

## **MODULE-1**

### **INTRODUCTION TO SYSTEM SOFTWARE**

The Software is set of instructions or programs written to carry out certain task on digital computers. It is classified into system software and application software. System software consists of a variety of programs that support the operation of a computer. Application software focuses on an application or problem to be solved. System software consists of a variety of programs that support the operation of a computer.

Examples for system software are Operating system, compiler, assembler, macro processor, loader or linker, debugger, text editor, database management systems (some of them) and, software engineering tools. These software's make it possible for the user to focus on an application or other problem to be solved, without needing to know the details of how the machine works internally.

#### **System Software and Machine Architecture:**

One characteristic in which most system software differs from application software is machine dependency.

System software supports operation and use of computer. Application software provides solution to a problem. Assembler translates mnemonic instructions into machine code. The instruction formats, addressing modes etc., are of direct concern in assembler design. Similarly,

Compilers must generate machine language code, taking into account such hardware characteristics as the number and type of registers and the machine instructions available. Operating systems are directly concerned with the management of nearly all of the resources of a computing system.

There are aspects of system software that do not directly depend upon the type of computing system, general design and logic of an assembler, general design and logic of a compiler and code optimization techniques, which are independent of target machines. Likewise, the process of linking together independently assembled subprograms does not usually depend on the computer being used.

## The Simplified Instructional Computer (SIC):

Simplified Instructional Computer (SIC) is a hypothetical computer that includes the hardware features most often found on real machines. There are two versions of SIC, they are, standard model (SIC), and, extension version (SIC/XE) (extra equipment or extra expensive).

### SIC Machine Architecture:

We discuss here the SIC machine architecture with respect to its Memory and Registers, Data Formats, Instruction Formats, Addressing Modes, Instruction Set, Input and Output

- Memory:

There are  $2^{15}$  bytes in the computer memory, that is 32,768 bytes. It uses Little Endian format to store the numbers, 3 consecutive bytes form a word, each location in memory contains 8-bit bytes.

- Registers:

There are five registers, each 24 bits in length. Their mnemonic, number and use are given in the following table.

Mnemonic	Number	Use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; JSUB
PC	8	Program counter
SW	9	Status word, including CC

- Data Formats:

Integers are stored as 24-bit binary numbers. 2's complement representation is used for negative values, characters are stored using their 8-bit ASCII codes. No floating-point hardware on the standard version of SIC.

- Instruction Formats:

Opcode(8) x                      Address (15)

All machine instructions on the standard version of SIC have the 24-bit format as shown above

- Addressing Modes:

Mode	Indication	Target address calculation
Direct	$x = 0$	$TA = \text{address}$
Indexed	$x = 1$	$TA = \text{address} + (x)$

There are two addressing modes available which are as shown in the above table. Parentheses are used to indicate the contents of a register or a memory location.

- Instruction Set :

1. SIC provides, load and store instructions (LDA, LDX, STA, STX, etc.). Integer arithmetic operations: (ADD, SUB, MUL, DIV, etc.).
2. All arithmetic operations involve register A and a word in memory, with the result being left in the register. Two instructions are provided for subroutine linkage.
3. COMP compares the value in register A with a word in memory, this instruction sets a condition code CC to indicate the result. There are conditional jump instructions: (JLT, JEQ, JGT), these instructions test the setting of CC and jump accordingly.
4. JSUB jumps to the subroutine placing the return address in register L, RSUB returns by jumping to the address contained in register L.

- Input and Output:

Input and Output are performed by transferring 1 byte at a time to or from the rightmost 8 bits of register A (accumulator). The Test Device (TD) instruction tests whether the addressed device is ready to send or receive a byte of data. Read Data (RD), Write Data (WD) are used for reading or writing the data.

- **Data movement and Storage Definition**

LDA, STA, LDL, STL, LDX, STX ( A- Accumulator, L – Linkage Register, X – Index Register), all uses 3-byte word. LDCH, STCH associated with characters uses 1-byte. There are no memory-memory move instructions.

Storage definitions are

- WORD - ONE-WORD CONSTANT
- RESW - ONE-WORD VARIABLE
- BYTE - ONE-BYTE CONSTANT
- RESB - ONE-BYTE VARIABLE

### Example Programs (SIC):

#### Example 1: Simple data and character movement operation

LDA FIVE  
STA ALPHA  
  
LDCH      CHARZ  
  
STCH      C1  
  
ALPHA     RESW    1  
  
FIVE       WORD    5  
  
CHARZ     BYTE C'Z'  
  
C1        RESB 1

#### Example 2: Arithmetic operations

LDA ALPHA

ADD INCR

SUB ONE

STA BETA

.....

.....

.....

ONE WORD 1

ALPHA RESW 1

BEEITA RESW 1

INCR RESW 1

### Example 3: Looping and Indexing operation

LDX ZERO ; X = 0

MOVECH LDCH STR1, X ; LOAD A FROM STR1

STCH STR2, X ; STORE A TO STR2

TIX ELEVEN ; ADD 1 TO X, TEST

JLT MOVECH

STR1 BYTE C 'HELLO WORLD'

STR2 RESB 11

ZERO WORD 0

ELEVEN WORD 11

### Example 4: Input and Output operation

```
INLOOP TD INDEV      : TEST INPUT DEVICE  
        JEQ INLOOP    : LOOP UNTIL DEVICE IS READY  
        RD  INDEV     : READ ONE BYTE INTO A  
        STCH DATA    : STORE A TO DATA
```

```
OUTLP TD OUTDEV     : TEST OUTPUT DEVICE  
        JEQ OUTLP    : LOOP UNTIL DEVICE IS READY  
        LDCH DATA    : LOAD DATA INTO A  
        WD  OUTDEV    : WRITE A TO OUTPUT DEVICE
```

```
INDEV  BYTE X 'F5'   : INPUT DEVICE NUMBER  
OUTDEV BYTE X '08'   : OUTPUT DEVICE NUMBER  
DATA   RESB 1: ONE-BYTE VARIABLE
```

### Example 5: To transfer two hundred bytes of data from input device to memory

```
LDX  ZERO  
CLOOP TD INDEV  
        JEQ CLOOP  
        RD  INDEV  
        STCH RECORD, X
```

TIX B200

JLT CLOOP

INDEV BYTE X 'F5'

RECORD RESB 200

ZERO WORD 0

B200 WORD 200

## SIC/XE Machine Architecture:

- Memory

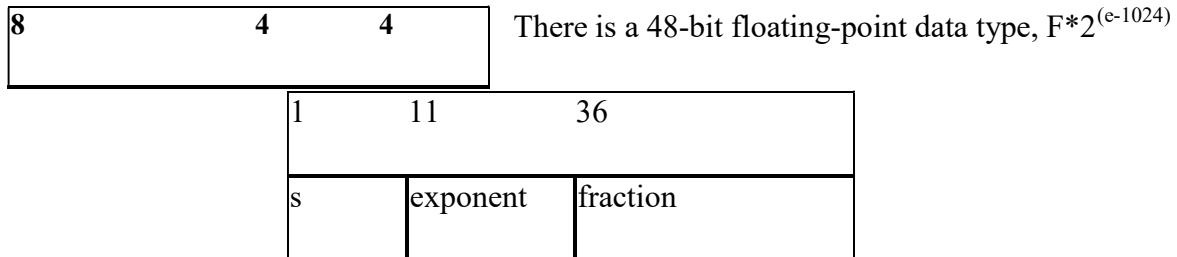
Maximum memory available on a SIC/XE system is 1 Megabyte ( $2^{20}$  bytes).

- Registers

Additional B, S, T, and F registers are provided by SIC/XE, in addition to the registers of SIC.

Mnemonic	Number	Special use
B	3	Base register
S	4	General working register
T	5	General working register
F	6	Floating-point accumulator (48 bits)

- Floating-point data type:

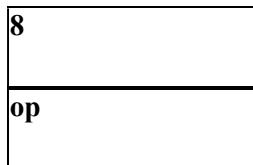


- Instruction Formats:

The new set of instruction formats from SIC/XE machine architecture are as follows.

- Format 1 (1 byte): contains only operation code (straight from table).
- Format 2 (2 bytes): first eight bits for operation code, next four for register 1 and following four for register 2. The numbers for the registers go according to the numbers indicated at the registers section (ie, register T is replaced by hex 5, F is replaced by hex 6).
- Format 3 (3 bytes): First 6 bits contain operation code, next 6 bits contain flags, last 12 bits contain displacement for the address of the operand. Operation code uses only 6 bits, thus the second hex digit will be affected by the values of the first two flags (n and i). The flags, in order, are: n, i, x, b, p, and e. Its functionality is explained in the next section. The last flag e indicates the instruction format (0 for 3 and 1 for 4).
- Format 4 (4 bytes): same as format 3 with an extra 2 hex digits (8 bits) for addresses that require more than 12 bits to be represented.

### **Format 1 (1 byte)**



### **Format 2 (2 bytes)**

<b>op</b>	<b>r1</b>	<b>r2</b>
-----------	-----------	-----------

Formats 1 and 2 are instructions do not reference memory at all

### Format 3 (3 bytes)

<b>6</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>12</b>
<b>op</b>	<b>n</b>	<b>i</b>	<b>x</b>	<b>b</b>	<b>p</b>	<b>e</b>

### Format 4 (4 bytes)

<b>6</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>20</b>
<b>op</b>	<b>n</b>	<b>i</b>	<b>x</b>	<b>b</b>	<b>p</b>	<b>e</b>	<b>address</b>

- Addressing modes & Flag Bits

Five possible addressing modes plus the combinations are as follows.

1. **Direct** (x, b, and p all set to 0): operand address goes as it is. n and i are both set to the same value, either 0 or 1. While in general that value is 1, if set to 0 for format 3 we can assume that the rest of the flags (x, b, p, and e) are used as a part of the address of the operand, to make the format compatible to the SIC format.
2. **Relative** (either b or p equal to 1 and the other one to 0): the address of the operand should be added to the current value stored at the B register (if b = 1) or to the value stored at the PC register (if p = 1)
3. **Immediate**(i = 1, n = 0): The operand value is already enclosed on the instruction (ie. lies on the last 12/20 bits of the instruction)
4. **Indirect**(i = 0, n = 1): The operand value points to an address that holds the address for the operand value.

5. **Indexed** ( $x = 1$ ): value to be added to the value stored at the register  $x$  to obtain real address of the operand. This can be combined with any of the previous modes except immediate.

The various flag bits used in the above formats have the following meanings

$e - > e = 0$  means format 3,  $e = 1$  means format 4

Bits  $x, b, p$  : Used to calculate the target address using relative, direct, and indexed addressing Modes.

Bits  $i$  and  $n$ : Says, how to use the target address

$b$  and  $p$  - both set to 0, disp field from format 3 instruction is taken to be the target address.

For a format 4 bits  $b$  and  $p$  are normally set to 0, 20 bit address is the target address

~~SVIT~~  
 $x - x$  is set to 1,  $X$  register value is added for target address calculation

~~SVIT~~  
 $i=1, n=0$  Immediate addressing, TA: TA is used as the operand value, no memory reference

~~SVIT~~  
 $i=0, n=1$  Indirect addressing, ((TA)): The word at the TA is fetched. Value of TA is taken as the address of the operand value

~~SVIT~~  
 $i=0, n=0$  or  $i=1, n=1$  Simple addressing, (TA): TA is taken as the address of the operand value

Two new relative addressing modes are available for use with instructions assembled using format 3.

Mode	Indication	Target address calculation
------	------------	----------------------------

Base relative	$b=1, p=0$	$TA=(B)+ disp$ $(0 \leq disp \leq 4095)$
Program-counter relative	$b=0, p=1$	$TA=(PC)+ disp$ $(-2048 \leq disp \leq 2047)$

- Instruction Set:

SIC/XE provides all of the instructions that are available on the standard version. In addition we have, Instructions to load and store the new registers LDB, STB, etc, Floating-point arithmetic operations, ADDF, SUBF, MULF, DIVF, Register move instruction : RMO,

Register-to-register arithmetic operations, ADDR, SUBR, MULR, DIVR and, Supervisor call instruction : SVC.

- Input and Output:

There are I/O channels that can be used to perform input and output while the CPU is executing other instructions. Allows overlap of computing and I/O, resulting in more efficient system operation. The instructions SIO, TIO, and HIO are used to start, test and halt the operation of I/O channels.

### Example Programs (SIC/XE)

#### Example 1: Simple data and character movement operation

```

LDA    #5
STA    ALPHA
LDA    #90
STCH   C1
.
.
```

ALPHA RESW 1

C1 RESB 1

### Example 2: Arithmetic operations

LDS INCR

LDA ALPHA

ADD S,A

SUB #1

STA BETA

.....

.....

ALPHA RESW 1

BETA RESW 1

INCR RESW 1

SVIT

### Example 3: Looping and Indexing operation

LDT #11

LDX #0 : X = 0

MOVECH LDCH STR1,X : LOAD A FROM STR1

STCH STR2,X : STORE A TO STR2

TIXR T : ADD 1 TO X, TEST (T)

JLT MOVECH

.....

.....

.....

STR1 BYTE C ‘HELLO WORLD’

STR2 RESB 11

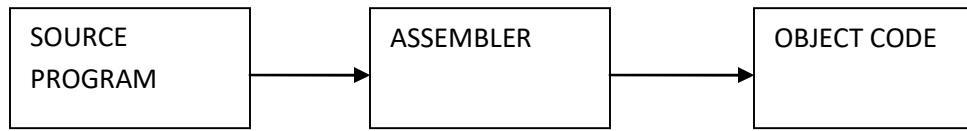
SVIT

## ASSEMBLERS-1

### Basic Assembler Functions:

The basic assembler functions are:

- Translating mnemonic language code to its equivalent object code.
- Assigning machine addresses to symbolic labels.



- The design of assembler can be to perform the following:
  - Scanning (tokenizing)
  - Parsing (validating the instructions)
  - Creating the symbol table
  - Resolving the forward references
  - Converting into the machine language
- SIC Assembler Directive:
  - START: Specify name & starting address.
  - END: End of the program, specify the first execution instruction.
  - BYTE, WORD, RESB, RESW
  - End of record: a null char(00)

End of file: a zero length record

- The design of assembler in other words:
  - Convert mnemonic operation codes to their machine language equivalents

- Convert symbolic operands to their equivalent machine addresses
- Decide the proper instruction format Convert the data constants to internal machine representations
- Write the object program and the assembly listing

So for the design of the assembler we need to concentrate on the machine architecture of the SIC/XE machine. We need to identify the algorithms and the various data structures to be used. According to the above required steps for assembling the assembler also has to handle *assembler directives*, these do not generate the object code but directs the assembler to perform certain operation. These directives are:

The assembler design can be done:

- Single pass assembler
- Multi-pass assembler

### Single-pass Assembler:

In this case the whole process of scanning, parsing, and object code conversion is done in single pass. The only problem with this method is resolving forward reference. This is shown with an example below:

10	1000	FIRST	STL	RETADR	141033
--					
--					
--					
--					
95	1033	RETADR	RESW	1	

In the above example in line number 10 the instruction STL will store the linkage register with the contents of RETADR. But during the processing of this instruction the value of this symbol is not known as it is defined at the line number 95. Since I single-pass assembler the scanning, parsing and object code conversion happens simultaneously. The instruction is fetched; it is scanned for tokens, parsed for syntax and semantic validity. If it is valid then it has to be converted to its equivalent object code. For this the object code is generated for the opcode STL and the value for the symbol RETADR need to be added, which is not available.

Due to this reason usually the design is done in two passes. So a multi-pass assembler resolves the forward references and then converts into the object code. Hence the process of the multi-pass assembler can be as follows:

### **Pass-1**

- Assign addresses to all the statements
- Save the addresses assigned to all labels to be used in *Pass-2*
- Perform some processing of assembler directives such as RESW, RESB to find the length of data areas for assigning the address values.
- Defines the symbols in the symbol table(generate the symbol table)

### **Pass-2**

- Assemble the instructions (translating operation codes and looking up addresses).
- Generate data values defined by BYTE, WORD etc.
- Perform the processing of the assembler directives not done during *pass-1*.
- Write the object program and assembler listing.

### **Assembler Design:**

The most important things which need to be concentrated is the generation of Symbol table and resolving *forward references*.

- Symbol Table:
  - This is created during pass 1
  - All the labels of the instructions are symbols
  - Table has entry for symbol name, address value.
- Forward reference:
  - Symbols that are defined in the later part of the program are called forward referencing.
  - There will not be any address value for such symbols in the symbol table in pass 1.

## Example Program:

The example program considered here has a main module, two subroutines

- Purpose of example program
  - Reads records from input device (code F1)
  - Copies them to output device (code 05)
  - At the end of the file, writes EOF on the output device, then RSUB to the operating system
- Data transfer (RD, WD)
  - A buffer is used to store record
  - Buffering is necessary for different I/O rates
  - The end of each record is marked with a null character (00)16
  - The end of the file is indicated by a zero-length record
- Subroutines (JSUB, RSUB)
  - RDREC, WRREC
  - Save link register first before nested jump

195					
200		:		SUBROUTINE TO WRITE RECORD FROM BUFFER	
205		-			
210	2061	WRREC	LIX	ZERO	041030
215	2064	WLOOP	TD	OUTPUT	E02079
220	2067		JEQ	WLOOP	302064
225	206A		LDCH	BUFFER,X	509039
230	206D		WD	OUTPUT	DC2079
235	2070		TIX	LENGTH	2C1036
240	2073		JLT	WLOOP	382064
245	2076		RSUB		400000
250	2079	OUTPUT	BYTE	X'05'	05
255			END	FIRST	

---

The first column shows the line number for that instruction, second column shows the addresses allocated to each instruction. The third column indicates the labels given to the statement, and is followed by the instruction consisting of opcode and operand. The last column gives the equivalent object code.

The *object code* later will be loaded into memory for execution. **The simple object program we use contains three types of records:**

- **Header record**

- Col. 1 H
- Col. 2~7 Program name
- Col. 8~13 Starting address of object program (hex)
- Col. 14~19 Length of object program in bytes (hex)

- **Text record**

- Col. 1 T
- Col. 2~7 Starting address for object code in this record (hex)

- Col. 8~9 Length of object code in this record in bytes (hex)
  - Col. 10~69 Object code, represented in hex (2 col. per byte)
- **End record**
    - Col.1 E
    - Col.2~7 Address of first executable instruction in object program (hex) “^” is only for separation only

## Simple SIC Assembler

The program below is shown with the object code generated. The column named LOC gives the machine addresses of each part of the assembled program (assuming the program is starting at location 1000). The translation of the source program to the object program requires us to accomplish the following functions.

1. Convert the mnemonic operation codes to their machine language equivalent.
2. Convert symbolic operands to their equivalent machine addresses.
3. Build the machine instructions in the proper format.
4. Convert the data constants specified in the source program into their internal machine representations in the proper format.
5. Write the object program and assembly listing.

All these steps except the second can be performed by sequential processing of the source program, one line at a time. Consider the instruction

10	1000	LDA	ALPHA	00-----
----	------	-----	-------	---------

This instruction contains the forward reference, i.e. the symbol ALPHA is used is not yet defined. If the program is processed ( scanning and parsing and object code conversion) is done line-by-line, we will be unable to resolve the address of this symbol. Due to this problem most of the assemblers are designed to process the program in two passes.

In addition to the translation to object program, the assembler has to take care of handling assembler directive. These directives do not have object conversion but gives direction to the assembler to perform some function. Examples of directives are the statements like BYTE and WORD, which directs the assembler to reserve memory locations without generating data values. The other directives are START which indicates the beginning of the program and END indicating the end of the program.

The assembled program will be loaded into memory for execution. The simple object program contains three types of records: Header record, Text record and end record. The header record contains the 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 to be loaded. The end record marks the end of the object program and specifies the address where the execution is to begin.

The format of each record is as given below.

Header record:

Col 1	H
Col. 2-7	Program name
Col 8-13	Starting address of object program (hexadecimal)
Col 14-19	Length of object program in bytes (hexadecimal)

SVIT

Text record:

Col. 1	T
Col 2-7.	Starting address for object code in this record (hexadecimal)
Col 8-9	Length off object code in this record in bytes (hexadecimal)
Col 10-69	Object code, represented in hexadecimal (2 columns per byte of object code)

End record:

Col. 1	E
Col 2-7	Address of first executable instruction in object program (hexadecimal)

The assembler can be designed either as a single pass assembler or as a two pass assembler. The general description of both passes is as given below:

- Pass 1 (define symbols)
  - Assign addresses to all statements in the program
  - Save the addresses assigned to all labels for use in Pass 2
  - Perform assembler directives, including those for address assignment, such as BYTE and RESW
- Pass 2 (assemble instructions and generate object program)
  - Assemble instructions (generate opcode and look up addresses)
  - Generate data values defined by BYTE, WORD
  - Perform processing of assembler directives not done during Pass 1
  - Write the object program and the assembly listing

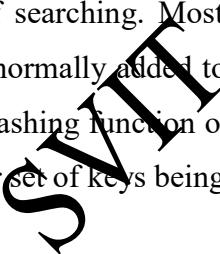
## Algorithms and Data structure

The simple assembler uses two major internal data structures: the operation Code Table (OPTAB) and the Symbol Table (SYMTAB).

### OPTAB:

- It is used to lookup mnemonic operation codes and translates them to their machine language equivalents. In more complex assemblers the table also contains information about instruction format and length.

- In pass 1 the OPTAB is used to look up and validate the operation code in the source program. In pass 2, it is used to translate the operation codes to machine language. In simple SIC machine this process can be performed in either in pass 1 or in pass 2. But for machine like SIC/XE that has instructions of different lengths, we must search OPTAB in the first pass to find the instruction length for incrementing LOCCTR.
- In pass 2 we take the information from OPTAB to tell us which instruction format to use in assembling the instruction, and any peculiarities of the object code instruction.
- OPTAB is usually organized as a hash table, with mnemonic operation code as the key. The hash table organization is particularly appropriate, since it provides fast retrieval with a minimum of searching. Most of the cases the OPTAB is a static table- that is, entries are not normally added to or deleted from it. In such cases it is possible to design a special hashing function or other data structure to give optimum performance for the particular set of keys being stored.



### SYMTAB:

- This table includes the name and value for each label in the source program, together with flags to indicate the error conditions (e.g., if a symbol is defined in two different places).
- During Pass 1: labels are entered into the symbol table along with their assigned address value as they are encountered. All the symbols address value should get resolved at the pass 1.
- During Pass 2: Symbols used as operands are looked up the symbol table to obtain the address value to be inserted in the assembled instructions.
- SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval. Since entries are rarely deleted, efficiency of deletion is the important criteria for optimization.

