2.
- Now the next character to be scanned c.there is no transition from a state2 that is labeled with c.
- The automation must stop in state 2.
- Because this not a final state, the finite automation fails to recognizes the input string.
- If you try some other examples, you will discover that the finite automation recognizes tokens of the form abc…abc… where the grouping abc is repeated one or more times, and the c within each grouping may also be repeated.

SYNTACTIC ANALYSIS

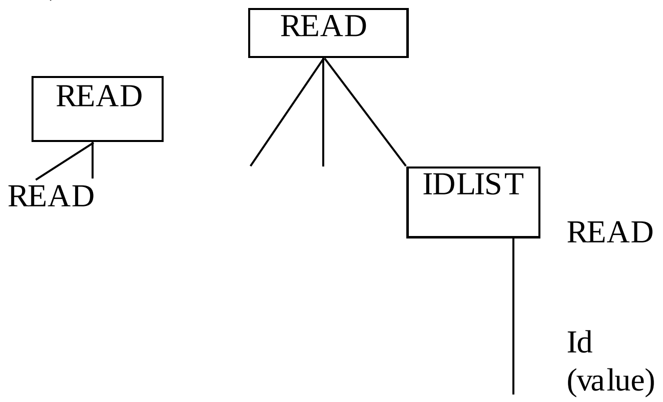
- During syntactic analysis, the source statements written by the programming are recognized as language constructs described by the grammar being used.
- Parsing techniques are divided into two general classes bottom-up and top-down – according to the way in which the parse tree is constructed.
- Top-down methods begin with rule of the grammar that specifies the goal of the analysis, and attempt to construct the tree so that the terminal nodes match the statements being analyzed.

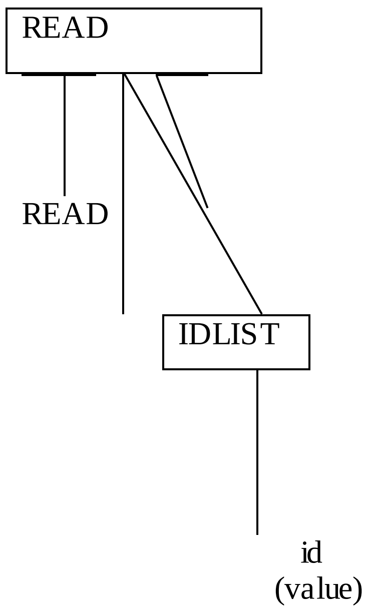
- Bottom-up methods begin with the terminal nodes of the tree, and attempt to combine these into successively higher-level nodes until the root is reached.

RECURSIVE-DESCENT PARSING

- A top-down method which is known as recursive descent is made up of a procedure for each nonterminal symbol in the grammar.
- When a procedure is called, it attempts to find a substring of the input, beginning with the current token, that can be interpreted as the nonterminal with which the procedure, or even for other nonterminals.
- If a procedure finds the nonterminal that is its goal, it returns an indication of success to its caller.
- If the procedure is unable to find a substring that can be interpreted as the desired nonterminal, it returns an indication of failure, or invokes an error diagnosis and recovery routine.
- The procedure for <read> in a recursive-descent parser first examines the next two input tokens, looking for READ and.
- If that procedure succeeds, the <read>procedure examines the next input token, looking for).
- If all these tests are successful, the <read>procedure returns an indication of success to its caller and advances to the next token following).

```
Procedure READ
        Begin
            FOUND: = FALSE
            If TOKEN = 8 (READ)*then
              Begin
                  Advance to next token
                  If TOKEN = 20 { { } then
                    Begin
                         Advance to next token
                         If IDLIST returns success then
                           If TOKEN = 21 { { } then
                             Begin
                                 FOUND: = TRUE
                                 Advance to next token
                              End (if  }  }
                       End  (if  {  }
                        If FOUND =TRUE then
                            Return success
                          Else
                            Return failure
                        End (READ)
Procedure IDLIST
    Begin
      FOUND: = FALSE
      If TOKEN = 22 (id) then
        Begin
          FOUND: = TRUE
          Advance to next token
          While (TOKEN) = 14 {,}) and (FOUND = TRUE) do
            Begin
              Advance to next token
              If TOKEN = 22 (id) then
                Advance to next token
              Else
                FOUND: = FALSE
            End (while)
        End (if id)
      If FOUND = TRUE then
        Return success
      Else
        Return failure
  End (IDLIST
```

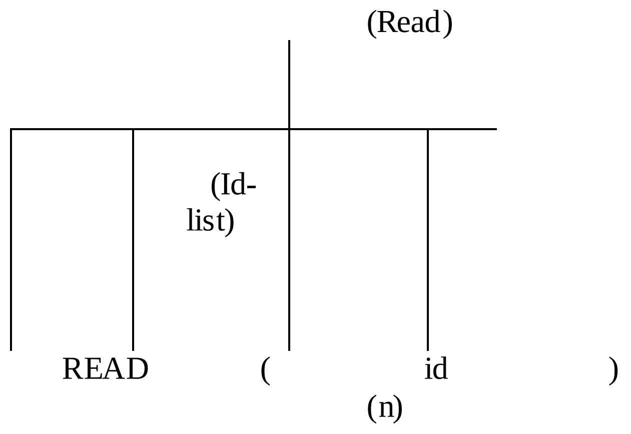




CODE GENERATION

- The code-generation techniques we describe involves a set of routines, one for each rule or alternative rule in the grammar.
- When the parser recognizes a portion of the source program according to some rule of the grammar, the corresponding routine is executed.
- Such routines are often called semantic routines because the processing performed is related to the meaning we associated with the corresponding construct in the language.
- In our simple scheme, these semantic routines generated object code directly, so we refer to them as code-generation routines.
- In more, complex compilers, the semantic routines might-generated an intermediate form of the program that would be analyzed further in an attempt to generate more efficient object code.
- As we have seen, neither of the parsing techniques discussed the constructs specified by this grammar.
- The operator –precedence method ignores certain nonterminals, and the recursive-descent method must use a slightly modified grammar.
- We choose to use this grammar in our discussion of code generation to emphasize the point that code-generation techniques need not be associated with any particular parsing method.
- The specific code to be generated clearly depends upon the computer for which the program is being complied.
- Our code-generation routines make use of two data structures for working storage a list and a stack.

- Items inserted into the list are removed in the order of their insertion, first in first out.
- Items pushed onto the stack are removed in the opposite order, last in first out.
- The variable LISTCOUNT is used to keep a count of the number of items currently in the list.
- The code-generation routines also make use of the token specifies.
- For a token id,s(id) is the name of the identifiers, or a pointer to the symbol-table entry for it.
- For a token int, s(int) is the value of the integer, such as #100.



```
<id-list>:= id
                  Add s (id) to list
                  Add 1 to LISTCOUNT
 <id-list> ::= <id-list>  , Id
                  Add s (id) to list
                  Add 1 to LISTCOUNT
<read>:: = READ ( <id-list> )
                    Generate (+JSUB XREAD)
                     Record external reference to XREAD
                    Generate (WORD LISTCOUNT)
```

```
For each item on list do
    Begin
        Removes s (ITEM) from list
        Generate (WORD S (ITEM))
    End
LISTCOUNT: =0
```

(B)

```
+JSUB    XREAD
WORD     1
WORD     VALUE
```

(C)