- Both pass 1 and pass 2 require reading the source program. Apart from this an intermediate file is created by pass 1 that contains each source statement together with its assigned address, error indicators, etc. This file is one of the inputs to the pass 2.
- A copy of the source program is also an input to the pass 2, which is used to retain the operations that may be performed during pass 1 (such as scanning the operation field for symbols and addressing flags), so that these need not be performed during pass 2. Similarly, pointers into OPTAB and SYMTAB is retained for each operation code and symbol used. This avoids need to repeat many of the table-searching operations.

## **LOCCTR:**

Apart from the SYMTAB and OPTAB, this is another important variable which helps in the assignment of the addresses. LOCCTR is initialized to the beginning address mentioned in the START statement of the program. After each statement is processed, the length of the assembled instruction is added to the LOCCTR to make it point to the next instruction. Whenever a label is encountered in an instruction the LOCCTR value gives the address to be associated with that label.

## **The Algorithm for Pass 1:**

Begin

read first input line

if OPCODE = ‘START’ then begin

    save #[Operand] as starting addr

    initialize LOCCTR to starting address

    write line to intermediate file

    read next line

end( if START)

```
else
    initialize LOCCTR to 0
    While OPCODE != 'END' do
        begin
            if this is not a comment line then
                begin
                    if there is a symbol in the LABEL field then
                        begin
                            search SYMTAB for LABEL
                            if found then
                                set error flag (duplicate symbol)
                            else
                                (if symbol)
                            search OPTAB for OPCODE
                            if found then
                                add 3 (instr length) to LOCCTR
                            else if OPCODE = 'WORD' then
                                add 3 to LOCCTR
                            else if OPCODE = 'RESW' then
                                add 3 * #[OPERAND] to LOCCTR
                            else if OPCODE = 'RESB' then
```

**SVIT**

```
add #[OPERAND] to LOCCTR

else if OPCODE = 'BYTE' then

begin

    find length of constant in bytes

    add length to LOCCTR

end

else

set error flag (invalid operation code)

end (if not a comment)

write line to intermediate file

read next input line

end { while not END}

write last line to intermediate file

Save (LOCCTR – starting address) as program length

End {pass 1}

• The algorithm scans the first statement START and saves the operand field (the address) as the starting address of the program. Initializes the LOCCTR value to this address. This line is written to the intermediate line.

• If no operand is mentioned the LOCCTR is initialized to zero. If a label is encountered, the symbol has to be entered in the symbol table along with its associated address value.

• If the symbol already exists that indicates an entry of the same symbol already exists. So an error flag is set indicating a duplication of the symbol.
```

SVIT

- It next checks for the mnemonic code, it searches for this code in the OPTAB. If found then the length of the instruction is added to the LOCCTR to make it point to the next instruction.
- If the opcode is the directive WORD it adds a value 3 to the LOCCTR. If it is RESW, it needs to add the number of data word to the LOCCTR. If it is BYTE it adds a value one to the LOCCTR, if RESB it adds number of bytes.
- If it is END directive then it is the end of the program it finds the length of the program by evaluating current LOCCTR – the starting address mentioned in the operand field of the END directive. Each processed line is written to the intermediate file.

### The Algorithm for Pass 2:

```
Begin  
    read 1st input line  
    if OPCODE = ‘START’ then  
        begin  
            write listing line  
            read next input line  
        end  
        write Header record to object program  
        initialize 1st Text record  
    while OPCODE != ‘END’ do  
        begin  
            if this is not comment line then  
                begin
```

SVIT

```
search OPTAB for OPCODE

if found then

begin

if there is a symbol in OPERAND field then

begin

search SYMTAB for OPERAND field then

if found then

begin

store symbol value as operand address

else

begin

store 0 as operand address

set error flag (undefined symbol)

end

end (if symbol)

else store 0 as operand address

assemble the object code instruction

else if OPCODE = ‘BYTE’ or ‘WORD’ then

convert constant to object code

if object code doesn’t fit into current Text record then

begin
```

SVIT

```
Write text record to object code  
initialize new Text record  
end  
add object code to Text record  
end {if not comment}  
write listing line  
read next input line  
end  
write listing line  
read next input line  
write last listing line  
End {Pass 2}
```

SVIT

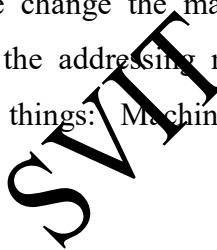
Here the first input line is read from the intermediate file. If the opcode is START, then this line is directly written to the list file. A header record is written in the object program which gives the starting address and the length of the program (which is calculated during pass 1). Then the first text record is initialized. Comment lines are ignored. In the instruction, for the opcode the OPTAB is searched to find the object code.

If a symbol is there in the operand field, the symbol table is searched to get the address value for this which gets added to the object code of the opcode. If the address not found then zero value is stored as operands address. An error flag is set indicating it as undefined. If symbol itself is not found then store 0 as operand address and the object code instruction is assembled.

If the opcode is BYTE or WORD, then the constant value is converted to its equivalent object code( for example, for character EOF, its equivalent hexadecimal value ‘454f46’ is stored). If the object code cannot fit into the current text record, a new text record is created and the rest of the instructions object code is listed. The text records are written to the object program. Once the whole program is assemble and when the END directive is encountered, the End record is written.

### Design and Implementation Issues

Some of the features in the program depend on the architecture of the machine. If the program is for SIC machine, then we have only limited instruction formats and hence limited addressing modes. We have only single operand instructions. The operand is always a memory reference. Anything to be fetched from memory requires more time. Hence the improved version of SIC/XE machine provides more instruction formats and hence more addressing modes. The moment we change the machine architecture the availability of number of instruction formats and the addressing modes changes. Therefore the design usually requires considering two things: Machine-dependent features and Machine-independent features.



### Machine-Dependent Assembler Features:

- Instruction formats and addressing modes
- Program relocation.

### Instruction formats and Addressing Modes

The instruction formats depend on the memory organization and the size of the memory. In SIC machine the memory is byte addressable. Word size is 3 bytes. So the size of the memory is  $2^{12}$  bytes. Accordingly it supports only one instruction format. It has only two registers: register A and Index register. Therefore the addressing modes supported by this architecture are direct, indirect, and indexed. Whereas the memory of a SIC/XE machine is  $2^{20}$  bytes (1 MB). This supports four different types of instruction types, they are:

- 1 byte instruction
  - 2 byte instruction
  - 3 byte instruction
  - 4 byte instruction
- 
- Instructions can be:
    - Instructions involving register to register
    - Instructions with one operand in memory, the other in Accumulator (Single operand instruction)
    - Extended instruction format
  - Addressing Modes are:
    - Index Addressing(SIC): Opcode m, x
    - Indirect Addressing: Opcode @m
    - PC-relative: Opcode m
    - Base relative: Opcode m
    - Immediate addressing: Opcode #c

SVIT

### 1. Translations for the Instruction involving Register-Register addressing mode:

**During pass 1** the registers can be entered as part of the symbol table itself. The value for these registers is their equivalent numeric codes. **During pass2**, these values are assembled along with the mnemonics object code. If required a separate table can be created with the register names and their equivalent numeric values.

### 2. Translation involving Register-Memory instructions:

In SIC/XE machine there are four instruction formats and five addressing modes. For formats and addressing modes

Among the instruction formats, format -3 and format-4 instructions are Register-Memory type of instruction. One of the operand is always in a register and the other operand is in the

memory. The addressing mode tells us the way in which the operand from the memory is to be fetched.

There are two ways: Program-counter relative and Base-relative. This addressing mode can be represented by either using format-3 type or format-4 type of instruction format. In format-3, the instruction has the opcode followed by a 12-bit displacement value in the address field. Where as in format-4 the instruction contains the mnemonic code followed by a 20-bit displacement value in the address field.

## Program-Counter Relative:

In this usually format-3 instruction format is used. The instruction contains the opcode followed by a 12-bit displacement value.

The range of displacement values are from 0 -2048. This displacement (should be small enough to fit in a 12-bit field) value is added to the current contents of the program counter to get the target address of the operand required by the instruction.

This is relative way of calculating the address of the operand relative to the program counter. Hence the displacement of the operand is relative to the current program counter value. The following example shows how the address is calculated:

10 0000 FIRST STL RETADR  
RETADR is at address  $(0030)_{16}$   
After the SIC fetches this instruction,  $(PC) = (0003)_{16}$   
 $TA = (PC) + disp \Rightarrow disp = TA - (PC) = 0030 - 0003 = (02D)_{16}$

op	n	i	x	b	p	e	disp
000101	1	1	0	0	1	0	02D $\Rightarrow 17202D$

40	0017	J	CLOOP																
CLOOP is at address $(0006)_{16}$																			
After the SIC fetches this instruction, $(PC) = (001A)_{16}$																			
$TA = (PC) + \text{disp} \Rightarrow \text{disp} = TA - (PC) = 0006 - 001A = (FEC)_{16}$																			
		<table border="1"> <thead> <tr> <th>op</th> <th>n</th> <th>i</th> <th>x</th> <th>b</th> <th>p</th> <th>e</th> <th>disp</th> </tr> </thead> <tbody> <tr> <td>001111</td> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> <td>FEC</td> </tr> </tbody> </table>	op	n	i	x	b	p	e	disp	001111	1	1	0	0	1	0	FEC	<p>12-bits FEC <math>\Rightarrow 3F2FEC</math></p>
op	n	i	x	b	p	e	disp												
001111	1	1	0	0	1	0	FEC												
70	002A	J	@RETADR																
CLOOP is at address $(0030)_{16}$																			
After the SIC fetches this instruction, $(PC) = (002D)_{16}$																			
$TA = (PC) + \text{disp} \Rightarrow \text{disp} = TA - (PC) = 0030 - 002D = (0003)_{16}$																			
		<table border="1"> <thead> <tr> <th>op</th> <th>n</th> <th>i</th> <th>x</th> <th>b</th> <th>p</th> <th>e</th> <th>disp</th> </tr> </thead> <tbody> <tr> <td>001111</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> <td>003</td> </tr> </tbody> </table>	op	n	i	x	b	p	e	disp	001111	1	0	0	0	1	0	003	<p>3E2003</p>
op	n	i	x	b	p	e	disp												
001111	1	0	0	0	1	0	003												

## Base-Relative Addressing Mode:

In this mode the base register is used to mention the displacement value. Therefore the target address is

$$TA = (\text{base}) + \text{displacement value}$$

- This addressing mode is used when the range of displacement value is not sufficient. Hence the operand is not relative to the instruction as in PC-relative addressing mode. Whenever this mode is used it is indicated by using a directive BASE.
- The moment the assembler encounters this directive the next instruction uses base-relative addressing mode to calculate the target address of the operand.
- When NOBASE directive is used then it indicates the base register is no more used to calculate the target address of the operand. Assembler first chooses PC-relative, when the displacement field is not enough it uses Base-relative.

LDB #LENGTH (*instruction*)

BASE LENGTH (*directive*)

:

NOBASE

For example:

```

12      0003  LDB      #LENGTH          69202D
        BASE      LENGTH
        ::

100     0033  LENGTH    RESW      1
        BUFFER    RESB      4096
        ::

160     104E  STCH      BUFFER,   X      57C003
        TIXR      T          B850

```

In the above example the use of directive **BASE** indicates that Base-relative addressing mode is to be used to calculate the target address. PC-relative is no longer used. The value of the LENGTH is stored in the base register. If PC-relative is used then the target address calculated is:

- The LDB instruction loads the value of length in the base register which 0033. BASE directive explicitly tells the assembler that it has the value of LENGTH.

BUFFER is at location  $(0036)_{16}$

$$(B) = (0033)_{16}$$

$$\text{disp} = 0036 - 0033 = (0003)_{16}$$

op      n    i    x    b    p    e	disp									
<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>010101</td><td>1</td><td>1</td></tr><tr><td></td><td>1</td><td>1</td></tr><tr><td></td><td>0</td><td>0</td></tr></table>	010101	1	1		1	1		0	0	003 $\Rightarrow$ 57C003
010101	1	1								
	1	1								
	0	0								

```

20      000A          LDA      LENGTH          032026

```

::

175	1056	EXIT	STX	LENGTH	134000
-----	------	------	-----	--------	--------

Consider Line 175. If we use PC-relative

$$\text{Disp} = \text{TA} - (\text{PC}) = 0033 - 1059 = \text{EFDA}$$

PC relative is no longer applicable, so we try to use BASE relative addressing mode.

### Immediate Addressing Mode

In this mode no memory reference is involved. If immediate mode is used the target address is the operand itself.

55	0020	LDA	#3	
				Immediate operand
		TA = $(0003)_{16}$		
		op    n i    x b p e      disp		
		000000    0 1    0 0 0 0		003 $\Rightarrow$ 010003
133	103C	+LDT	#4096	
				Extended instruction format
		TA = $(01000)_{16}$		
		op    n i    x b p e      disp(20 bits)		
		011101    0 1    0 0 0 1		01000 $\Rightarrow$ 75101000

If the symbol is referred in the instruction as the immediate operand then it is immediate with PC-relative mode as shown in the example below:

12	0003	LDB	#LENGTH	
		LENGTH is at address 0033		
		TA = (PC) + disp $\Rightarrow$ disp = 0033 - 0006 = $(002D)_{16}$		
		op    n i    x b p e      disp		
		011010    0 1    0 0 1 0		02D $\Rightarrow$ 69202D

## Indirect and PC-relative mode:

In this type of instruction the symbol used in the instruction is the address of the location which contains the address of the operand. The address of this is found using PC-relative addressing mode. For example:

70	002A	J	@RETADR
		:	:
95	0030 RETADR	RESW	1
RETADR is at address 0030			
TA = (PC) + disp $\Rightarrow$ disp = 0030 - 002D = (0003) <sub>16</sub>			
op	n i	x b p e	disp
001111	1 0	0 0 1 0	003 $\Rightarrow$ 3E2003

The instruction jumps the control to the address location RETADR which in turn has the address of the operand. If address of RETADR is 0030, the target address is then 0003 as calculated above.

## Program Relocation

Sometimes it is required to load and run several programs at the same time. The system must be able to load these programs wherever there is place in the memory. Therefore the exact starting is not known until the load time.

### Absolute Program

In this the address is mentioned during assembling itself. This is called *Absolute Assembly*.

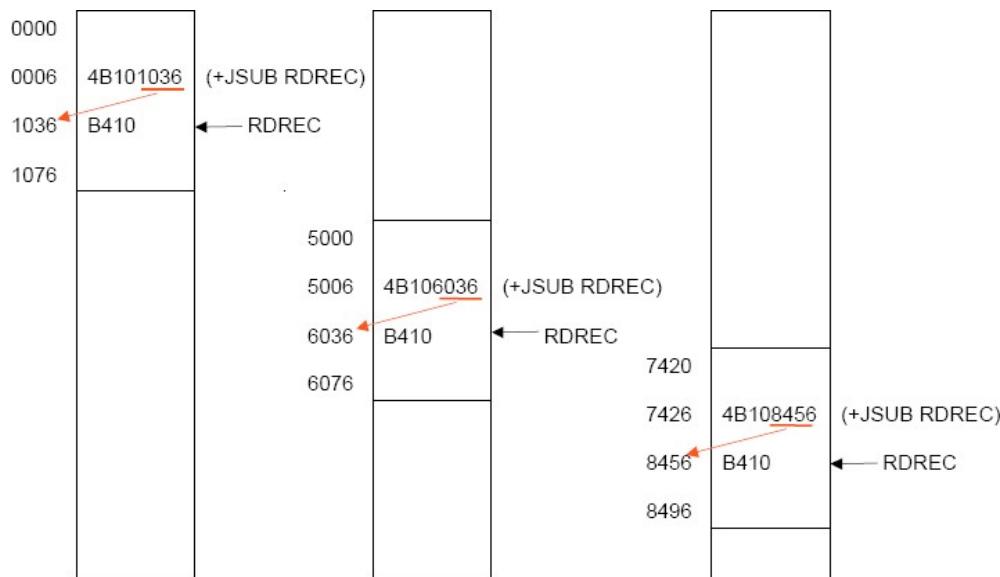
Consider the instruction:

55 101B LDA THREE 00102D

- This statement says that the register A is loaded with the value stored at location 102D. Suppose it is decided to load and execute the program at location 2000 instead of location 1000.
- Then at address 102D the required value which needs to be loaded in the register A is no more available. The address also gets changed relative to the displacement

of the program. Hence we need to make some changes in the address portion of the instruction so that we can load and execute the program at location 2000.

- Apart from the instruction which will undergo a change in their operand address value as the program load address changes. There exist some parts in the program which will remain same regardless of where the program is being loaded.
- Since assembler will not know actual location where the program will get loaded, it cannot make the necessary changes in the addresses used in the program. However, the assembler identifies for the loader those parts of the program which need modification.
- An object program that has the information necessary to perform this kind of modification is called the relocatable program.



- The above diagram shows the concept of relocation. Initially the program is loaded at location 0000. The instruction JSUB is loaded at location 0006.
- The address field of this instruction contains 01036, which is the address of the instruction labeled RDREC. The second figure shows that if the program is to be loaded at new location 5000.

- The address of the instruction JSUB gets modified to new location 6036. Likewise the third figure shows that if the program is relocated at location 7420, the JSUB instruction would need to be changed to 4B108456 that correspond to the new address of RDREC.
- The only part of the program that require modification at load time are those that specify direct addresses. The rest of the instructions need not be modified. The instructions which doesn't require modification are the ones that is not a memory address (immediate addressing) and PC-relative, Base-relative instructions.
- From the object program, it is not possible to distinguish the address and constant. The assembler must keep some information to tell the loader. The object program that contains the modification record is called a relocatable program.
- For an address label, its address is assigned relative to the start of the program (START 0). The assembler produces a *Modification record* to store the starting location and the length of the address field to be modified. The command for the loader must also be a part of the object program. The Modification has the following format:

SVIT

### Modification record

Col. 1            M

Col. 2-7         Starting location of the address field to be modified, relative to the beginning of the program (Hex)

Col. 8-9         Length of the address field to be modified, in half-bytes (Hex)

One modification record is created for each address to be modified. The length is stored in half-bytes (4 bits). The starting location is the location of the byte containing the leftmost bits of the address field to be modified. If the field contains an odd number of half-bytes, the starting location begins in the middle of the first byte.

```

HCOPY    000000001077
T0000001D17202D69202D4B1010360320262900003320074B10105D3F2FEC032010
T00001D130F20160100030F200D4B10105D3E2003454F46
T0010361DB410B400844075101000E32019332FFADB2013A00433200857C003B850
T0010531D3B2F6A1340004F0000F1B410774000E32011332FFA53C003DF2008B850
T001070073B2F6F4F000005
M00000705
M00001405
M00002705
E000000

```

5 half-bytes

In the above object code the red boxes indicate the addresses that need modifications. The object code lines at the end are the descriptions of the modification records for those instructions which need change if relocation occurs. M00000705 is the modification suggested for the statement at location 0007 and requires modification 5-half bytes. Similarly the remaining instructions indicate.

SVIT

### Machine-Independent features:

These are the features which do not depend on the architecture of the machine. These are:

- Literals
- Expressions
- Program blocks
- Control sections

#### Literals:

A literal is defined with a prefix = followed by a specification of the literal value.

Example:

```
45    001A ENDFIL      LDA  =C'EOF'      032010
-
-
93          LTORG
          002D  *           =C'EOF'      454F46
```

The example above shows a 3-byte operand whose value is a character string EOF. The object code for the instruction is also mentioned. It shows the relative displacement value of the location where this value is stored. In the example the value is at location (002D) and hence the displacement value is (010). As another example the given statement below shows a 1-byte literal with the hexadecimal value ‘05’.

```
215   1062 WLOOP      TD    =X'05'      E32011
```

It is important to understand the difference between a constant defined as a literal and a constant defined as an immediate operand. In case of literals the assembler generates the specified value as a constant at some other memory location In immediate mode the operand value is assembled as part of the instruction itself. Example

55	0020	LDA #03	010003
----	------	---------	--------

All the literal operands used in a program are gathered together into one or more *literal pools*. This is usually placed at the end of the program. The assembly listing of a program containing literals usually includes a listing of this literal pool, which shows the assigned addresses and the generated data values. In some cases it is placed at some other location in the object program. An assembler directive LTORG is used. Whenever the LTORG is encountered, it creates a literal pool that contains all the literal operands used since the beginning of the program. The literal pool definition is done after LTORG is encountered. It is better to place the literals close to the instructions.

A literal table is created for the literals which are used in the program. The literal table contains the *literal name, operand value and length*. The literal table is usually created as a hash table on the literal name.

Implementation of Literals:

### During Pass-1:

The literal encountered is searched in the literal table. If the literal already exists, no action is taken; if it is not present, the literal is added to the LITTAB and for the address value it waits till it encounters LTORG for literal definition. When Pass 1 encounters a LTORG statement or the end of the program, the assembler makes a scan of the literal table. At this time each literal currently in the table is assigned an address. As addresses are assigned, the location counter is updated to reflect the number of bytes occupied by each literal.

## During Pass-2:

The assembler searches the LITTAB for each literal encountered in the instruction and replaces it with its equivalent value as if these values are generated by BYTE or WORD. If a literal represents an address in the program, the assembler must generate a modification relocation for, if it all it gets affected due to relocation. The following figure shows the difference between the SYMTAB and LITTAB

SYMTAB	Name	Value
	COPY	0
	FIRST	0
	CLOOP	6
	ENDFIL	1A
	RETADR	30
	LENGTH	33
	BUFFER	36
	BUFEND	1036
	MAXLEN	1000
	RDREC	1036
	RLOOP	1040
	EXIT	1056
	INPUT	105C
	WRBC	105D
	WLOOP	1062

LITTAB	Literal	Hex Value	Length	Address
	C'EOF'	454F46	3	002D
	X'05'	05	1	1076

## Symbol-Defining Statements:

- **EQU Statement:**



Most assemblers provide an assembler directive that allows the programmer to define symbols and specify their values. The directive used for this **EQU** (Equate). The general form of the statement is

Symbol                  EQU                  value

This statement defines the given symbol (i.e., entering in the SYMTAB) and assigning to it the value specified. The value can be a constant or an expression involving constants and any other symbol which is already defined. One common usage is to define symbolic names that can be used to improve readability in place of numeric values. For example

+LDT                  #4096

This loads the register T with immediate value 4096, this does not clearly what exactly this value indicates. If a statement is included as:

```
MAXLEN    EQU      4096 and then  
           +LDT      #MAXLEN
```

Then it clearly indicates that the value of MAXLEN is some maximum length value. When the assembler encounters EQU statement, it enters the symbol MAXLEN along with its value in the symbol table. During LDT the assembler searches the SYMTAB for its entry and its equivalent value as the operand in the instruction. The object code generated is the same for both the options discussed, but is easier to understand. If the maximum length is changed from 4096 to 1024, it is difficult to change if it is mentioned as an immediate value wherever required in the instructions. We have to scan the whole program and make changes wherever 4096 is used. If we mention this value in the instruction through the symbol defined by EQU, we may not have to search the whole program but change only the value of MAXLENGTH in the EQU statement (only once).

Another common usage of EQU statement is for defining values for the general-purpose registers. The assembler can use the mnemonics for register usage like a-register A , X – index register and so on. But there are some instructions which requires numbers in place of names in the instructions. For example in the instruction RMO 0,1 instead of RMO A,X. The programmer can assign the numerical values to these registers using EQU directive.

```
A        EQU      0  
X        EQU      1 and so on
```

These statements will cause the symbols A, X, L... to be entered into the symbol table with their respective values. An instruction RMO A, X would then be allowed. As another usage if in a machine that has many general purpose registers named as R1, R2,..., some may be used as base register, some may be used as accumulator. Their usage may change from one program to another. In this case we can define these requirement using EQU statements.

```
BASE    EQU      R1
```

INDEX	EQU	R2
COUNT	EQU	R3

One restriction with the usage of EQU is whatever symbol occurs in the right hand side of the EQU should be predefined. For example, the following statement is not valid:

BETA	EQU	ALPHA
ALPHA	RESW	1

As the symbol ALPHA is assigned to BETA before it is defined. The value of ALPHA is not known.

- **ORG Statement:**

This directive can be used to indirectly assign values to the symbols. The directive is usually called ORG (for origin). Its general format is:

ORG            value

Where value is a constant or an expression involving constants and previously defined symbols. When this statement is encountered during assembly of a program, the assembler resets its location counter (LOCCTR) to the specified value. Since the values of symbols used as labels are taken from LOCCTR, the ORG statement will affect the values of all labels defined until the next ORG is encountered. ORG is used to control assignment storage in the object program. Sometimes altering the values may result in incorrect assembly.

ORG can be useful in label definition. Suppose we need to define a symbol table with the following structure:

SYMBOL	6 Bytes
VALUE	3 Bytes
FLAG	2 Bytes

The table looks like the one given below.

STAB (100 entries)	SYMBOL	VALUE	FLAGS
⋮	⋮	⋮	⋮

The symbol field contains a 6-byte user-defined symbol; VALUE is a one-word representation of the value assigned to the symbol; FLAG is a 2-byte field specifies symbol type and other information. The space for the ttable can be reserved by the statement:

```
STAB      RESB      1100
```

If we want to refer to the entries of the table using indexed addressing, place the offset value of the desired entry from the beginning of the table in the index register. To refer to the fields SYMBOL, VALUE, and FLAGS individually, we need to assign the values first as shown below:

SYMBOL	EQU	STAB
VALUE	EQU	STAB+6
FLAGS	EQU	STAB+9

To retrieve the VALUE field from the table indicated by register X, we can write a statement:

```
LDA      VALUE, X
```

The same thing can also be done using ORG statement in the following way:

	STAB	RESB	1100
		ORG	STAB
	SYMBOL	RESB	6
	VALUE	RESW	1
	FLAG	RESB	2
		ORG	STAB+1100

The first statement allocates 1100 bytes of memory assigned to label STAB. In the second statement the ORG statement initializes the location counter to the value of STAB. Now the LOCCTR points to STAB. The next three lines assign appropriate memory storage to each of SYMBOL, VALUE and FLAG symbols. The last ORG statement reinitializes the LOCCTR to a new value after skipping the required number of memory for the table STAB (i.e., STAB+1100).

While using ORG, the symbol occurring in the statement should be predefined as is required in EQU statement. For example for the sequence of statements below:

	ORG	ALPHA
BYTE1	RESB	1
BYTE2	RESB	1
BYTE3	RESB	1
	ORG	
ALPHA	RESB	1

The sequence could not be processed as the symbol used to assign the new location counter value is not defined. In first pass, as the assembler would not know what value to assign to ALPHA, the other symbol in the next lines also could not be defined in the symbol table. This is a kind of problem of the forward reference.

## Expressions:

Assemblers also allow use of expressions in place of operands in the instruction. Each such expression must be evaluated to generate a single operand value or address. Assemblers generally arithmetic expressions formed according to the normal rules using arithmetic operators +, - \*, /. Division is usually defined to produce an integer result. Individual terms may be constants, user-defined symbols, or special terms. The only special term used is \* (the current value of location counter) which indicates the value of the next unassigned memory location. Thus the statement

```
BUFFEND EQU *
```

Assigns a value to BUFFEND, which is the address of the next byte following the buffer area. Some values in the object program are relative to the beginning of the program and some are absolute (independent of the program location, like constants). Hence, expressions are classified as either absolute expression or relative expressions depending on the type of value they produce.

- **Absolute Expressions:** The expression that uses only absolute terms is absolute expression. Absolute expression ~~may~~ contain relative term provided the relative terms occur in pairs with opposite signs for each pair. Example:

```
MAXLEN EQU BUFEND-BUFFER
```

In the above instruction the difference in the expression gives a value that does not depend on the location of the program and hence gives an absolute immaterial of the relocation of the program. The expression can have only absolute terms. Example:

```
MAXLEN EQU 1000
```

- **Relative Expressions:** All the relative terms except one can be paired as described in “absolute”. The remaining unpaired relative term must have a positive sign. Example:

```
STAB EQU OPTAB + (BUFEND - BUFFER)
```

- **Handling the type of expressions:** to find the type of expression, we must keep track the type of symbols used. This can be achieved by defining the type in the symbol table against each of the symbol as shown in the table below:

Symbol	Type	Value
RETADR	R	0030
BUFFER	R	0036
BUFEND	R	1036
MAXLEN	A	1000

## Program Blocks:

Program blocks allow the generated machine instructions and data to appear in the object program in a different order by Separating blocks for storing code, data, stack, and larger data block.

### *Assembler Directive USE:*

USE [blockname]

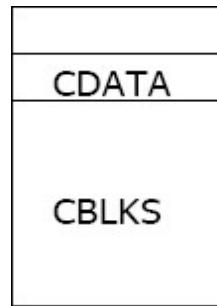
At the beginning, statements are assumed to be part of the *unnamed* (default) block. If no USE statements are included, the entire program belongs to this single block. Each program block may actually contain several separate segments of the source program. Assemblers rearrange these segments to gather together the pieces of each block and assign address. Separate the program into blocks in a particular order. Large buffer area is moved to the end of the object program. *Program readability is better* if data areas are placed in the source program close to the statements that reference them.

In the example below three blocks are used :

Default: executable instructions

CDATA: all data areas that are less in length

CBLKS: all data areas that consists of larger blocks of memory



## Example Code

(default) block		Block number				
0000	0	COPY	START	0		
0000	0	FIRST	STL	RETADR	172063	
0003	0	CLOOP	JSUB	RDREC	4B2021	
0006	0		LDA	LENGTH	032060	
0009	0		COMP	#0	290000	
000C	0		JEQ	ENDFIL	332006	
000F	0		JSUB	WRREC	4B203B	
0012	0		J	CLOOP	3F2FEE	
0015	0	ENDFIL	LDA	=C'EOF'	032055	
0018	0		STA	BUFFER	0F2056	
001B	0		LDA	#3	010003	
001E	0		STA	LENGTH	0F2048	
0021	0		JSUB	WRREC	4B2029	
0024	0		J	@RETADR	3E203F	
0000	1		USE	CDATA	CDATA block	
0000	1	RETADR	RESW	1		
0003	1	LENGTH	RESW	1		
0000	2		USE	CBLKS	CBLKS block	
0000	2	BUFFER	RESB	4096		
1000	2	BUFEND	EQU	*		
1000	2	MAXLEN	EQU	BUFEND-BUFFER		

				(default) block
0027	0	RDREC	USE	
0027	0		CLEAR	X B410
0029	0		CLEAR	A B400
002B	0		CLEAR	S B440
002D	0		+LDT	#MAXLEN 75101000
0031	0	RLOOP	TD	INPUT E32038
0034	0		JEQ	RLOOP 332FFA
0037	0		RD	INPUT DB2032
003A	0		COMPR	A,S A004
003C	0		JEQ	EXIT 332008
003F	0		STCH	BUFFER,X 57A02F
0042	0		TIXR	T B850
0044	0		JLT	RLOOP 3B2FEA
0047	0	EXIT	STX	LENGTH 13201F
004A	0		RSUB	4F0000
0006	1		USE	
0006	1	INPUT	BYTE	CDATA F1

				(default) block
004D	0	WRREC	USE	
004D	0		CLEAR	X B410
004F	0		LDT	LENGTH 772017
0052	0	WLOOP	TD	=X'05' E3201B
0055	0		JEQ	WLOOP 332FFA
0058	0		LDCH	BUFFER,X 53A016
005B	0		WD	=X'05' DF2012
005E	0		TIXR	T B850
0060	0		JLT	WLOOP 3B2FEF
0063	0		RSUB	4F0000
0007	1		USE	CDATA
0007	1	*	LTORG	← CDATA block 454F46
000A	1	*	=C'EOF	05
			=X'05'	
			END	FIRST

## Arranging code into program blocks:

### Pass 1

- A separate location counter for each program block is maintained.
- Save and restore LOCCTR when switching between blocks.
- At the beginning of a block, LOCCTR is set to 0.
- Assign each label an address relative to the start of the block.

- Store the block name or number in the SYMTAB along with the assigned relative address of the label
- Indicate the block length as the latest value of LOCCTR for each block at the end of Pass1
- Assign to each block a starting address in the object program by concatenating the program blocks in a particular order

### Pass 2

- Calculate the address for each symbol relative to the start of the object program by adding
  - The location of the symbol relative to the start of its block
  - The starting address of this block

### Control Sections:

A *control section* is a part of the program that maintains its identity after assembly; each control section can be loaded and relocated independently of the others. Different control sections are most often used for subroutines or other logical subdivisions. The programmer can assemble, load, and manipulate each of these control sections separately.

Because of this, there should be some means for linking control sections together. For example, instructions in one control section may refer to the data or instructions of other control sections. Since control sections are independently loaded and relocated, the assembler is unable to process these references in the usual way. Such references between different control sections are called *external references*.

The assembler generates the information about each of the external references that will allow the loader to perform the required linking. When a program is written using multiple control sections, the beginning of each of the control section is indicated by an assembler directive

- assembler directive: **CSECT**

## The syntax

**secname CSECT**

- separate location counter for each control section

Control sections differ from program blocks in that they are handled separately by the assembler. Symbols that are defined in one control section may not be used directly another control section; they must be identified as external reference for the loader to handle. The external references are indicated by two assembler directives:

- **EXTDEF** (external Definition):

It is the statement in a control section, names symbols that are defined in this section but may be used by other control sections. Control section names do not need to be named in the EXTREF as they are automatically considered as external symbols.

- **EXTREF** (external Reference):

It names symbols that are used in this section but are defined in some other control section.

The order in which these symbols are listed is not significant. The assembler must include proper information about the external references in the object program that will cause the loader to insert the proper value where they are required.

	<b>Implicitly defined as an external symbol first control section</b>		
COPY	START	0	COPY FILE FROM INPUT TO OUTPUT
	EXTDEF	BUFFER,BUFEND,LENGTH	
	EXTREF	RDREC,WRREC	
FIRST	STL	RETADR	SAVE RETURN ADDRESS
CLOOP	+JSUB	RDREC	READ INPUT RECORD
	LDA	LENGTH	TEST FOR EOF (LENGTH=0)
	COMP	#0	
	JEQ	ENDFIL	
	+JSUB	WRREC	
	J	CLOOP	EXIT IF EOF FOUND
ENDFIL	LDA	=C'EOF'	WRITE OUTPUT RECORD
	STA	BUFFER	LOOP
	LDA	#3	INSERT END OF FILE MARKER
	STA	LENGTH	
	+JSUB	WRREC	
	J	@RETADR	SET LENGTH = 3
RETADR	RESW	1	WRITE EOF
LENGTH	RESW	1	RETURN TO CALLER
	LTORG		LENGTH OF RECORD
BUFFER	RESB	4096	4096-BYTE BUFFER AREA
BUFEND	EQU	*	
MAXLEN	EQU	BUFFEND-BUFFER	
	<b>Implicitly defined as an external symbol second control section</b>		
RDREC	CSECT		
:	SUBROUTINE TO READ RECORD INTO BUFFER		
	EXTREF	BUFFER,LENGTH,BUFFEND	
	CLEAR	X	CLEAR LOOP COUNTER
	CLEAR	A	CLEAR A TO ZERO
	CLEAR	S	CLEAR S TO ZERO
	LDT	MAXLEN	
RLOOP	TD	INPUT	TEST INPUT DEVICE
	JEQ	RLOOP	LOOP UNTIL READY
	RD	INPUT	READ CHARACTER INTO REGISTER A
	COMPR	A,S	TEST FOR END OF RECORD (X'00')
	JEQ	EXIT	EXIT LOOP IF EOR
	+STCH	BUFFER,X	STORE CHARACTER IN BUFFER
	TIXR	T	LOOP UNLESS MAX LENGTH HAS
	JLT	RLOOP	BEEN REACHED
EXIT	+STX	LENGTH	SAVE RECORD LENGTH
	RSUB		RETURN TO CALLER
INPUT	BYTE	X'F1'	CODE FOR INPUT DEVICE
MAXLEN	WORD	BUFFEND-BUFFER	

Implicitly defined as an external symbol  
third control section

---

```

WRREC      CSECT
:          SUBROUTINE TO WRITE RECORD FROM BUFFER
          EXTREF LENGTH,BUFFER
          CLEAR   X           CLEAR LOOP COUNTER
          +LDT    LENGTH
          TD      =X'05'      TEST OUTPUT DEVICE
          JEQ    WLOOP       LOOP UNTIL READY
          +LDCH   BUFFER,X    GET CHARACTER FROM BUFFER
          WD      =X'05'      WRITE CHARACTER
          TIXR    T           LOOP UNTIL ALL CHARACTERS HAVE
          JLT     WLOOP       BEEN WRITTEN
          RSUB
          END    FIRST        RETURN TO CALLER

```

## Handling External Reference

### Case 1

15	0003	CLOOP	+JSUB RDREC	4B100000
----	------	-------	-------------	----------

- The operand RDREC is an external reference.
  - The assembler has no idea where RDREC is
  - inserts an address of zero
  - can only use extended formatto provide enough room (that is, relative addressing for external reference is invalid)
- The assembler generates information for each external reference that will allow the loader to perform the required linking.

### Case 2

190	0028	MAXLEN	WORD	BUFEND-BUFFER	000000
-----	------	--------	------	---------------	--------

- There are two external references in the expression, BUFEND and BUFFER.
- The assembler inserts a value of zero
- passes information to the loader

- Add to this data area the address of BUFEND
- Subtract from this data area the address of BUFFER

### Case 3

On line 107, BUFEND and BUFFER are defined in the same control section and the expression can be calculated immediately.

```
107    1000 MAXLEN    EQU      BUFEND-BUFFER
```

### Object Code for the example program:

0000	COPY	START	0	
		EXTDEF	BUFFER,BUFFEND,LENGTH	
		EXTREF	RDREC,WRREC	
0000	FIRST	STL	RETADR	172027
0003	CLOOP	+JSUB	RDREC	4B100000
0007		LDA	LENGTH	032023
000A		COMP	#0	290000
000D		JEQ	ENDFIL	332007
0010		+JSUB	WRREC	4B100000
0014		J	CLOOP	3F2FEC
0017	ENDFIL	LDA	=C'EOF'	032016
001A		STA	BUFFER	0F2016
001D		LDA	#3	010003
0020		STA	LENGTH	0F200A
0023		+JSUB	WRREC	4B100000
0027		J	@RETADR	3E2000
002A	RETADR	RESW	1	
002D	LENGTH	RESW	1	
		LTORG		
0030	*	=C'EOF'		454F46
0033	BUFFER	RESB	4096	
1033	BUFEND	EQU	*	
1000	MAXLEN	EQU	BUFEND-BUFFER	

Case 1

0000	RDREC	CSECT	
:		SUBROUTINE TO READ RECORD INTO BUFFER	
0000		EXTREF BUFFER,LENGTH,BUFEND	
0000		CLEAR X	B410
0002		CLEAR A	B400
0004		CLEAR S	B440
0006		LDT MAXLEN	77201F
0009	RLOOP	TD INPUT	E3201B
000C		JEQ RLOOP	332FFA
000F		RD INPUT	DB2015
0012		COMPR A,S	A004
0014		JEQ EXIT	332009
0017		+STCH BUFFER,X	57900000
001B		TIXR T	B850
001D		JLT RLOOP	3B2FE9
0020	EXIT	+STX LENGTH	13100000
0024		RSUB	4F0000
0027	INPUT	BYTE X'F1'	F1
0028	MAXLEN	WORD BUFFEND-BUFFER	000000

Case 2

0000	WRREC	CSECT	
:		SUBROUTINE TO WRITE RECORD FROM BUFFER	
0000		EXTREF LENGTH,BUFFER	
0000		CLEAR X	B410
0002		+LDT LENGTH	77100000
0006	WLOOP	TD =X'05'	E32012
0009		JEQ WLOOP	332FFA
000C		+LDCH BUFFER,X	53900000
0010		WD =X'05'	DF2008
0013		TIXR T	B850
0015		JLT WLOOP	3B2FEE
0018		RSUB	4F0000
		END FIRST	
001B	*	=X'05'	05

The assembler must also include information in the object program that will cause the loader to insert the proper value where they are required. The assembler maintains two new record in the object code and a changed version of modification record.

## Define record (EXTDEF)

- Col. 1 D
- Col. 2-7 Name of external symbol defined in this control section
- Col. 8-13 Relative address within this control section (hexadecimal)
- Col.14-73 Repeat information in Col. 2-13 for other external symbols

## Refer record (EXTREF)

- Col. 1 R
- Col. 2-7 Name of external symbol referred to in this control section
- Col. 8-73 Name of other external reference symbols

## Modification record

- Col. 1 M
- Col. 2-7 Starting address of the field to be modified (hexadecimal)
- Col. 8-9 Length of the field to be modified, in half-bytes (hexadecimal)
- Col.11-16 External symbol whose value is to be added to or subtracted from the indicated field

A define record gives information about the external symbols that are defined in this control section, i.e., symbols named by EXTDEF. A refer record lists the symbols that are used as external references by the control section, i.e., symbols named by EXTREF.

The new items in the modification record specify the modification to be performed: adding or subtracting the value of some external symbol. The symbol used for modification may be defined either in this control section or in another section.

The object program is shown below. There is a separate object program for each of the control sections. In the *Define Record* and *refer record* the symbols named in EXTDEF and EXTREF are included.

In the case of *Define*, the record also indicates the relative address of each external symbol within the control section. For EXTREF symbols, no address information is available. These symbols are simply named in the *Refer record*.

## COPY

```
HCOPY .000000001033
DBUFFER000033BUFEND001033LENGTH00002D
RRDREC WRREC
T0000001D1720274B1000000320232900003320074B1000003F2FE0320160F2016
T00001D0D0100030F200A4B1000003E2000
T00003003454F46
M00000405+RDREC
M00001105+WRREC
M00002405+WRREC
E000000
```

## RDREC

```
HRDREC 00000000002B
RBUFFERLENGTHBUFEND
T0000001DB410B400B44077201FE3201B332FFADB2015A00433200957900000B850
T00001D0E3B2FE9131000004F0000F1000000
M00001805+BUFFER
M00002105+LENGTH
M00002806+BUFEND
M00002806-BUFFER
E
```

} BUFEND - BUFFER

## WRREC

```
HWRREC 00000000001C
RLENGTHBUFFER
T0000001CB41077100000E3201232FFA53900000DF2008B8503B2FEE4F000005
M00000305+LENGTH
M00000D05+BUFFER
E
```

## Handling Expressions in Multiple Control Sections:

The existence of multiple control sections that can be relocated independently of one another makes the handling of expressions complicated. It is required that in an expression that all the relative terms be paired (for absolute expression), or that all except one be paired (for relative expressions).

When it comes in a program having multiple control sections then we have an extended restriction that:

- Both terms in each pair of an expression must be within the same control section
  - If two terms represent relative locations within the same control section , their difference is an absolute value (regardless of where the control section is located.
    - **Legal:** BUFEND-BUFFER (both are in the same control section)
  - If the terms are located in different control sections, their difference has a value that is unpredictable.
    - **Illegal:** RDREC-COPY (both are of different control section) it is the difference in the load addresses of the two control sections. This value depends on the way run-time storage is allocated; it is unlikely to be of any use.
- **How to enforce this restriction**
  - When an expression involves external references, the assembler cannot determine whether or not the expression is legal.
  - The assembler evaluates all of the terms it can, combines these to form an initial expression value, and generates Modification records.
  - The loader checks the expression for errors and finishes the evaluation.

## ASSEMBLER DESIGN OPTIONS

Here we are discussing

- The structure and logic of one-pass assembler. These assemblers are used when it is necessary or desirable to avoid a second pass over the source program.
- Notion of a multi-pass assembler, an extension of two-pass assembler that allows an assembler to handle forward references during symbol definition.

## One-Pass Assembler

The main problem in designing the assembler using single pass was to resolve forward references. We can avoid to some extent the forward references by:

- Eliminating forward reference to data items, by defining all the storage reservation statements at the beginning of the program rather at the end.
- Unfortunately, forward reference to labels on the instructions cannot be avoided. (forward jumping)
- To provide some provision for handling forward references by prohibiting forward references to data items.

There are two types of one-pass assemblers:

- One that produces object code directly in memory for immediate execution (Load-and-go assemblers).
- The other type produces the usual kind of object code for later execution.

## Load-and-Go Assembler

- Load-and-go assembler generates their object code in memory for immediate execution.
- No object program is written out, no loader is needed.
- It is useful in a system with frequent program development and testing
  - The efficiency of the assembly process is an important consideration.
- Programs are re-assembled nearly every time they are run; efficiency of the assembly process is an important consideration.

Line	Loc	Source statement			Object code
0	1000	COPY	START	1000	
1	1000	EOF	BYTE	C'EOF'	454F46
2	1003	THREE	WORD	3	000003
3	1006	ZERO	WORD	0	000000
4	1009	RETADR	RESW	1	
5	100C	LENGTH	RESW	1	
6	100F	BUFFER	RESB	4096	
9	.				
10	200F	FIRST	STL	RETADR	141009
15	2012	CLOOP	JSUB	RDREC	48203D
20	2015		LDA	LENGTH	00100C
25	2018		COMP	ZERO	281006
30	201B		JEQ	ENDFIL	302024
35	201E		JSUB	WRREC	482062
40	2021		J	CLOOP	302012
45	2024	ENDFIL	LDA	EOF	001000
50	2027		STA	BUFFER	0C100F
55	202A		LDA	THREE	001003
60	202D		STA	LENGTH	0C100C
65	2030		JSUB	WRREC	482062
70	2033		LDL	RETADR	081009
75	2036		RSUB		4C0000
110	.				

SVIT

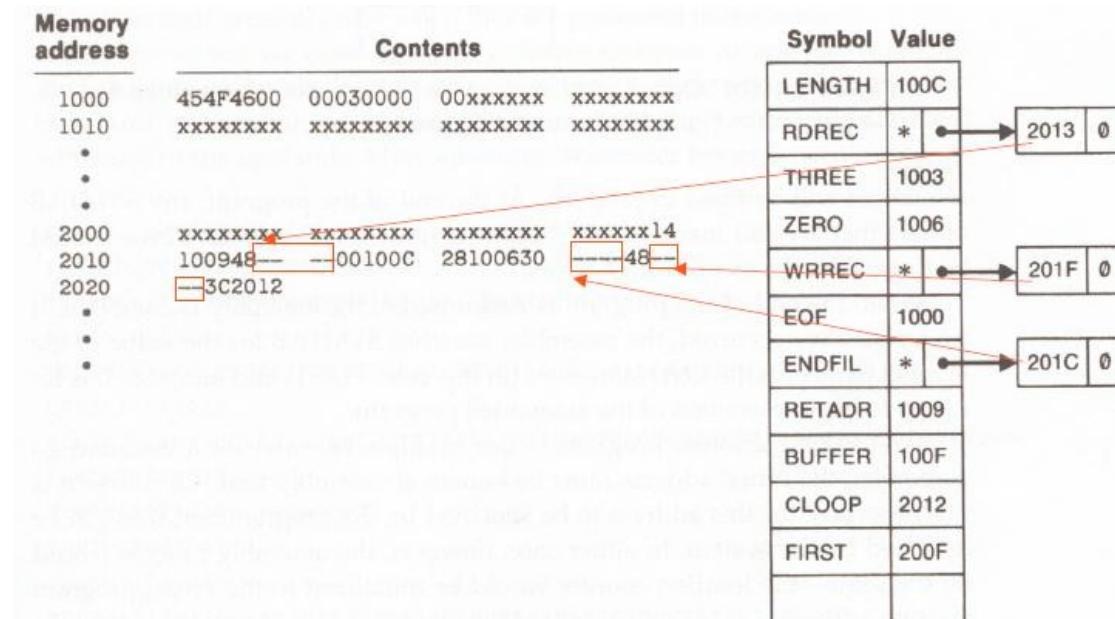
**Forward Reference in One-Pass Assemblers:** In load-and-Go assemblers when a forward reference is encountered :

- Omits the operand address if the symbol has not yet been defined
- Enters this undefined symbol into SYMTAB and indicates that it is undefined
- Adds the address of this operand address to a list of forward references associated with the SYMTAB entry
- When the definition for the symbol is encountered, scans the reference list and inserts the address.
- At the end of the program, reports the error if there are still SYMTAB entries indicated undefined symbols.
- For Load-and-Go assembler
  - Search SYMTAB for the symbol named in the END statement and jumps to this location to begin execution if there is no error

After Scanning line 40 of the program:

40 2021 J CLOOP 302012

The status is that upto this point the symbol RREC is referred once at location 2013, ENDFIL at 201F and WRREC at location 201C. None of these symbols are defined. The figure shows that how the pending definitions along with their addresses are included in the symbol table.



The status after scanning line 160, which has encountered the definition of RDREC and ENDFIL is as given below:

Memory address	Contents				Symbol	Value
1000	454F4600	00030000	00xxxxxx	xxxxxxxx	LENGTH	100C
1010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	RDREC	203D
*					THREE	1003
*					ZERO	1006
*					WRREC	*
2000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxx14		
2010	10094820	3D00100C	28100630	202448		
2020	3C2012	0010000C	100F0010	0306100C	EOF	1000
2030	4B	10094C00	00F10010	00041006	ENDFIL	2024
2040	001006E0	20393020	43D82039	28100630	RETADR	1009
2050	5490	0F			BUFFER	100F
*					CLOOP	2012
*					FIRST	200F
*					MAXLEN	203A
					INPUT	2039
					EXIT	*
					RLOOP	2043

If One-Pass needs to generate object code:

- If the operand contains an undefined symbol, use 0 as the address and write the Text record to the object program.
- Forward references are entered into lists as in the load-and-go assembler.
- When the definition of a symbol is encountered, the assembler generates another Text record with the correct operand address of each entry in the reference list.
- When loaded, the incorrect address 0 will be updated by the latter Text record containing the symbol definition.

Object Code Generated by One-Pass Assembler:

```

HCOPY 00100000107A
T00100009454F46000003000000
T00200F1514100948000000100C2810063000004800003C2012
T00201C022024
T002024190010000C100F0010030C100C480000810094C0000F1001000
T00201302203D
T00203D1E041006001006E02039302043D8203928100630000054900F2C203A382043
T00205002205B
T00205B0710100C4C000005
T00201F022062
T002031022062
T00206218041006E0206130206550900FDC20612C100C3820654C0000
E00200F

```

## **Multi\_Pass Assembler:**

- For a two pass assembler, forward references in symbol definition are not allowed:

ALPHA EQU BETA

BETA EQU DELTA

DELTA RESW 1

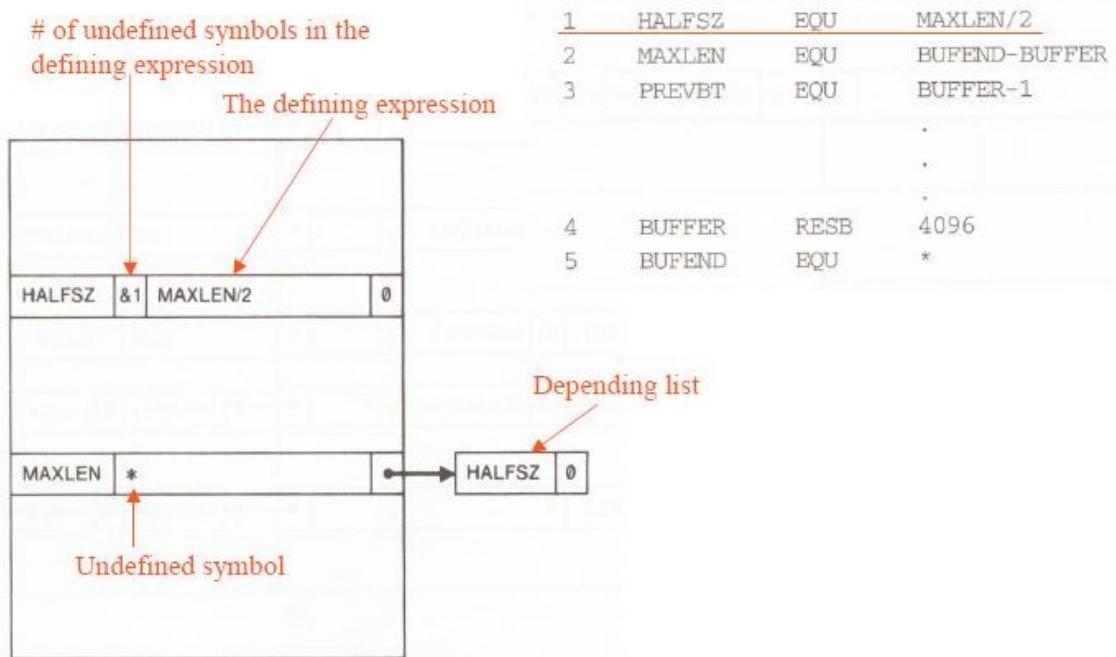
- Symbol definition must be completed in pass 1.
- Prohibiting forward references in symbol definition is not a serious inconvenience.
  - Forward references tend to create difficulty for a person reading the program.

## **Implementation Issues for Modified Two-Pass Assembler:**

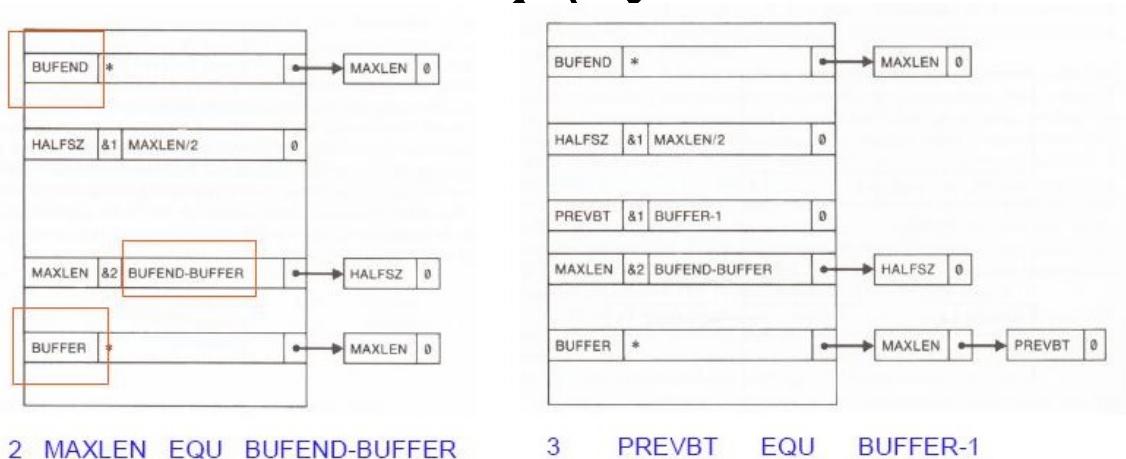
Implementation Isuues when forward referencing is encountered in *Symbol Defining statements* :

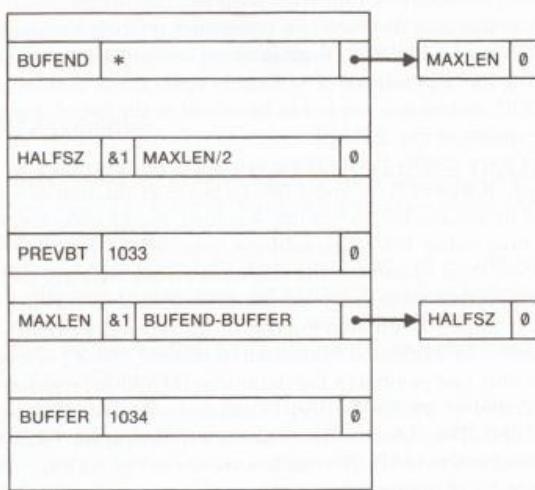
- For a forward reference in symbol definition, we store in the SYMTAB:
  - The symbol name
  - The defining expression
  - The number of undefined symbols in the defining expression
- The undefined symbol (marked with a flag \*) associated with a list of symbols depend on this undefined symbol.
- When a symbol is defined, we can recursively evaluate the symbol expressions depending on the newly defined symbol.

## Multi-Pass Assembler Example Program

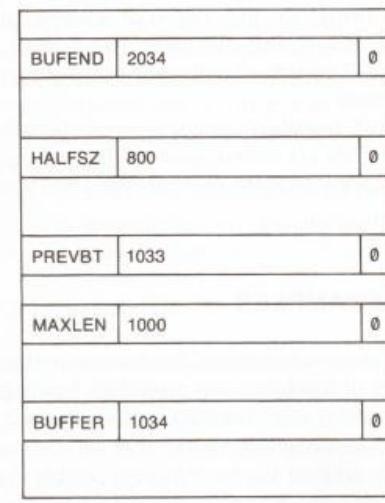


Multi-Pass Assembler : Example for forward reference in Symbol Defining Statements:





4 BUFFER RESB 4096



5 BUFEND EQU \*

SVIT

## BASIC LOADER FUNCTIONS

### Introduction

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

This contains the following three processes, and they are,

- **Loading** - which allocates memory location and brings the object program into memory for execution - (Loader)
- **Linking**- which combines two or more separate object programs and supplies the information needed to allow references between them - (Linker)
- **Relocation** - which modifies the object program so that it can be loaded at an address different from the location originally specified - (Linking Loader)

### Basic Loader Functions:

A loader is a system program that performs the loading function. It brings object program into memory and starts its execution. The role of loader is as shown in the figure 4.1. Translator may be assembler/complier, which generates the object program and later loaded to the memory by the loader for execution. In figure 4.2 the translator is specifically an assembler, which generates the object loaded, which becomes input to the loader. The figure4.3 shows the role of both loader and linker.

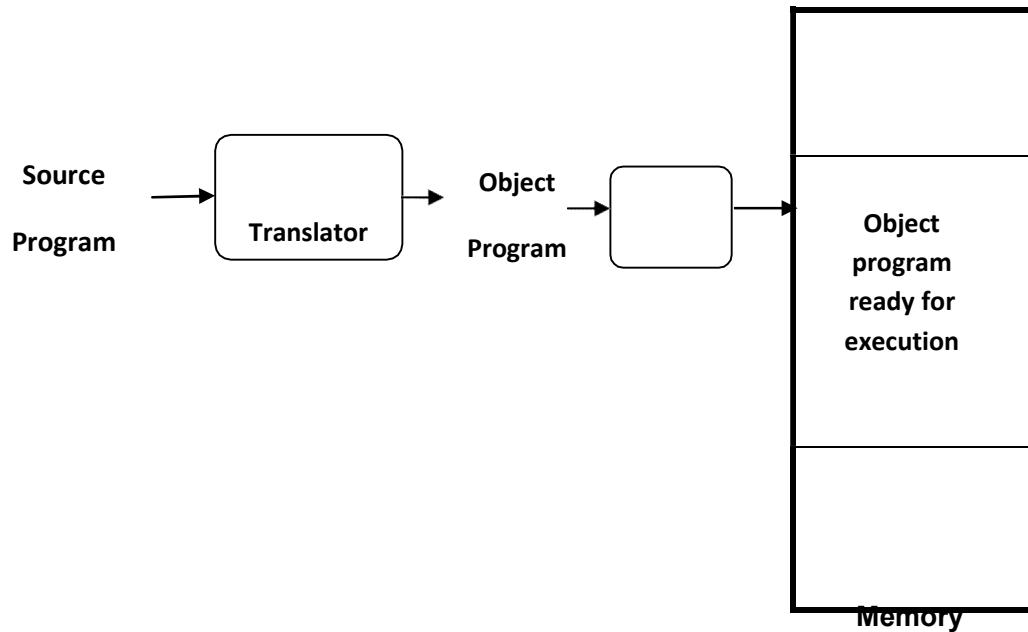


Figure 4.1 : The Role of Loader

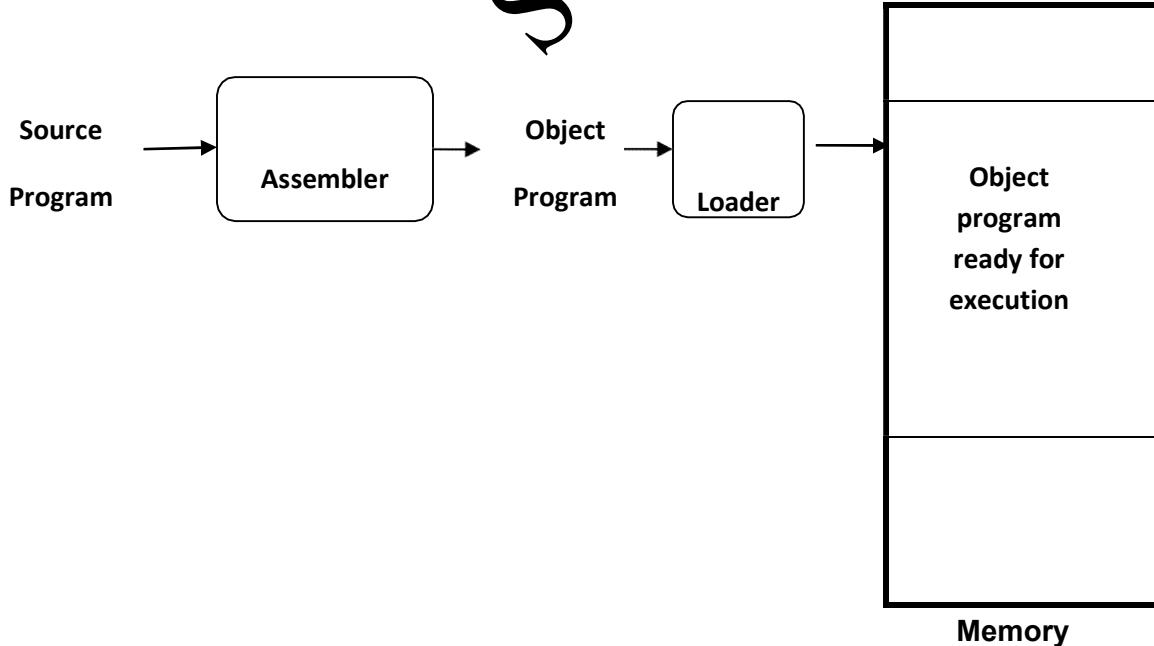


Figure 4.2: The Role of Loader with Assembler

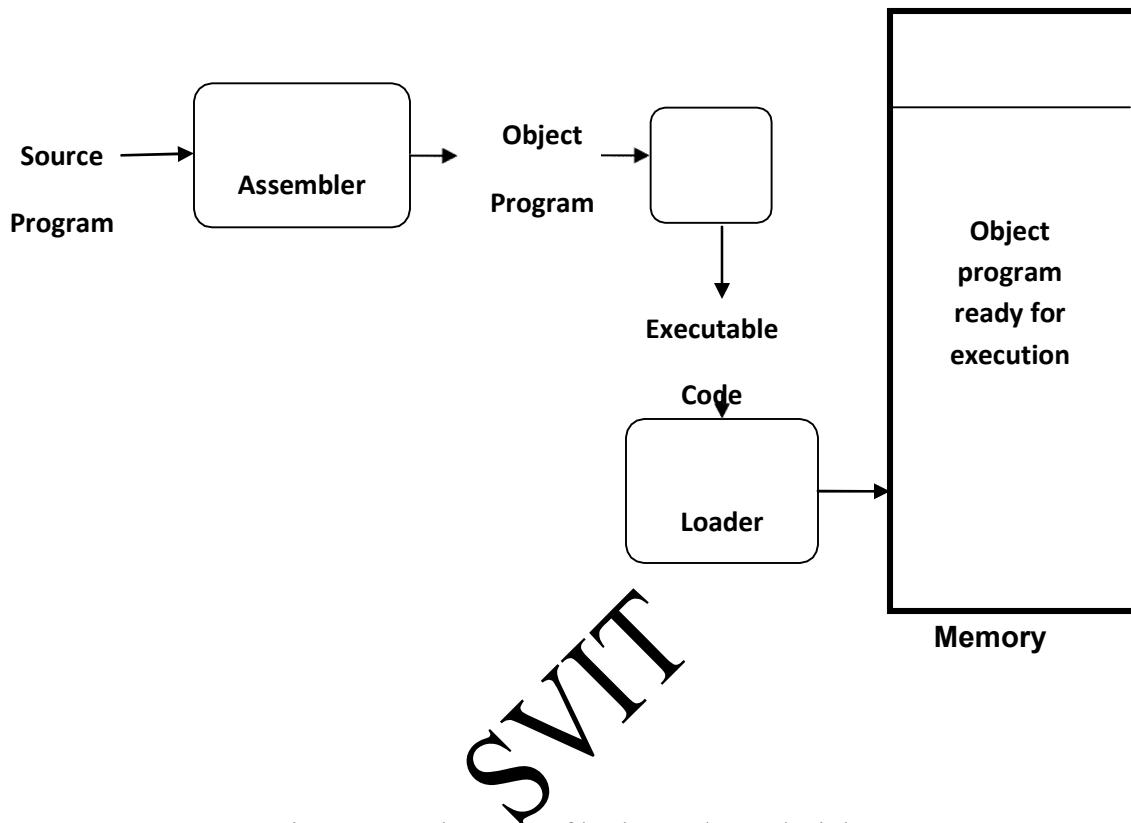


Figure 4.3: The Role of both Loader and Linker

## Type of Loaders

The different types of loaders are, absolute loader, bootstrap loader, relocating loader (relative loader), and, direct linking loader. The following sections discuss the functions and design of all these types of loaders.

### **Design of Absolute Loader:**

The operation of absolute loader is very simple. The object code is loaded to specified locations in the memory. At the end the loader jumps to the specified address to begin execution of the loaded program. The role of absolute loader is as shown in the figure 4.4.

The advantage of absolute loader is simple and efficient. But the disadvantages are, the need for programmer to specify the actual address, and, difficult to use subroutine libraries.

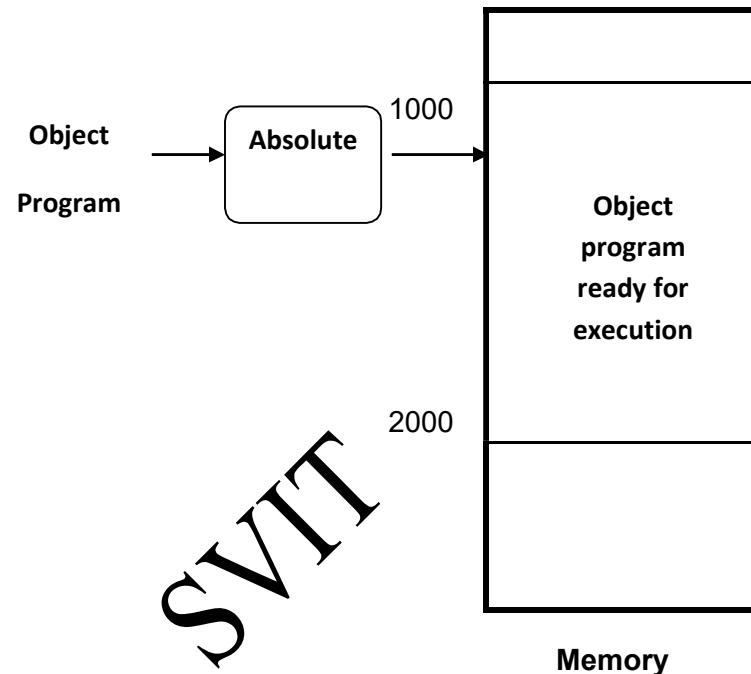


Figure 4.4: The Role of Absolute Loader

The algorithm for this type of loader is given here. The object program and, the object program loaded into memory by the absolute loader are also shown. Each byte of assembled code is given using its hexadecimal representation in character form. Easy to read by human beings. Each byte of object code is stored as a single byte. Most machine store object programs in a binary form, and we must be sure that our file and device conventions do not cause some of the program bytes to be interpreted as control characters.

### Begin

read Header record

verify program name and length

read first Text record

**while** record type is  $\neq$  ‘E’ **do**

**begin**

{if object code is in character form, convert into internal representation}

move object code to specified location in memory

read next object program record

**end**

jump to address specified in End record

**end**

```
BCOPY CC100000107A
T0010001E1410334820390010362810303010154820613C100300102A0C103900102D
T00101E150C10364820610810334C0000454F460C0003000000
T0020391E041030001030E0205030203FD8205D2810303020575490392C205E38203F
T0020571C1010364C0000F1001000041030E020793020645090390C20792C1036
T002073073820644C000005
E001000
```

(a) Object program

<b>Memory address</b>	<b>Contents</b>			
0000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
:	:	:	:	:
0FF0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
1000	14103348	20390010	36281030	30101548
1010	20613C10	0300102A	0C103900	102D0C10
1020	36482061	0810334C	0000454F	46000003
1030	000000xx	xxxxxxxx	xxxxxxxx	xxxxxxxx
:	:	:	:	:
2030	xxxxxxxx	xxxxxxxx	xx041030	001030E0
2040	205D3020	3FD8205D	28103030	20575490
2050	392C205E	38203F10	10364C00	00F10010
2060	00041030	E0207930	20645090	39DC2079
2070	2C103638	20644C00	0005xxxx	xxxxxxxx
2080	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
:	:	:	:	:

(b) Program loaded in memory

### A Simple Bootstrap Loader

When a computer is first turned on or restarted, a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first program to be run by the computer -- usually an operating system. The bootstrap itself begins at address 0. It loads the OS starting address 0x80. No header record or control information, the object code is consecutive bytes of memory.

The algorithm for the bootstrap loader is as follows

**Begin**

X=0x80 (the address of the next memory location to be loaded)

**Loop**

A $\leftarrow$ GETC (and convert it from the ASCII character  
code to the value of the hexadecimal digit)  
save the value in the high-order 4 bits of S

A $\leftarrow$ GETC

combine the value to form one byte A $\leftarrow$  (A+S)  
store the value (in A) to the address in register X

X $\leftarrow$ X+1

**End**

It uses a subroutine GETC, which is

GETC      A $\leftarrow$ read one character

if A=0x04 then jump to 0x80

if A<48 then GETC

A  $\leftarrow$  A-48 (0x30)

if A<10 then return

A  $\leftarrow$  A-7

return

SVIT

# Chapter 1: Lexical Analysis

## What are we studying in this chapter?

- ♦ Compilers
  - Analysis of source program
  - The phases of a compiler
  - Cousins of a compiler
  - The grouping of phases
  - Compiler-construction tools
- ♦ Lexical analysis
  - The role of Lexical Analyzer
  - Input buffering
  - Specifications of tokens
  - Recognition of tokens

- 6 hours

### 1.1 Introduction

We know that the programming languages are used to describe computations to people and to machines. The programming languages are very important since all the software running on all the computers was written in some programming language. But, before we run a program on any computer, it must be translated into a form which can be understood and executed by the computer. For this purpose we use language processors. In the first section, we discuss something about language processors.

#### 1.1.1 Language processors

Now, let us see “What is a language processor?”

**Definition:** A language processor is a program that accepts source language as the input and translates it into another language called target language. The language processors are used for software development without worrying much about the architecture of the machine i.e., the programmers can ignore the machine-dependent details during software development. It performs translating and interpreting a specified programming language.

Now, let us see “What are the various types of language processors?” The various types of language processors are shown below:

- ♦ Preprocessor
- ♦ Compiler
- ♦ Interpreter
- ♦ Assembler
- ♦ Hybrid compiler

## 1.2 □ Lexical Analyzer

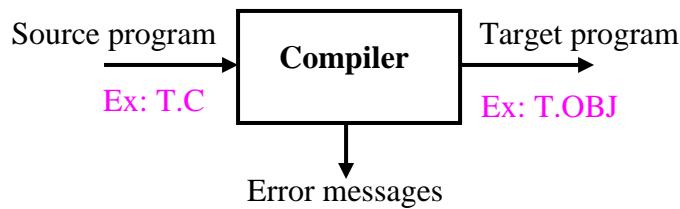
---

Now, let us see “What is a compiler? What are the activities performed by the compiler?”

**Definition:** A compiler is a translator that accepts the source program written in high level language such as C/C++ and converts it into object program or low level language such as assembly language. The various activities done by the compiler are shown below:

- ◆ The syntax errors are identified and appropriate error messages are displayed along with line numbers
- ◆ An optional list file can be generated by the compiler
- ◆ The compiler replaces each executable statement in high level language into one or more machine language instructions
- ◆ Converts HLL into assembly language or object program if the program is syntactically correct.

This can be pictorially represented as shown below:



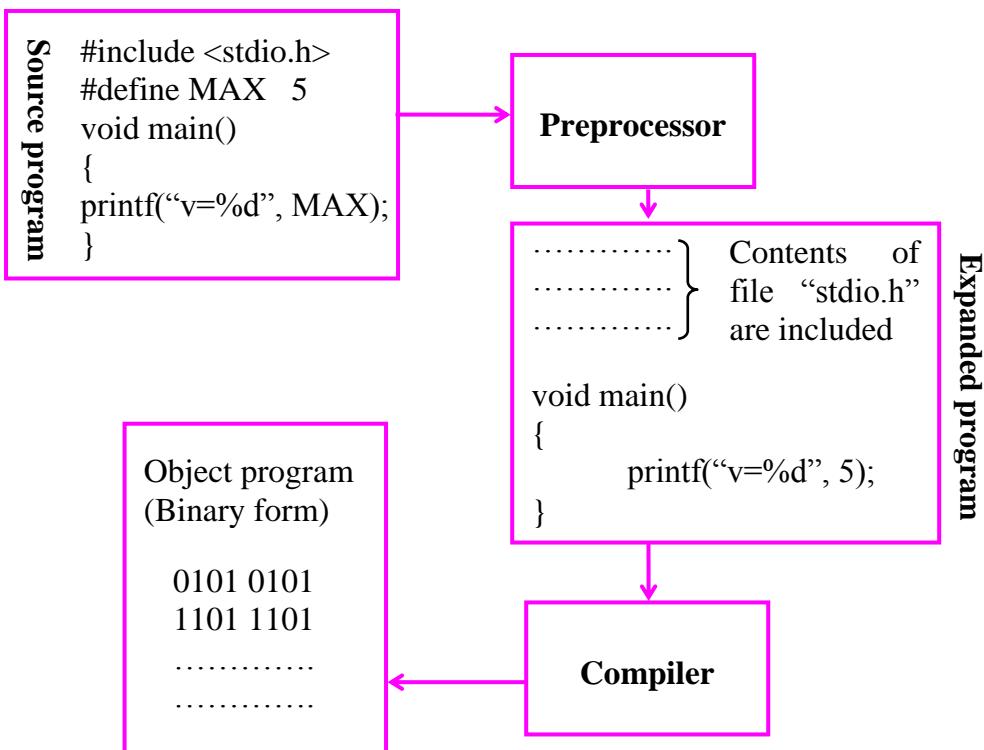
If the target program is an executable machine code, it can be called by the user to process the inputs and produce output which is pictorially represented as shown below:



Now, let us see “What is a preprocessor?”

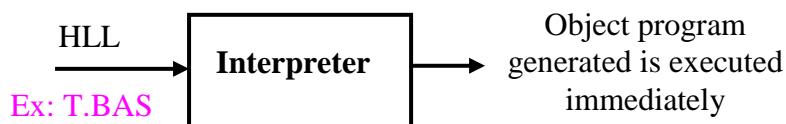
**Definition:** The **preprocessor** is a program that accepts source program as the input and prepares the input program for compilation. A source program is normally divided into modules and will be stored in separate files. All these files are collected by the preprocessor with the help of #include directive. The preprocessor may also expand the shorthands called macros into source language statements with the help of #define directive.

Thus, the expanded version of the source program is input to the compiler. This is pictorially represented as shown below:



Now, let us see “What is an interpreter?”

**Definition:** Interpreter is a translator that accepts the source program written in high level language and converts it into machine code and executes it. The interpreter converts one high level language statement into machine code and then executes immediately. The statements within the loop are translated into machine code each time the control enters into the loop. Whenever an error is encountered, the execution of the program is halted and an error message is displayed. This can be pictorially represented as shown below:



Some of the interpreters are Java virtual machine, BASIC etc.

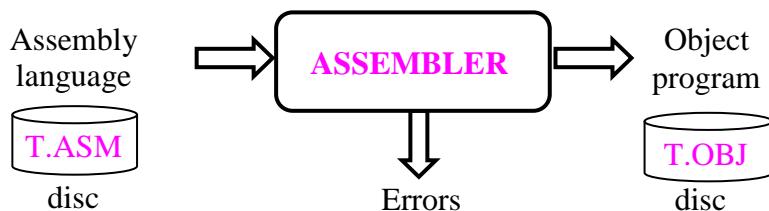
Now, let us see “What are the differences between a compiler and interpreter?” The difference between compiler and interpreter are shown below:

## 1.4 □ Lexical Analyzer

COMPILER	INTERPRETER
◆ Converts the entire source program into machine code. The linker uses the machine code and creates an executable file	◆ The interpreter takes one statement at a time, translates it into machine code and execute that instruction
◆ The compiler will not execute the machine code	◆ The interpreter executes the machine code generated
◆ Compiler requires more memory while translating	◆ Interpreter requires less memory while translating
◆ Compiler is not required to execute the executable code	◆ Interpreter is required whenever a program has to be executed
◆ Compiler is costlier	◆ It is cheaper than the compiler
◆ Compiled program runs faster	◆ Interpreted code runs slower
◆ Object program is created only when there are no syntax and semantic errors in the program	◆ Program can be executed even if the syntax errors are present in later stages but execution stops whenever syntax error is encountered.

Now, let us see “What is an assembler? What are the activities performed by the compiler?”

**Definition:** An assembler is a translator which converts assembly language program into equivalent machine code. The machine code thus produced is written into a file called object program. The block diagram of the assembler is shown below:



Now, let us see “What are the various functions of the assembler? The various functions of the assembler are shown below:

- ◆ Convert assembly language instructions to their machine language equivalent
- ◆ Convert the data constants into equivalent machine representation. For example, if decimal data 4095 is used as operand, it is converted into hexadecimal value (machine representation) FFF.
- ◆ Build the machine instruction using the appropriate instruction format.

## Compiler Design - 1.5

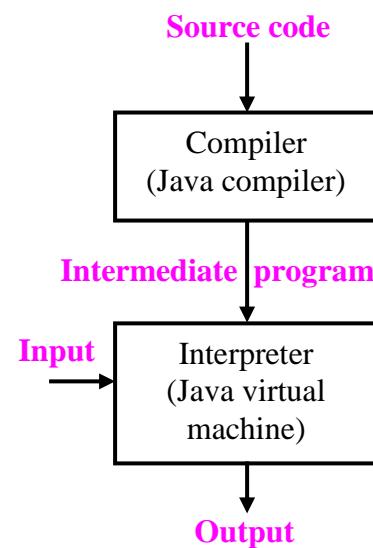
- ◆ Process the instructions given to the assembler directives. These directives help the assembler while converting from assembly language to machine language
- ◆ Create the list file. The list file consists of source code along with address for each instruction and corresponding object code.
- ◆ Display appropriate errors on the screen so that the programmer can correct the program and re-assemble it.
- ◆ Create the object program. The object program consists of machine instructions and instructions to the loader.

Now, let us see “What is hybrid compiler?”

**Definition:** A translator that has features of a compiler as well as features of an interpreter is called *hybrid compiler*. The hybrid compilers translate the source program into intermediate byte code for later interpretation.

For example, the java language processor combines compilation and interpretation as shown in the figure. Observe the following points:

- ◆ The java source program is first translated into intermediate form called *bytecodes* by the java compiler.
- ◆ The *bytecodes* are then interpreted by java virtual machine.
- ◆ During execution, the inputs are accepted and appropriate results are displayed as the output.



**Advantages:** The *bytecodes* compiled on one machine can be interpreted on another machine which has java virtual machine and hence portability issues will not arise.

**Note:** Now, let us see “What is portability with respect to software?” The software written for one machine (with Intel processor) is copied into another machine (with Motorola processor) and that software is perfectly executed in both machines, then we say that the software is portable. Otherwise, the software is not portable.

**Ex1:** The object program obtained from a C compiler in one machine (with Intel processor) can be copied into another machine (with Motorola processor). But, the copied object program cannot be loaded and executed by the Motorola processor. So, we say object program is not portable.

## 1.6 □ Lexical Analyzer

---

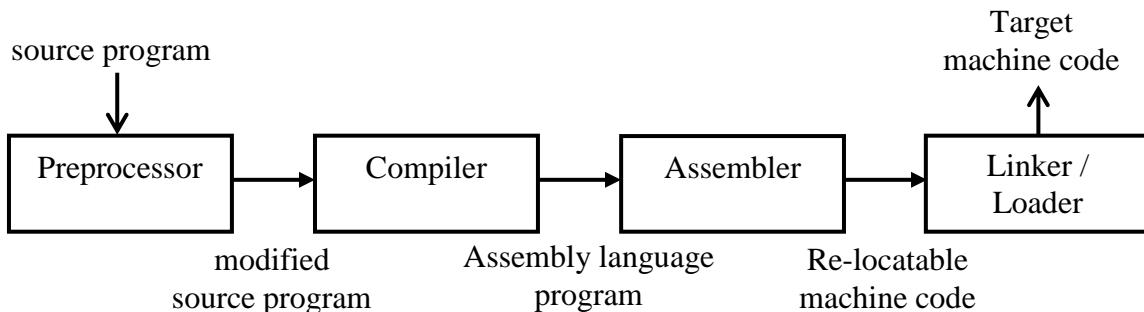
**Ex2:** Any C program may be copied from one machine (with Intel processor) to another machine (with Motorola processor). The C program can be compiled using C compilers in both machines and can be executed in both machines. This is possible if both machines have C compilers. Now, we say that C programs are portable.

**Ex3:** For all java programs, the byte codes can be generated using java compilers. Since, all the machines with java virtual machine can execute the byte codes, we say that all java programs are portable.

### 1.1.2 Language processing system (Cousins of the compiler)

Now, let us see “**What are the cousins of the compiler?**” or let us “**Explain language processing system**” During software development, in addition to a compiler several other programs may be required to create an executable target program. All the programs that are helpful in generating the required executable target program are called *cousins of the compiler*. All these cousins of the compiler are together represent language processing system.

The language processing system or the various cousins of the compiler and their interaction are shown in block diagram below:



Thus, the various programs that constitute language processing system (or cousins of the compiler) are:

**Preprocessor:** The preprocessor is executed before the actual compilation of code begins. A source program may be divided into modules and can be stored in separate files. The preprocessor will collect all source files and may expand macros into source language statements. Thus the output of the preprocessor is also a source program but expanded and modified. This expanded source program is input to the compiler. The main activities performed by the preprocessor are:

- 1) Macro processing which is identified using #define directive
- 2) File inclusion which is identified using #include directive
- 3) It also helps in conditional compilation with the help of the directives such as #if, #else etc.

## Compiler Design - 1.7

**Compiler:** The compiler is a translator that accepts the expanded and modified source program as the input from preprocessor and converts it into assembly language program. If any errors are present, they are displayed along with appropriate error messages and line numbers so that the user can correct the program. The various activities performed by the compiler are:

- 1) The syntax errors are identified and appropriate error messages are displayed along with line numbers
- 2) An optional list file can be generated by the compiler
- 3) The compiler replaces each executable statement in high level language into one or more machine language instructions
- 4) Converts HLL into assembly language or object program if the program is syntactically correct.

**Assembler:** An assembler is a translator which converts assembly language program into equivalent machine code. The machine code thus produced is written into a file called object program. The various functions of the assembler are shown below:

- 1) Convert assembly language instructions to their machine language equivalent
- 2) Convert the data constants into equivalent machine representation. For example, if decimal data 4095 is used as operand, it is converted into hexadecimal value (machine representation) FFF.
- 3) Build the machine instruction using the appropriate instruction format.
- 4) Process the instructions given to the assembler directives. These directives help the assembler while converting from assembly language to machine language
- 5) Create the list file. The list file consists of source code along with address for each instruction and corresponding object code.
- 6) Display appropriate errors on the screen so that the programmer can correct the program and re-assemble it.
- 7) Create the object program. The object program consists of machine instructions and instructions to the loader.

**Linker:** Large programs are often compiled in pieces and generate object programs and stored in files. The *linker* or *linkage editor* accepts one or more object programs generated by a compiler and links the library files and creates an executable file. The linker resolves external reference symbols where code in one file may refer to a location in another file.

**Loader:** A **loader** is the part of an operating system that is responsible for loading programs into main memory for execution. This is one of the important stages in the process of executing a program. Loading a program involves reading the contents of executable file, loading the file containing the program text into memory, and then carrying out other tasks to prepare the executable for running. Once loading is complete, the operating system starts the program by passing control to the loaded program code and execution begins.

## 1.8 □ Lexical Analyzer

---

### 1.2 Analysis of source program

Now, let us “Discuss the analysis of source program with respect to compilation process” The process of converting from high level language to target language is called compilation. The entire compilation process is mapped into two parts namely: Analysis and synthesis

**Analysis part:** It acts as the front end of the compiler and performs various activities as shown below:

- ◆ Accepts the source program and breaks it into pieces and imposes a grammatical structure on them
- ◆ It then uses the above structure to create an intermediate representation of the source program
- ◆ It also displays appropriate error messages whenever a syntax error or semantic error is encountered during compilation. This helps the programmer to correct the program.
- ◆ It also gathers the information about the source program and stores the variables and data in a data structure called symbol table along with type of data, line numbers etc.
- ◆ The symbol table thus obtained along with intermediate representation of the source program is sent to syntheses part.

**Synthesis part:** This is the back end of the compilation process and performs the following activities:

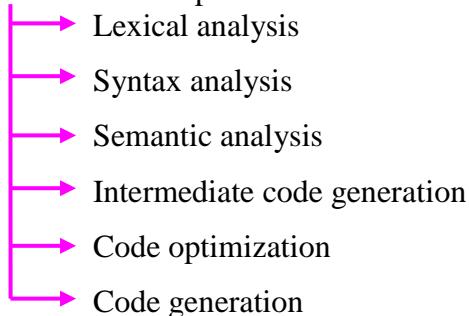
- ◆ It accepts the intermediate representation of source program along with symbol table from the analysis part.
- ◆ It generates the target program or object program as the output.

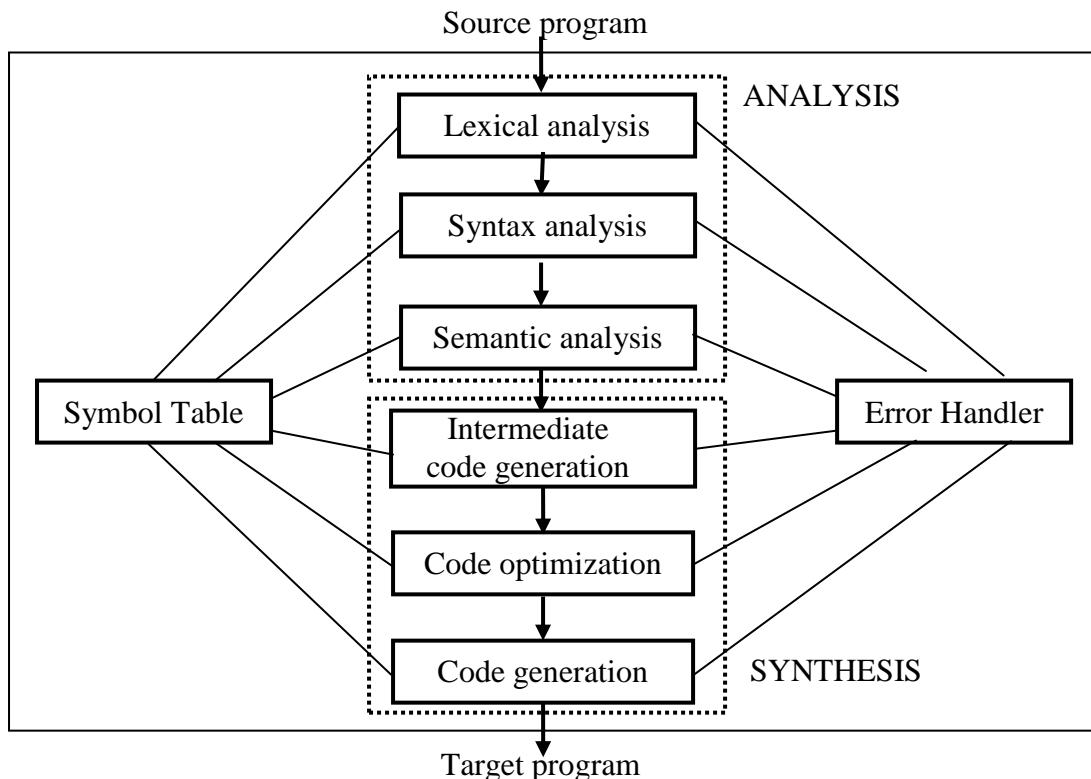
To understand the compilation process in more detail, this large system should be divided into various modules with proper interfaces as discussed in the next section.

#### 1.2.1 The phases of a compiler

In this section, let us see “What are the different phases of the compiler?” or “Explain the block diagram of the compiler”

The various phases of the compiler and the block diagram is shown below:





**Lexical analysis:** It is the first phase of the compiler. It accepts the source program as the input and produces tokens as the output. During this phase, the lexical analyzer performs following activities:

- ◆ Removes white spaces (tabs, spaces and newline characters).
- ◆ Removes comments from the program.
- ◆ Interacts with symbol table to insert variables and constants into symbol table and returns a token consisting of an integer code and pointer to variable and constant in the symbol table.
- ◆ Detects invalid identifiers and interacts with error handler to display the error messages.

**Example 1.1:** Show the various tokens generated by the lexical analyzer for the following statement:

$a = b + c$

The various tokens generated by the lexical analyzer for the statement “ $a = b + c$ ” are shown below:

## 1.10 □ Lexical Analyzer

---

<b>lexemes</b>	<b>tokens</b>
a	ID, 1
=	ASSIGN
b	ID, 2
+	PLUS
c	ID, 3

**Note:** For all symbols and keywords used in the program the unique numbers are assigned and available to both lexical analyzer and syntax analyzer as shown in header file:

```
#define ASSIGN 1
#define PLUS 2
#define MUL 3
#define LP 4
.....
.....
#define ID 100
#define NUM 101
```

**Syntax analysis:** It is the second phase of the compiler. It accepts the tokens from the lexical analyzer and imposes a grammatical structure on them and checks whether the program is syntactically correct or not. The various activities done by the parser are:

- ♦ Detects syntax errors (such as undefined symbol, semicolon expected etc.)
- ♦ Interacts with symbol table and the error handler to display appropriate error messages
- ♦ Generates parse tree (or syntax tree which is the compressed form of the parse tree)

**Semantic analysis:** It is the third phase of the compiler. Type checking and type conversion is done by this phase. Other operations that are performed by this phase are:

- ♦ Collects type information (such as int, float, char etc.) of all variables and constants for subsequent code generation phase
- ♦ Checks the type of each operand in an arithmetic expression and report error if any (For example, subscript variable *i* in an array *a* should not be **float** or **double**)
- ♦ Type conversion at the appropriate place is done in this phase. (For example, conversion from 10 to 10.0)
- ♦ Detects error whenever there is a mismatch in the type of arguments and parameters in the function call and function header

**Intermediate code generation:** It is the fourth phase of the compiler. The syntax tree which is the output of the semantic analyzer is input to this phase. Using this syntax tree,

## Compiler Design - 1.11

it generates intermediate code which is suitable for generating the code. Some of the activities that are performed by this phase are:

- ◆ High level language statements such as while, if, switch etc., are translated into low level conditional statements
- ◆ Produces stream of simple instructions and fed to the next phase of the compiler

For example, consider the statement:

$$b = a * b + c$$

The intermediate code generated by this phase may be:

$$\begin{aligned}t1 &= a * b \\t2 &= t1 + c \\b &= t2\end{aligned}$$

**Note:** The above intermediate code is input to code optimizer which is the next phase of the compiler

**Code optimization:** This is the fifth phase of the compiler. The intermediate code generated in the previous phase is the input. The output is another intermediate code that does the same job as the original but saves time and space. That is, output of this phase is the optimized code.

For example, consider the statement:

$$\begin{aligned}a &= b + c * d \\e &= f + c * d\end{aligned}$$

The above two statements might be evaluated as:

$$\begin{aligned}t1 &= c * d \\a &= b + t1 \\e &= f + t1\end{aligned}$$

Observe that computation of  $c * d$  is done only once. So, the code is optimized.

**Code generation:** This is the last phase of the compiler. The optimized code obtained from the previous phase is the input. It converts the intermediate code into equivalent assembly language instructions. This phase has to utilize the registers very efficiently to

## **1.12 □ Lexical Analyzer**

---

generate the efficient code. For example, if the intermediate code generated by the previous phase is:

$$a = b + c$$

The code generated by this phase can be:

LOAD	B
ADD	C
STORE	A

**Symbol table:** This is the module that can be accessed by all phases of the compiler. The information of all the variables, constants along with type of variables and constants are stored here.

**Error handler:** This routine is invoked whenever an error is encountered in any of the phases. This routine attempts a correction so that subsequent statements need not be discarded and many errors can be identified in a single compilation.

### **1.2.2 The grouping of phases**

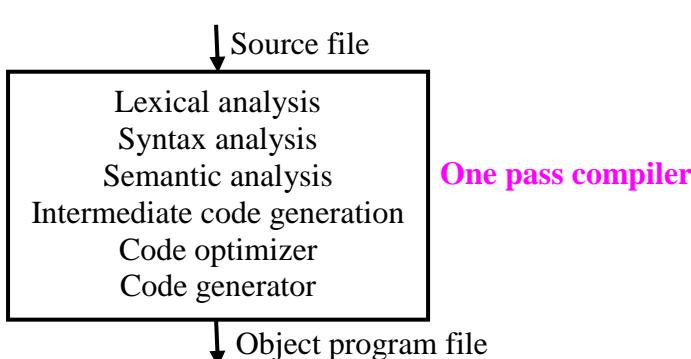
In this section, let us see “What is the difference between a phase and pass?”

First, let us see “What is a phase?”

**Definition:** Since compiler is a very complex program, to understand the design of the compiler, it is logically divided into various sub-programs called modules. These sub-programs or *modules* are also called *phases*. Each phase or module accepts one representation of the program as the input and generates another representation of the program as the output. Thus, a *phase* is nothing but a module of the compiler that accepts one representation of the program as the input and produces another representation of the program as the output. The various phases of the compiler along with input and output are pictorially shown in page 1.9.

Now, let us see “What is a pass?”

**Definition:** A group of one or more phases of a compiler that perform *analysis* or *synthesis* of the source program is called a *pass* of the compiler. Even though the compiler is divided into various phases, in the actual implementation of the compiler, one or more phase may be grouped together into a pass. Each pass reads an input file and writes an output file. The compilers based on the number of passes are classified as shown below:

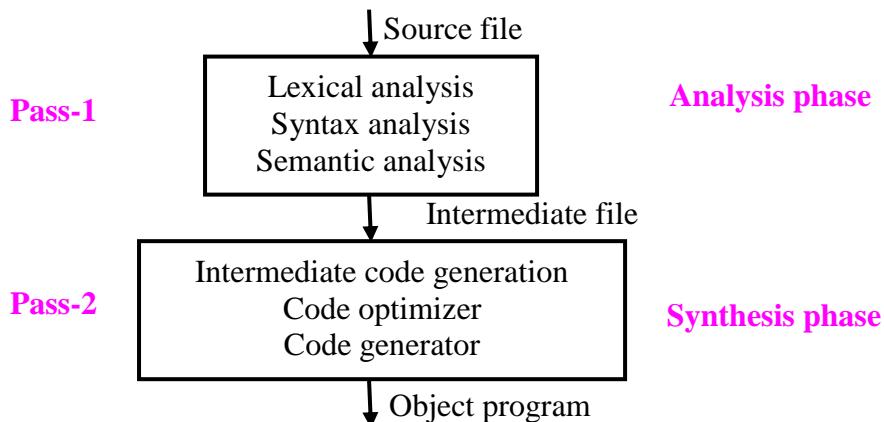
- ◆ One pass compiler
  - ◆ Two pass compiler
  - ◆ Three pass compiler
  - ◆ Multipass compiler
- ◆ **One pass compiler:** If analysis and syntheses of source program is done in one flow from beginning to end of the program, then the compiler is called *one pass compiler*. In this case, all the phases of a compiler starting from *lexical analysis phase* to *code generation phase* are combined together as one unit to convert the source program into object program in one single flow. This is pictorially represented as shown below:
- 
- The single pass compiler or one pass compiler is beneficial because of the following reasons:
- Simplifies the job of writing a compiler
  - One pass compilers generally perform compilations faster than multi-pass compilers
- ◆ **Two pass compiler:** If analysis is of source program is done in one pass and syntheses of source program is done in another pass, then the compiler is called *two pass compiler*.

In this case, the first three phases i.e., *lexical analysis* phase, *syntax analysis* phase, *semantic analysis* phase are implemented as one unit and constitute *first pass of compiler*. This is often called *front-end of compiler*. The other three phases i.e., *intermediate code generation* phase, *code optimizing* phase and *code generation* phase are implemented as second unit and constitute *second pass of compiler*. This is often called *back-end of compiler*.

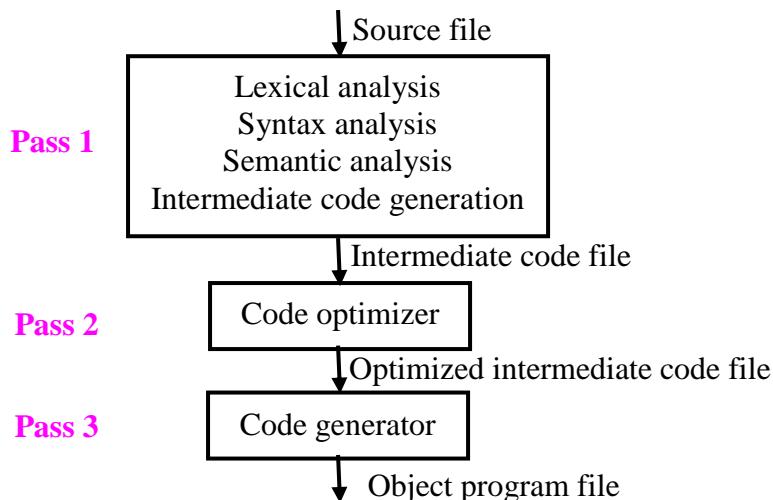
- The output of the first pass is normally stored a file.
- The second pass accepts this file as the input and produces the object program.

## 1.14 □ Lexical Analyzer

This is pictorially represented as shown below:



- ♦ **Three pass compiler:** If analysis and synthesis of source program is done in three different passes, then the compiler is called *three pass compiler*. In this case, the first four phases i.e., *lexical analysis* phase to *intermediate code generation phase* are implemented as one unit (pass), the *code optimizer* as second unit (pass) and *code generation* as third unit(pass). All the three passes are pictorially represented as shown below:



- **Pass 1:** The front end of the phases such as *lexical analysis*, *syntax analysis*, *semantic analysis* and *intermediate code generation* phases may be grouped into one unit called pass1 of compiler. Pass 1 accepts the source program and produces the intermediate code file. This file is internal and stored in main memory. In some compilers, this file is also stored in the disk and hence it is visible when we list the directory after pass 1.

## Compiler Design - 1.15

- **Pass 2:** The intermediate file created in pass 1 is input to pass 2. It produces another intermediate code file which is optimized. Each line in this file contains simple statements or expressions with maximum of 2 or 3 operands along with operators.
- **Pass 3:** The optimized intermediate file created in pass 2 is input to pass 3. It converts the intermediate code into assembly language program or the target program. It may have built in assembler that converts assembly language into machine code.

**Note:** Even though single pass compiler has some advantages, it has some disadvantages too. Some of the disadvantages are:

- ◆ It is not possible to perform many of the sophisticated optimizations needed to generate high quality code.
- ◆ It is difficult to count the number of passes made by an optimizing compiler. For example, one expression may be analyzed many times whereas another expression may be analyzed only once.

All the above disadvantages are overcome using multi-pass compiler. Now, let us see “What is a multi-pass compiler? Explain the need for multiple passes in compiler?”

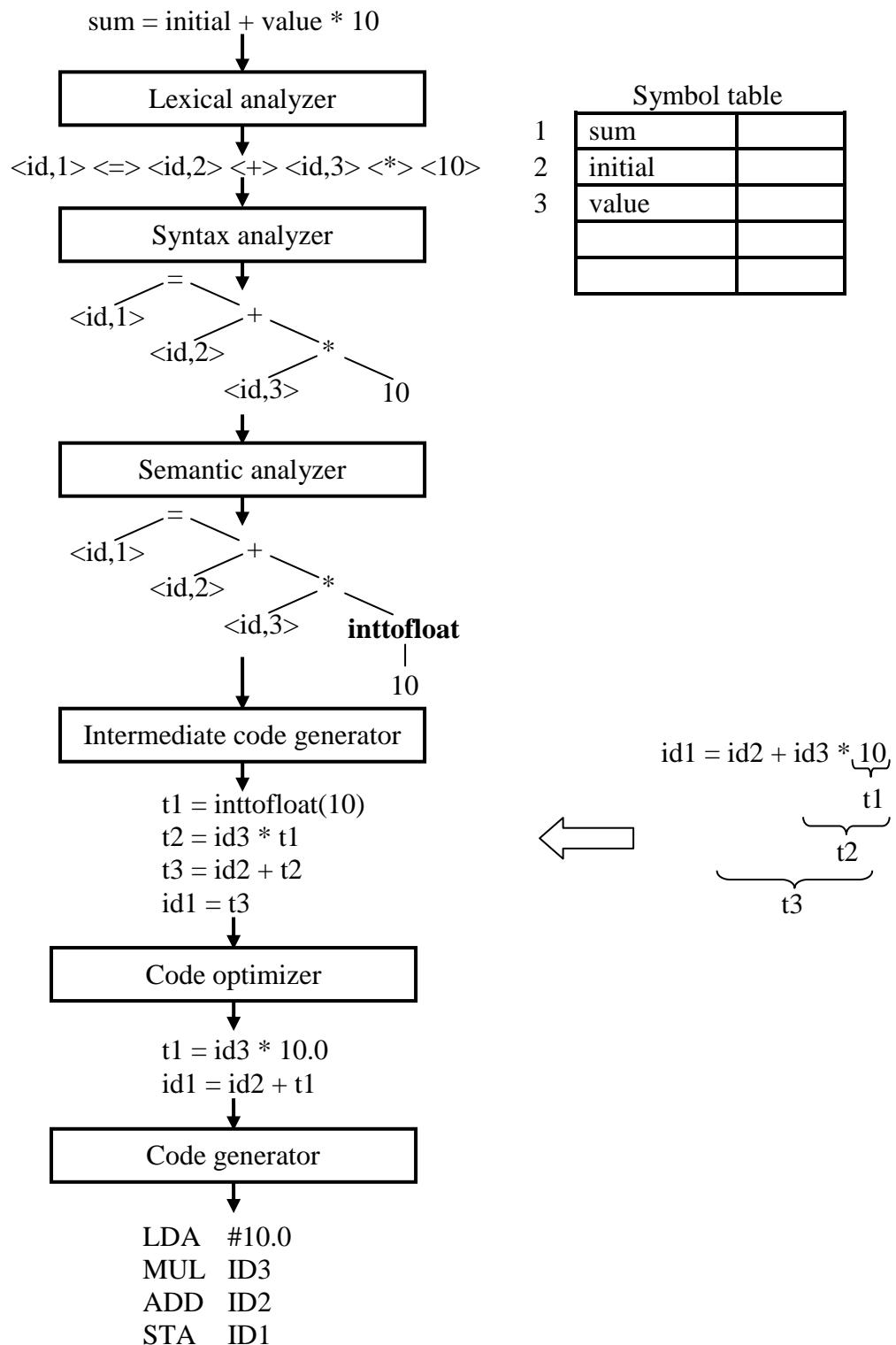
**Definition:** A **multi-pass compiler** is a type of compiler that processes the source code two or more times. The multi-pass compilers are slower, but much more efficient when compiling.

Multi-pass compilers are used for the following reasons:

- ◆ By changing the front-end of the compiler and retaining the back-end of the compiler for a particular target machine, it is possible to write compilers for different languages.
- ◆ On similar lines, by retaining the front-end and changing the back-end of the compiler, it is possible to produce a compiler for different target machines.
- ◆ If there is feature of a language that requires a compiler to perform more than one pass over the source, then multi-pass compiler is used.
- ◆ Splitting a compiler up into small programs is a technique used by researchers and designers interested in producing provably correct compilers. Proving the correctness of a set of small programs often requires less effort and time than proving the correctness of a larger, single, equivalent program.

Now, let us “Show the output of each phase of the compiler for the assignment statement:  
***sum = initial + value \* 10***“ The translation process is shown below:

## 1.16 □ Lexical Analyzer



### 1.3 Compiler construction tools

In this section, let us see “What are compiler construction tools?” The compiler writer, like any software developer, can profitably use modern software environments containing tools such as editors, debuggers and so on. Some of the tools that help the programmer to build the compiler very easily and efficiently are listed below:

- ◆ **Parser generators:** They accept grammatical description of a programming language and produce syntax analyzers.
- ◆ **Scanner generators:** They accept regular expression description of the tokens of a language and produce lexical analyzers.
- ◆ **Syntax-directed translation engines:** They produce various routines for walking a parse tree and generating the intermediate code.
- ◆ **Code generator generators:** Using the intermediate code generated, they produce the target language. The target language may be object program or assembly language program.
- ◆ **Data flow analysis engines:** It gathers the information of how values are transmitted from one part of a program to other part of the program. Data flow analysis is key part of the code optimization
- ◆ **Compiler construction toolkits:** They provide a set of routines for constructing various phases of the compiler.

### 1.4 Lexical analysis

Now, let us see “What is lexical analysis?”

**Definition:** The process of reading the source program and converting it into sequence of tokens is called lexical analysis.

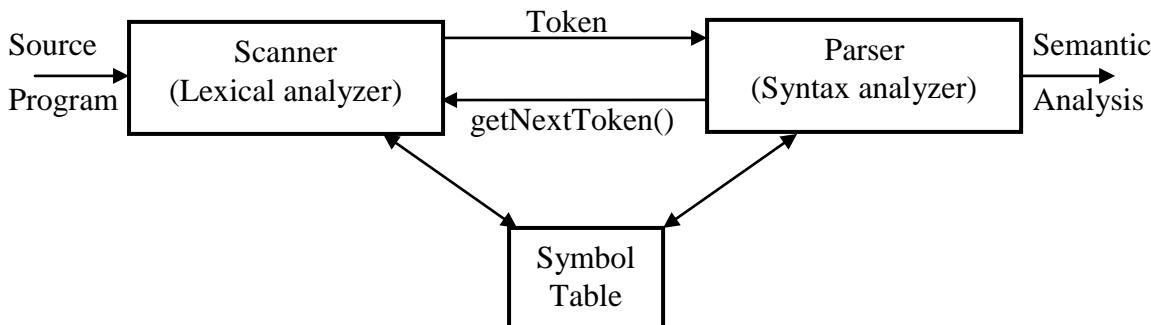
#### 1.4.1 The role of lexical analyzer

Now, let us see “What is the role of lexical analyzer?” The lexical analyzer is the first phase of the compiler. The various tasks that are performed by the lexical analyzer are:

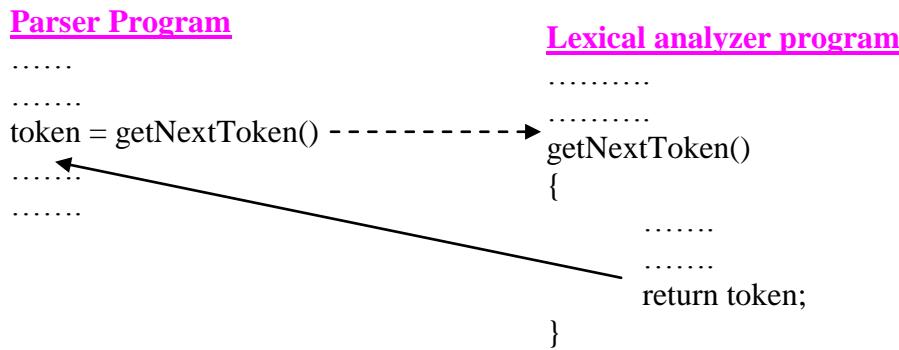
- ◆ Read a sequence of characters from the source program and produce the tokens.
- ◆ The tokens thus generated are sent to the parser for syntax analysis. The parser is also called syntax analyzer.
- ◆ During this process, lexical analyzer interacts with symbol table to insert various identifiers and constants. Sometimes, information of identifier is read from symbol table to assist in determining the proper token to send to the parser.

The interaction between the lexical analyzer and the parser is pictorially represented as shown below:

## 1.18 □ Lexical Analyzer



- ◆ The parser program calls the function `getNextToken()` which is the function defined in lexical analyzer (See the calling sequence below)



- ◆ The function `getNextToken()` of lexical analyzer returns the token back to parser for parsing.
- ◆ If the token obtained is an identifier, it is entered into the symbol table along with various attribute values and returns a token as a pair consisting of an integer code denoted by ID and a pointer to the symbol table for that identifier.
- ◆ The other actions that are performed by the parser are:
  - Removes comments from the program.
  - Remove white spaces such as blanks, tabs and newline characters from the source program and then tokens are obtained.
  - Keep track of line numbers so as to associate line numbers with error messages.
  - If any errors are encountered, the lexical analyzer displays appropriate error messages along with line numbers
  - Preprocessing may be done during lexical analysis phase

### 1.4.2 Lexical analysis verses parsing

Now, let us see “**Why analysis portion of the compiler is separated into lexical analysis and syntax analysis phase?**” The various reasons for which lexical analysis is separated from syntax analysis are shown below:

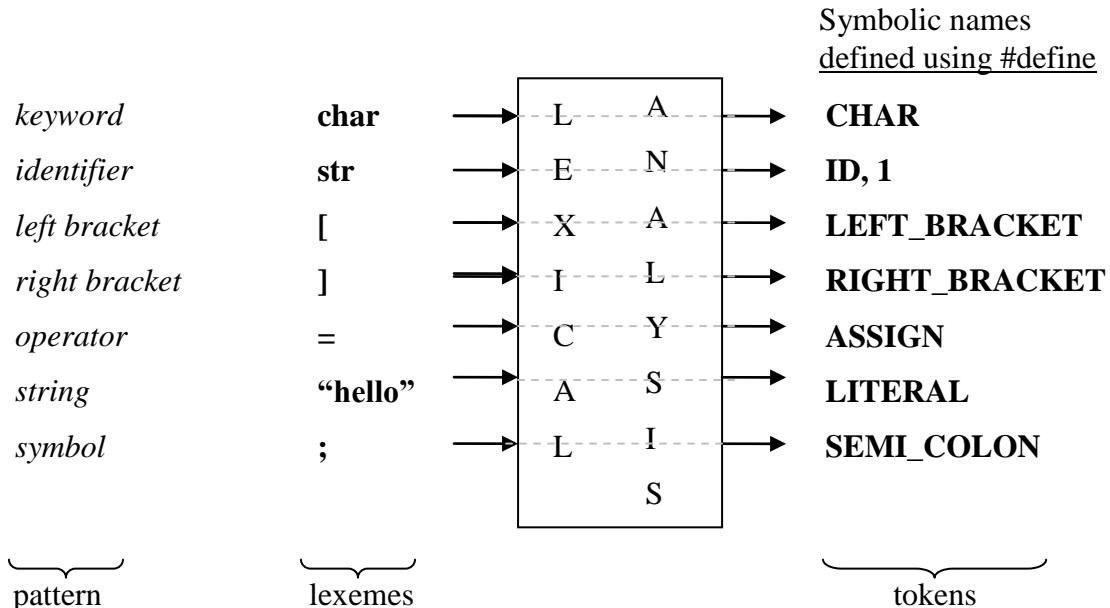
- ♦ **Simplicity of the design:** The separation of lexical analysis and syntax analysis allows us to simplify some of the tasks listed below:
  - A parser that deals with comments and white spaces would be more complex. So, to simplify this task and reduce the complexity of parser, removing of comments and white spaces is done by the lexical analysis
  - When designing a new language, the separation of first two phases results in cleaner overall language design
- ♦ **Compiler efficiency is improved:** A separate lexical analyzer allows us to use specialized techniques for lexical task and generation of tokens will be very easier and efficient. Also, specialized buffering techniques are used for reading the input characters which will speed up the compiler efficiently.
- ♦ **Portability enhancement:** By restricting the input-device-specific peculiarities, the compiler portability can be enhanced.

#### 1.4.3 Tokens, patterns and lexemes

In this section, before proceeding further, let us see “**What is the meaning of *patterns*, *lexemes* and *tokens*?**” To explain the meaning of these three words, consider the following statement:

```
char str[] = "hello";
```

In the above statement, the *patterns*, *lexemes* and respective *tokens* are shown below:



## 1.20 □ Lexical Analyzer

---

Observe that lexemes are input to lexical analyzer and lexical analyzer produces tokens. The description of a lexeme is a pattern. Now, let us study the meaning of token, pattern and lexemes in detail.

### 1.4.3.1 Tokens

Now, let us see “What is a token?”

**Definition:** A token is a pair consisting of *token name* and an optional *attribute value*. The *token names* are basically integer codes represented using symbolic names written in capital letters such as INT, FLOAT, SEMI\_COLON etc (defined in the file token.h in next page). The attribute values are optional and will not be present for keywords, operators and symbols. The attribute values are present for all identifiers and constants.

- ♦ For every keyword there is a unique token name. For example, INT, FLOAT, CHAR, IF, WHILE etc
- ♦ For every symbol there is a unique token name. For example, SEMI\_COLON, COLON, COMMA, LEFT\_PARANTHESIS, RIGHT\_PARANTHESIS, ASSIGN etc.
- ♦ For an identifier *sum*, the token is <ID, 1> where ID is the token name and 1 is the position of the identifier in the symbol table

Whenever there is a request from the parser, the lexical analyzer sends the token. So, tokens are output of the lexical analyzer and input to the syntax analyzer. The syntax analyzer uses these tokens to check whether the program is syntactically correct or not by deriving the tokens from the grammar. All the tokens are represented using symbolic constants defined using #define directive as shown below:

---

**Note:** A simple header file “**token.h**” which is shared by both lexical analyzer and syntax analyzer with various tokens and corresponding integer codes is shown below:

---

```
/* TOKENS with corresponding integer codes for keywords */  
#define IF 1  
#define ELSE 2  
#define INT 3  
#define CHAR 4  
#define DOUBLE 5  
#define FLOAT 6  
.....  
.....  
.....  
.....
```

```

/* TOKENS with corresponding integer codes for operators */
#define PLUS      50
#define MINUS     51
#define SUBTRACT  52
#define DIVIDE    53
.....
.....
/* TOKENS with corresponding integer codes for symbols */
#define LEFT_BRACKET   70
#define RIGHT_BRACKET  71
#define LEFT_PARANTHESIS 72
#define RIGHT_PARANTHESES 73
#define SEMI_COLON    74
.....
.....
/* TOKENS with corresponding integer codes for identifiers and literals */
#define ID          100
#define LITERAL     101

```

### 1.4.3.2 Lexeme

Now, let us see “What is a lexeme?”

**Definition:** A sequence of characters in the source program that matches the patterns such as identifiers, numbers, relational operators, arithmetic operators, symbols such as #, [ , ], ( , ) and so on are called lexemes. In other words, a *lexeme* is a string of patterns read from the source file that corresponds to a token.

### 1.4.3.3 Patterns

Now, let us see “What is a pattern?”

**Definition:** The description of a lexeme is called pattern. More formally, a pattern is described as a rule describing set of lexemes. The various patterns are shown below:

- ◆ **Keyword:** The pattern *keyword* is a sequence of characters that form reserve words of a language. For example, **int**, **if**, **else**, **while**, **do**, **switch** etc are all reserve words. They are also called keywords
- ◆ **Identifier:** The pattern *identifier* is described a sequence of letters or underscores followed by any number of letters or digits or underscores. For example, *sum*, *i*, *pos*, *first*, *rate\_of\_interest* that represent variables in a program or that represent names of functions, structures etc. are all treated as *identifiers*.

## 1.22 □ Lexical Analyzer

---

- ◆ **Relational Operator:** The pattern *relational operators* which is described as the symbols that represent various relational operators of a language. For example: <, <=, >, >=, ==, != represent patterns identifying the relational operators.
- ◆ **Symbols:** The pattern *symbols* is described as set of symbols such as #, \$, (, ), [ , ], { , }, : and so on

---

**Example 1.1:** Identify lexemes and tokens in the following statement:

a = b \* d;

---

**Solution:** The lexemes, patterns and tokens for the above expression are shown below:

- ◆ *a* is a lexeme matching the pattern *identifier* and returning the token <ID, 1> where ID is the token name and 1 is the position of identifier *a* in the symbol table
- ◆ *=* is a lexeme matching the *pattern symbol* '=' and returning the token ASSIGN
- ◆ *b* is a lexeme matching the pattern *identifier* and returning the token <ID, 2> where ID is the token name and 2 is the position of identifier *b* in the symbol table
- ◆ *\** is a lexeme matching the *pattern symbol* '\*' and returning the token STAR
- ◆ *d* is a lexeme matching the pattern *identifier* and returning the token <ID, 3> where ID is the token name and 3 is the position of identifier *d* in the symbol table
- ◆ ; is a lexeme matching the pattern *semicolon* and returning the token SEMICOLON

**Note:** All capital letters in the above solution represent symbolic names of integers defined in the file “tokens.h” given in section 1.4.3.1 (page 1.20)

---

**Example 1.2:** Identify lexemes and tokens in the following statement:

printf("Simple Interest = %f\n", si);

---

**Solution:** The lexemes, patterns and tokens for the given **printf** statements are shown below:

- ◆ *printf* is a lexeme matching the pattern *identifier* and returning the token <ID, 1> where ID is the token name and 1 is the position of identifier *printf* in the symbol table
- ◆ The character '(' is a lexeme matching the pattern *symbol* and returning the token LEFT\_PARANTHESES
- ◆ The sequence of characters “Simple Interest = %f\n” is a lexeme matching the pattern *string* and returning the token <LITERAL, 2> where LITERAL is the token name and 2 is the position of literal in the symbol table
- ◆ The character ',' is a lexeme matching the pattern *symbol* and returning the token COMMA

- ♦ *si* is a lexeme matching the pattern *identifier* and returning the token <ID, 3> where ID is the token name and 3 is the position of identifier *si* in the symbol table
- ♦ The character ‘;’ is a lexeme matching the pattern *symbol* and returning the token SEMI\_COLON

#### 1.4.4 Attributes for tokens

We know that a token is a pair consisting of token name and optional attribute value. Now, let us see “What is the need for returning attributes for tokens along with token name?”

When more than one lexeme matches the pattern, the lexical analyzer must provide additional information about the type of lexeme matched, to subsequent compiler phases. For example,

the pattern for *identifier* matches both *sum* and *pos*

The lexical analyzer returns the token ID for both lexemes. For the token ID, we need to associate more information such as:

- ♦ lexemes (For example, *sum* and *pos*)
- ♦ the type of identifier (such as int, float, char, double etc.)
- ♦ the location at which the identifier found (say found in line number 10 and 20). This information is required to display appropriate error messages along with line numbers

All the above information must be kept in the symbol table. Thus, appropriate attribute value for the identifier is “pointer to symbol table entry for an identifier”. But, we have used the position of identifier in the symbol table as the attribute value.

It is very important for the code generator to know which lexeme was found in the source program to generate the actual code and to associate correct value to that variable. Thus, lexical analyzer returns token name and its attribute value. The token name is used by the parser to check for syntax errors, whereas the attribute value is used during translation into machine code.

---

**Example 1.3:** Give the token names and attribute values for the following statement:

*E = M \* C \*\*2.*

where **\*\*** in FORTRAN language is used as exponent operator.

---

**Solution:** The various token names and attribute values for the above statement are shown below:

<ID, 1000>  
<ASSIGN>

## 1.24 □ Lexical Analyzer

<ID, 1010>  
<STAR>  
<ID, 1020>  
<POWER>  
<NUM, 2>

Symbol Table		
	lexemes	type
line no.		
1000	E	float
1010	M	float
1020	C	float

**Example 1.4:** Give the token names and attribute values for the following statement:

si = p \* t \* r / 100

**Solution:** Assume the variables *si*, *p*, *t*, and *r* are already declared and they are available in symbol table. The token names and associated attribute values for the above statement are shown below:

<ID, 1000>  
<ASSIGN>  
<ID, 1010>  
<STAR>  
<ID, 1020>  
<STAR>  
<ID, 1030>  
<DIVIDE>  
<NUM, 100>

Symbol Table		
	lexemes	type
line no.		
1000	si	float
1010	p	float
1020	t	float
1030	t	float

**Note:** In the above set of tokens, the second component of each pair such as 1000, 1010, 1020 and 1030 are the attribute values which contain the addresses to the symbol table.

## 1.5 Input buffering

Now, let us see “Why input buffering is required?” Input buffering is very essential for the following reasons:

- ♦ Since lexical analyzer is the first phase of the compiler, it is the only phase of the compiler that reads the source program character-by-character and consumes considerable time in reading the source program. Thus, the speed of lexical analysis is a concern while designing the compiler.
- ♦ Lexical analyzers may have to look one or more characters beyond the next lexeme before we have the right lexeme.

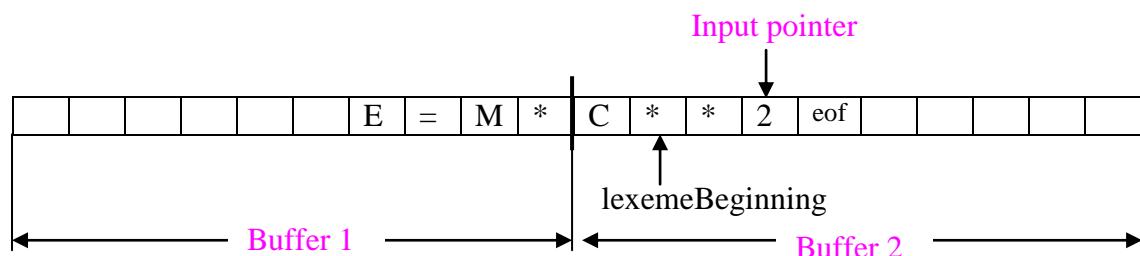
For this reason, we use the concept of input buffering where a block of 1024 or 4096 or more characters are read in one memory read operation and stored in the array to speed up the process.

Now, let us see “What is input buffering?”

**Definition:** The method of reading a block of characters (1K or 4K or more bytes) from the disk in one read operation and storing in memory (normally in the form of an array) for further processing and faster accessing is called *input buffering*. The memory (an array) where a block of characters read from the disk are stored is called *buffer*. Now, instead of reading character-by-character from the disk, we can directly read from the buffer to identify the lexemes for the tokens. This enhances the speed of the lexical analyzer as character-by-character reading from the disk consume considerable amount of time. The principle behind input buffering can be explained using:

- ◆ Buffer pairs
- ◆ Sentinel

**Buffer pairs:** In this input buffering technique two buffers are used as shown below:



- ◆ The size of each buffer is N where N is usually the size of the disk block. If size of disk block is 4K, in one read operation 4096 characters can be read into the buffer using one system command rather than using one system call per character which consumes lot of time.
- ◆ If less than 4096 characters are present in the disk, then a special character represented by **eof** is stored in buffer indicating no-more characters are present in the disk file to read.
- ◆ Two pointers to the input are maintained as shown below:
  - 1) A pointer *lexemeBeginning* is used and it is pointing to the current lexeme.
  - 2) Another pointer called *input pointer* is used to scan ahead until the pattern is found.
- ◆ Now, all the characters starting from *lexemeBeginning* till the *input pointer* (excluding it) is the current lexeme. The token corresponding to this lexeme must be returned by the lexical analyzer.
- ◆ Once the token is found the pointer *lexemeBeginning* points to the *input pointer* to identify the next lexeme.
- ◆ As the *input pointer* moves forward, if end of buffer is encountered, read one more block whose size is N from the disk and load into other buffer.

## 1.26 □ Lexical Analyzer

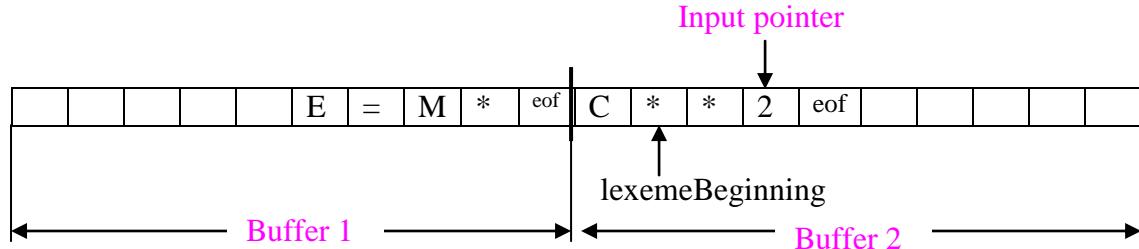
---

Now, let us see “What is the disadvantage of input buffering with buffer pairs?” or “Explain the use of sentinels in recognizing the tokens” In input buffering with buffer pairs technique, each time we move the *input pointer* towards right, it indicates that we have not moved off the buffer. If we moved off the buffer, we have to load the other buffer. Thus, for each character read we have to do two tests:

- ♦ The first test is to determine the end of the buffer
- ♦ Second test is to determine what character is read.

It is possible to combine the buffer-end test with the test for the current character, if we extend the each buffer to hold a *sentinel* character at the end.

**Sentinels:** A *sentinel* is a special character that cannot be the part of the source program. So, the natural choice is to use a character **eof** which acts as *sentinel* as well as end of the input program. The input buffering technique with *sentinels* also uses two buffers with *sentinel* added at the end as shown below:



- ♦ The size of each buffer is N where N is usually the size of the disk block. If size of disk block is 4K, in one read operation 4096 characters can be read into the buffer using one system command rather than using one system call per character which consumes lot of time.
- ♦ Irrespective of number of characters stored in the buffer, last character of each buffer is **eof**.
- ♦ Note that **eof** retains its use as a marker for the end of the entire input. Any **eof** that appears other than at the end of a buffer means that the input is at an end.
- ♦ The use of two pointers *lexemeBeginning* and *input pointer* and the method of accessing *lexeme* remains same as in buffer pairs.
- ♦ The algorithm consisting of lookahead code with sentinels is shown below:

```
switch (*inputPointer++)  
{  
    case eof: If (inputPointer is at end of first buffer)  
    {  
        reload the second buffer;  
        inputPointer = beginning of second buffer;  
        break;  
    }  
}
```

```

if (inputPointer is at the end of second buffer)
{
    reload the first buffer;
    inputPointer = beginning of first buffer;
    break;
}

/* eof within a buffer indicates the end of the input */
/* So, terminate lexical analysis */
break;

/* Cases for other characters */
}

```

Observe from the above algorithm that instead of having two tests as in *buffer pair* technique, there is only one test i.e., testing the **eof** marker.

## 1.6 Specifications of tokens

The regular expressions are the notations for specifying the pattern. Even though they cannot express all possible patterns, they are very effective in specifying those types of patterns that are necessary for tokens. In the next section, let us see how to build lexical analyzers by converting regular expressions to finite automata. The definition of some of the terms are discussed here:

**Definition:** An **alphabet** is any finite set of symbols. The set of alphabets is denoted by the symbol  $\Sigma$ . Some of the alphabets are letters, digits and punctuations.

**Ex 1:**  $\Sigma = \{0, 1\}$  is set of binary alphabet

**Ex 2:**  $\Sigma = \{0, 1, 2\}$  is a set of ternary alphabet

**Definition:** A **string** over an alphabet is a finite sequence of symbols drawn from alphabets. The length of string  $s$  denoted by  $|s|$  gives the number of symbols in the string  $s$ . For example, let  $\Sigma = \{a, b\}$ . The various strings that are obtained from  $\Sigma$  are shown below:

a, aa, ab, ba, bb, .....and so on

**Definitions:** A **language** is set of strings obtained from the alphabets denoted by  $\Sigma$ . For example, language consisting of one or more a's is shown below:

$L = \{ a, aa, aaa, aaaa, \dots \}$

## 1.28 □ Lexical Analyzer

---

**Definition:** A regular expression is recursively defined as shown below:

1.  $\phi$  is a regular expression denoting an empty language.
2.  $\epsilon$ - (epsilon) is a regular expression indicates the language containing an empty string.
3.  $a$  is a regular expression which indicates the language containing only  $\{a\}$
4. If  $R$  is a regular expression denoting the language  $L_R$  and  $S$  is a regular expression denoting the language  $L_S$ , then
  - a.  $R+S$  is a regular expression corresponding to the language  $L_R \cup L_S$ .
  - b.  $R.S$  is a regular expression corresponding to the language  $L_R.L_S$ .
  - c.  $R^*$  is a regular expression corresponding to the language  $L_R^*$ .
5. The expressions obtained by applying any of the rules from 1 to 4 are regular expressions.

The following table shows some examples of regular expressions and the language corresponding to these regular expressions.

Regular expressions	Meaning
$a^*$	String consisting of any number of $a$ 's (or string consisting of zero or more $a$ 's)
$a^+$	String consisting of at least one $a$ (or string consisting of one or more $a$ 's)
$(a+b)$	String consisting of either one $a$ or one $b$
$(a+b)^*$	Set of strings of $a$ 's and $b$ 's of any length including the NULL string.
$(a+b)^*abb$	Set of strings of $a$ 's and $b$ 's ending with the string $abb$
$ab(a+b)^*$	Set of strings of $a$ 's and $b$ 's starting with the string $ab$ .
$(a+b)^*aa(a+b)^*$	Set of strings of $a$ 's and $b$ 's having a sub string $aa$ .
$a^*b^*c^*$	Set of string consisting of any number of $a$ 's (may be empty string also) followed by any number of $b$ 's (may include empty string) followed by any number of $c$ 's (may include empty string).
$a^+b^+c^+$	Set of string consisting of at least one ' $a$ ' followed by string consisting of at least one ' $b$ ' followed by string consisting of at least one ' $c$ '.
$aa^*bb^*cc^*$	Set of strings consisting of at least one ' $a$ ' followed by string consisting of at least one ' $b$ ' followed by string consisting of at least one ' $c$ '.
$(a+b)^* (a + bb)$	Set of strings of $a$ 's and $b$ 's ending with either $a$ or $bb$

(aa)*(bb)*b	Set of strings consisting of even number of a's followed by odd number of b's
(0+1)*000	Set of strings of 0's and 1's ending with three consecutive zeros(or ending with 000)
(11)*	Set of strings consisting of even number of 1's
01* + 1	The language represented is the string 1 plus the string consisting of a zero followed by any number of 1's possibly including none.
(01)* + 1	The language consists of a string 1 or strings of (01)'s that repeat zero or more times.
0(1* + 1)	Set of strings consisting of a zero followed by any number of 1's
(1+ε)(00*1)*0*	Strings of 0's and 1's without any consecutive 1's
(0+10)*1*	Strings of 0's and 1's ending with any number of 1's (possibly none)
(a+b)(a+b)	Strings of a's and b's whose length is 2

### 1.7 Recognition of tokens and construction of lexical analyzer

The finite automaton represented using transition diagram can be easily written using which the tokens can be obtained. Using this transition diagram, we can easily construct the lexical analyzer. Now, let us see how to write the lexical analyzers by looking into transition diagrams. The two functions that are used while designing the lexical analyzer are:

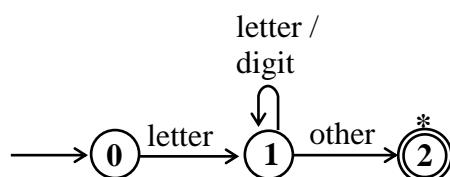
- ◆ **retract():** This function is invoked only if we want to unread the last character read. It is identified by having a an edge labeled other to the final state with \* marked to it. This function un reads the last character read
- ◆ **install\_ID():** Once the identifier is identified, the function install\_ID() is called. This function checks whether the identifier is already there in the symbol table. If it is not there, it is entered into the symbol table and returns the pointer to that entry. If it is already there, it returns the pointer to that entry.

---

#### Example 1.5: Design a lexical analyzer to identify an identifier

---

The transition diagram to identify an identifier is shown below:



## 1.30 □ Lexical Analyzer

---

Observe the following points from the above transition diagram:

**State 0:** If input symbol is a letter goto state 1

**State 1:** If input symbol is a letter or digit remain in state 1. But, any character other than letter and digit go to state 2.

**State 2:** Since other character is read before coming to state 2, push back the other character using the function retract() and return the token ID and pointer to symbol table entry for the identifier recognized by calling the function install\_ID();

So, the complete lexical analyzer to identify the identifier using the above procedure is shown below:

```
state = 0;

for (;;)
{
    switch (state)
    {
        case 0:
            ch = getchar()
            if (ch == letter) state = 1;
            else state = 3;           // Identify the next token
            break;

        case 1:
            ch = getchar()
            if (ch == letter or ch == digit) state = 1;
            else state = 2;
            break;

        case 2: retract();           // undo the last character read
        return (ID, install_ID())
    }

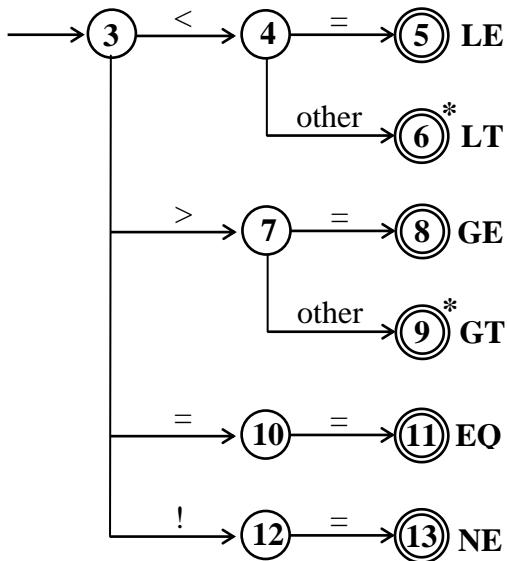
    case 3: /* Identify the next token */
}
```

---

**Example 1.6:** Design a lexical analyzer to identify the relation operators such as <, <=, >, >=, == and !=

---

**Note:** Observe from the previous problem that, states 0, 1 and 2 are used to recognize the token corresponding to an identifier. But, state 3 onwards, other tokens are identified. Now, we start from state 3 to recognize other tokens such as relational operators as shown in following transition diagram.



In the above transition diagram, observe that there are no transitions defined for states 3, 10 and 12 for symbols other than specified as labels. So, from state 3, 10 and 12 on any *other* symbol, we enter into state 14 which is used to identify some other token. The lexical analyzer for identifying relational operators is shown below:

```

state = 0;

for (;;) {
    switch (state) {
        /* Case 0-2: Identify the identifier */
        case 3:
            ch = getchar()
            if (ch == '<')           state = 4;
            else if (ch == '>')       state = 7;
            else if (ch == '=')       state = 10;
            else if (ch == '!')       state = 12;
            else                      state = 14;

            break;

        case 4:
            ch = getchar()
  
```

## 1.32 □ Lexical Analyzer

---

```
    if (ch == '=') state = 5;
    else           state = 6;

    break;

case 5: return LE;

case 6: retract();           // unread the last character read
            return LT;

case 7:
    ch = getchar();

    if (ch == '=') state = 8;
    else           state = 9;

    break;

case 8: return GE;

case 9: retract();           // unread the last character read
            return GT;

case 10:
    ch = getchar();

    if (ch == '=') state = 11;
    else           state = 14;

    break;
case 11: return EQ;

case 12:
    ch = getchar();

    if (ch == '=') state = 13;
    else           state = 14;

    break;

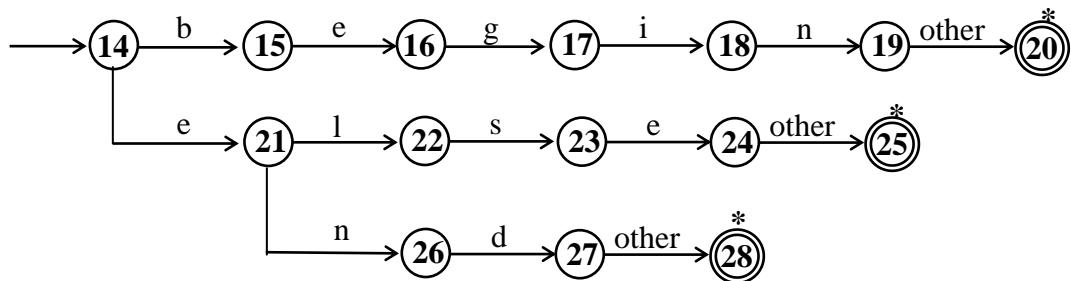
case 13: return NE;

case 14: /* Identify the next token */
}

}
```

**Example 1.7:** Design a lexical analyzer to identify the keywords **begin**, **else**, **end**. Sketch the program segment to implement it showing the first two states and one final state

**Note:** Observe from the previous problem that, states 0 to 13 are used to identify the identifier and relational operators. Let us identify the keywords **begin**, **else**, **end** using the following transition diagram starting from state 14 as shown below:



Similar to the previous two problems, we can write various cases to identify the tokens BEGIN, ELSE and END. The lexical analyzer is implemented by writing the code for first two states and one final state as shown below:

```

state = 0;
for (;;) {
{
    switch (state) {
        /* Case 0-13: Identify other tokens */
        case 14:
            ch = getchar()
            if (ch == 'b')
                state = 15;
            else if (ch == 'e')
                state = 21;
            else
                state = 29; /* other token */

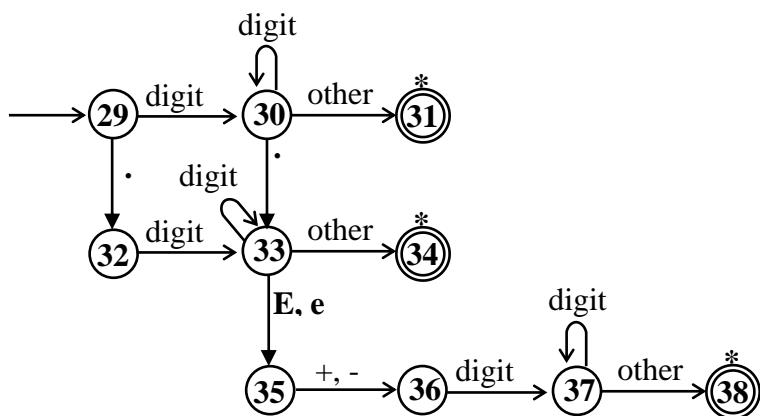
        case 15:
            ch = getchar();
            if (ch == 'e')
                state = 16;
            else
                state = 29; /* other token */
}
}

```

### 1.34 □ Lexical Analyzer

```
break;  
.....  
.....  
case 28: retract();  
return END;  
}  
}
```

**Example 1.8:** Design a lexical analyzer to recognize unsigned number. Sketch the program segment to implement it showing the first two states and one final state



Similar to the previous two problems, we can write various cases to identify unsigned integers and floating point numbers represented using decimal point and scientific method. The lexical analyzer is implemented by writing the code for first two states and one final state as shown below:

```
state = 0;  
for (;;) {  
    switch (state) {  
        /* Case 0-28: Identify other tokens */  
        case 29:  
            ch = getchar();  
  
            if (ch == digit)  
                state = 30;  
            else if (ch == '.')  
                state = 32;
```

```
        else
            state = 39; /* other token */
            break;
    case 30:
        ch = getchar();
        if (ch == digit)
            state = 30;
        else if (ch == '.')
            state = 33;
        else
            state = 31;
        break;
.....
.....
case 31: retract();
        return (NUM, InstallNUM());
    }
}
```

**Note:** Here, `Install_NUM()` is a function which checks whether the number is already in the numeric table. If it is not present, it is entered into the numeric table and returns the token `NUM` and pointer to that integer as the attribute value. Otherwise, it returns the pointer to that integer as the attribute value apart from returning the token `NUM`.

### Exercises

- 1) What is a language processor? What are the various types of language processors?
- 2) What is a compiler? What are the activities performed by the compiler?
- 3) What is an interpreter? What are the differences between a compiler and interpreter
- 4) What is an assembler? What are the activities performed by the compiler?
- 5) What is hybrid compiler?
- 6) What are the cousins of the compiler? or Explain language processing system
- 7) Discuss the analysis of source program with respect to compilation process
- 8) What are the different phases of the compiler? or “Explain the block diagram of the compiler construction method

### **1.36 □ Lexical Analyzer**

---

- 9) Show the output of each phase of the compiler for the assignment statement: ***sum = initial + value \* 10***
- 10) What is the difference between a phase and pass? What is a multi-pass compiler? Explain the need for multiple passes in compiler?
- 11) What are compiler construction tools?
- 12) What is lexical analysis? What is the role of lexical analyzer?
- 13) Why analysis portion of the compiler is separated into lexical analysis and syntax analysis phase?
- 14) What is the meaning of *patterns*, *lexemes* and *tokens*?
- 15) Identify lexemes and tokens in the following statement:  
 $a = b * d;$
- 16) Identify lexemes and tokens in the following statement:  
`printf("Simple Interest = %f\n", si);`
- 17) What is the need for returning attributes for tokens along with token name?"
- 18) Give the token names and attribute values for the following statement:  
 $E = M * C^{**2}.$   
where  $^{**}$  in FORTRAN language is used as exponent operator.
- 19) Give the token names and attribute values for the following statement:  
 $si = p * t * r / 100$
- 20) Why input buffering is required? What is input buffering?
- 21) What is the disadvantage of input buffering with buffer pairs? Explain the use of sentinels in recognizing the tokens.
- 22) Design a lexical analyzer to identify an identifier
- 23) Design a lexical analyzer to identify the relation operators such as  $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ,  $==$  and  $!=$
- 24) Design a lexical analyzer to identify the keywords **begin**, **else**, **end**. Sketch the program segment to implement it showing the first two states and one final state
- 25) Design a lexical analyzer to recognize unsigned number. Sketch the program segment to implement it showing the first two states and one final state

# Chapter 2: Syntax Analysis

## What are we studying in this chapter?

- ◆ The Role of the Parser
- ◆ Context-free Grammars
- ◆ Writing a Grammar
- ◆ Parsing techniques
  - Top-down Parsing
  - Bottom-up Parsing

- 6 hours

## 2.1 Introduction

Every programming language such as C or PASCAL has rules that prescribe the syntactic structure of well-formed programs. The syntax of programming language constructs can be described by CFG or BNF (Backus Naur Form) notation. The parser determines the syntax or structure of a program. That is, it checks whether the input is syntactically correct or not. Before proceeding further, let us see what is a context free grammar, what is derivation and some other important terms that are used in coming chapters.

## 2.2 Context-free Grammars

Now, let us see “What is a context free grammar?”

**Definition:** The context free grammar in short a CFG is 4-tuple  $G = (V, T, P, S)$  where

- ◆  $V$  is set of variables. The variables are also called non-terminals.
- ◆  $T$  is set of terminals.
- ◆  $P$  is set of productions. All productions in  $P$  are of the form  $A \rightarrow \alpha$  where  $A$  is a non-terminal and  $\alpha$  is string of grammar symbols.
- ◆  $S$  is the start symbol.

**Ex 1:** Grammar to generate one of more a's is shown below:

$$A \rightarrow a \mid aA$$

**Ex 2:** Grammar to recognize an if-statement is shown below:

$$\begin{aligned}S &\rightarrow i\ C\ t\ S \mid i\ C\ t\ S\ e\ S \mid a \\C &\rightarrow b\end{aligned}$$

## 2.2 □ Syntax Analyzer

---

where  $i$  – stands for **if** keyword  
 $t$  – stands for **then** keyword  
 $e$  – stands for **else** keyword  
 $a$  – stands for a statement  
 $b$  – stands for a statement

Now, let us see “**What are the various notations used when we write the grammars?**” The various notations used in a context free grammar are shown below:

1. The following symbols are terminals
  - a) The keywords such as **if**, **for**, **while**, **do-while** etc.
  - b) Digits from 0 to 9
  - c) Symbols such as +, -, \*, / etc
  - d) The lower case letters near the beginning of alphabets such as  $a, b, c, d$  etc.
  - e) The bold faced letters such as **id**
2. The following symbols are non-terminals
  - a) The lower case names such as *expression*, *operator*, *operand*, *statement* etc
  - b) The capital letters near the beginning of the alphabets such as A, B, C, D etc
  - c) The letter S is the start symbol
3. The lower case letters near the end of the alphabets such as u, v, w, x, y, z represents string of terminals.
4. The capital letters near the end of the alphabets such as U, V, W, X, Y, Z etc represent grammar symbols. A grammar symbol can be a terminal or a non-terminal.
5. The Greek letters such as  $\alpha, \beta, \gamma, \delta$  etc. represent string of grammar symbols.

## 2.3 Derivation

Now, let us see “**What is derivation?**”

**Definition:** The process of obtaining string of terminals and/or non-terminals from the start symbol by applying some set of productions (it may include all productions) is called *derivation*.

For example, if  $A \rightarrow \alpha B \gamma$  and  $B \rightarrow \beta$  are the productions, the string  $\alpha \beta \gamma$  can be obtained from A- production as shown below:

$$\begin{array}{ll} A \Rightarrow \alpha B \gamma & [\text{Apply the production } A \rightarrow \alpha B \gamma] \\ \Rightarrow \alpha \beta \gamma & [\text{Replace } B \text{ by } \beta \text{ using the production } B \rightarrow \beta] \end{array}$$

The above derivation can also be written as shown below:

$$A \stackrel{+}{\Rightarrow} \alpha \beta \gamma$$

## Systematic approach to Compiler Design - 2.3

Observe the following points:

- ♦ If a string is obtained by applying only one production, then it is called one-step derivation and is denoted by the symbol ' $\Rightarrow$ '.
- ♦ If one or more productions are applied to get the string  $\alpha\beta\gamma$  from A, then we write

$$A \xrightarrow{+} \alpha\beta\gamma$$

- ♦ If zero or more productions are applied to get the string  $\alpha\beta\gamma$  from A, then we write

$$A \xrightarrow{*} \alpha\beta\gamma$$

---

**Example 2.1:** Consider the grammar shown below from which any arithmetic expression can be obtained.

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E - E \\ E &\rightarrow E * E \\ E &\rightarrow E / E \\ E &\rightarrow \text{id} \end{aligned}$$

Obtain the string **id + id \* id** and show the derivation for the same.

---

**Solution:** The derivation to get the string **id + id \* id** is shown below.

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow \text{id} + E \\ &\Rightarrow \text{id} + E * E \\ &\Rightarrow \text{id} + \text{id} * E \\ &\Rightarrow \text{id} + \text{id} * \text{id} \end{aligned}$$

Thus, the above sequence of steps can also be written as:

$$E \xrightarrow{+} \text{id} + \text{id} * \text{id}$$

which indicates that the string **id + id \* id** is obtained in one or more steps by applying various productions.

Now, let us see “**What are the two types of derivations?**” The two types of derivations are:

- ♦ Leftmost derivation
- ♦ Rightmost derivation

### 2.3.1 Leftmost derivation

Now, let us see “**What is leftmost derivation?**”

## 2.4 □ Syntax Analyzer

---

**Definition:** The process of obtaining a string of terminals from a sequence of replacements such that only leftmost non-terminal is replaced at each and every step is called **leftmost derivation**.

For example, consider the following grammar:

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E * E \\ E &\rightarrow (E) \\ E &\rightarrow \text{id} \end{aligned}$$

The leftmost derivation for the string **id + id \* id** can be obtained as shown below:

$$\begin{aligned} E &\stackrel{lm}{\Rightarrow} E + E \\ &\Rightarrow \text{id} + E \\ &\Rightarrow \text{id} + E * E \\ &\Rightarrow \text{id} + \text{id} * E \\ &\Rightarrow \text{id} + \text{id} * \text{id} \end{aligned}$$

### 2.3.2 Rightmost derivation

Now, let us see “**What is rightmost derivation?**”

**Definition:** The process of obtaining a string of terminals from a sequence of replacements such that only right most non-terminal is replaced at each and every step is called **rightmost derivation**.

For example, consider the following grammar:

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E * E \\ E &\rightarrow (E) \\ E &\rightarrow \text{id} \end{aligned}$$

The rightmost derivation for the string **id + id \* id** can be obtained as shown below:

$$\begin{aligned} E &\stackrel{rm}{\Rightarrow} E + E \\ &\Rightarrow E + E * E \\ &\Rightarrow E + E * \text{id} \\ &\Rightarrow E + \text{id} * \text{id} \\ &\Rightarrow \text{id} + \text{id} * \text{id} \end{aligned}$$

## Systematic approach to Compiler Design - 2.5

### 2.4 Sentence

Now, let us see “What is a sentence?”

**Definition:** Let  $G = (V, T, P, S)$  be a CFG. Any string  $w \in (V \cup T)^*$  which is derivable from the start symbol  $S$  such that  $S \xrightarrow{*} w$  is called a **sentence** or ***sentential form*** of  $G$ . For example, consider the derivation:

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow \mathbf{id} + E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned}$$

The final string of terminals i.e.,  $\mathbf{id} + \mathbf{id} * \mathbf{id}$  is called sentence of the grammar.

Now, let us see “What the different sentential forms?” The two sentential forms are:

- ♦ Left sentential form
- ♦ Right sentential form

#### 2.4.1 Left sentential form

Now, let us see “What is left sentential form?”

**Definition:** If there is a derivation of the form  $S \xrightarrow{*} \alpha$ , where at each step in the derivation process only a left most variable is replaced, then  $\alpha$  is called ***left-sentential form*** of  $G$ .

For example, consider the following grammar and its leftmost derivation:

<u>Grammar</u>	<u>leftmost derivation</u>
$E \rightarrow E + E$	$E \xrightarrow{lm} E + E$
$E \rightarrow E * E$	$\Rightarrow \mathbf{id} + E$
$E \rightarrow (E)$	$\Rightarrow \mathbf{id} + E * E$
$E \rightarrow \mathbf{id}$	$\Rightarrow \mathbf{id} + \mathbf{id} * E$
	$\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$

In the above leftmost derivation, the string of grammar symbols obtained in each step such as:

$$\{ E + E, \mathbf{id} + E, \mathbf{id} + E * E, \mathbf{id} + \mathbf{id} * E, \mathbf{id} + \mathbf{id} * \mathbf{id} \}$$

are various ***left-sentential forms*** of the given grammar.

## 2.6 □ Syntax Analyzer

---

### 2.4.2 Right sentential form

Now, let us see “What is right sentential form?”

**Definition:** If there is a derivation of the form  $S \xrightarrow{*} \alpha$ , where at each step in the derivation process only a right most non-terminal is replaced, then  $\alpha$  is called *right-sentential form* of  $G$ .

For example, consider the following grammar and its rightmost derivation:

<u>Grammar</u>	<u>rightmost derivation</u>
$E \rightarrow E + E$	$E \xrightarrow{rm} E + E$
$E \rightarrow E * E$	$\Rightarrow E + E * E$
$E \rightarrow (E)$	$\Rightarrow E + E * id$
$E \rightarrow id$	$\Rightarrow E + id * id$
	$\Rightarrow id + id * id$

In the above rightmost derivation, the string of grammar symbols obtained in each step such as:

{  $E + E$ ,  $E + E * E$ ,  $E + E * id$ ,  $E + id * id$ ,  $id + id * id$  }

are various *right-sentential forms* of the given grammar.

---

**Example 2.2:** Obtain the leftmost derivation for the string **aaabbabbba** using the following grammar.

$$\begin{array}{lcl} S & \rightarrow & aB \mid bA \\ A & \rightarrow & aS \mid bAA \mid a \\ B & \rightarrow & bS \mid aBB \mid b \end{array}$$

---

The leftmost derivation for the string **aaabbabbba** is shown below:

$$\begin{array}{ll} S & \xrightarrow{lm} aB \quad (\text{Applying } S \rightarrow aB) \\ & \Rightarrow aaBB \quad (\text{Applying } B \rightarrow aBB) \\ & \Rightarrow aaaBBB \quad (\text{Applying } B \rightarrow aBB) \\ & \Rightarrow aaabBB \quad (\text{Applying } B \rightarrow b) \\ & \Rightarrow aaabbB \quad (\text{Applying } B \rightarrow b) \\ & \Rightarrow aaabbaBB \quad (\text{Applying } B \rightarrow aBB) \\ & \Rightarrow aaabbabB \quad (\text{Applying } B \rightarrow b) \\ & \Rightarrow aaabbabbS \quad (\text{Applying } B \rightarrow bS) \\ & \Rightarrow aaabbabbA \quad (\text{Applying } S \rightarrow bA) \\ & \Rightarrow aaabbabbba \quad (\text{Applying } A \rightarrow a) \end{array}$$

## Systematic approach to Compiler Design - 2.7

### 2.4.3 Language

Now, let us see “**What is the language generated by grammar?**” The formal definition of the language accepted by a grammar is defined as shown below.

**Definition:** Let  $G = (V, T, P, S)$  be a grammar. The language  $L(G)$  generated by the grammar  $G$  is

$$L(G) = \{ w \mid S \xrightarrow{*} w \text{ and } w \in T^* \}$$

i.e.,  $w$  is a string of terminals obtained from the start symbol  $S$  by applying various productions.

For example, for the grammar  $A \rightarrow a \mid aA$  the various strings that are generated are  $a, aa, aaa, \dots$  and so on.

$$\text{So, } L = \{ a, aa, aaa, aaaa, \dots \}$$

### 2.4.4 Derivation Tree (Parse tree)

The derivation can be shown in the form of a tree. Such trees are called derivation or parse trees. The leftmost derivation as well as the right most derivation can be represented using derivation trees. Now, let us see “**What is derivation tree or parse tree?**” The derivation tree can be defined as shown below.

**Definition:** Let  $G = (V, T, P, S)$  be a CFG. The tree is derivation tree (parse tree) with the following properties.

1. The root has the label  $S$ .
2. Every vertex has a label which is in  $(V \cup T \cup \epsilon)$ .
3. Every leaf node has label from  $T$  and an interior vertex has a label from  $V$ .
4. If a vertex is labeled  $A$  and if  $X_1, X_2, X_3, \dots, X_n$  are all children of  $A$  from left, then  $A \rightarrow X_1 X_2 X_3 \dots X_n$  must be a production in  $P$ .

For example, consider the following grammar and its rightmost derivation along with parse tree:

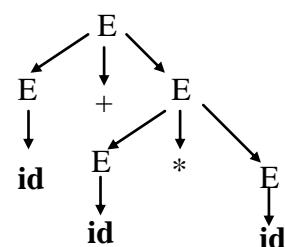
#### Grammar

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E * E \\ E &\rightarrow (E) \\ E &\rightarrow \text{id} \end{aligned}$$

#### rightmost derivation

$$\begin{aligned} E &\xrightarrow{rm} E + E \\ &\Rightarrow E + E * E \\ &\Rightarrow E + E * \text{id} \\ &\Rightarrow E + \text{id} * \text{id} \\ &\Rightarrow \text{id} + \text{id} * \text{id} \end{aligned}$$

#### Parse tree



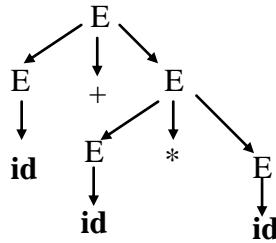
## 2.8 □ Syntax Analyzer

---

Now, let us see “What is the yield of the tree?” The *yield* of a tree can be formally defined as follows:

**Definition:** The *yield* of a tree is the string of symbols obtained by only reading the leaves of the tree from *left to right* without considering the  $\epsilon$ -symbols. The yield of the tree is derived always from the root and the yield of the tree is always a terminal string.

For example, consider the derivation tree (or parse tree) shown below:



If we read only the terminal symbols in the above parse tree from left to right we get **id + id \* id** and **id + id \* id** is the yield of the given parse tree.

## 2.5 Ambiguous grammar

In this section, let us see “What is ambiguous grammar?”

**Definition:** Let  $G = (V, T, P, S)$  be a context free grammar. A grammar  $G$  is *ambiguous* if and only if there exists at least one string  $w \in T^*$  for which two or more left derivations exist or two or more right derivations exist. That is, the ambiguous grammar has two or more meanings or interpretations.

Since, for every derivation a parse tree exist, the *ambiguous grammar* can also be defined as the one which has two or more different parse trees for the string  $w$  derived from start symbol  $S$ .

---

**Example 2.3:** Consider the following grammar from which an arithmetic expression can be obtained:

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E - E \\ E &\rightarrow E * E \\ E &\rightarrow E / E \\ E &\rightarrow (E) | I \\ I &\rightarrow \text{id} \end{aligned}$$

Show that the grammar is ambiguous.

---

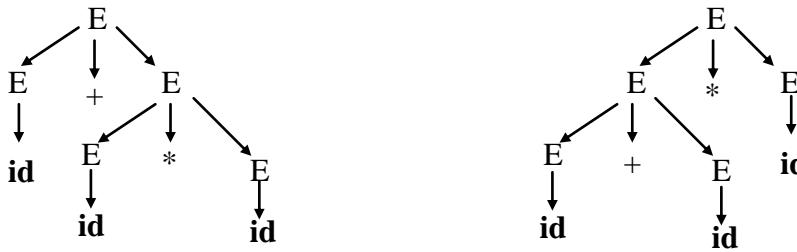
## □ Systematic approach to Compiler Design - 2.9

**Solution:** The sentence **id + id \* id** can be obtained from leftmost derivation in two ways as shown below.

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow \mathbf{id} + E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned}$$

$$\begin{aligned} E &\Rightarrow E * E \\ &\Rightarrow E + E * E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned}$$

The corresponding derivation trees for the two leftmost derivations are shown below:



Since the two parse trees are different for the same sentence **id + id \* id** by applying leftmost derivation, the grammar is ambiguous.

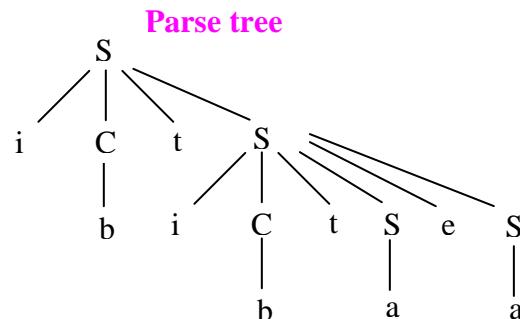
**Example 2.4:** Is the following grammar ambiguous? (if-statement or if-then-else)

$$\begin{aligned} S &\rightarrow iCtS \mid iCtSeS \mid a \\ C &\rightarrow b \end{aligned}$$

The string **ibtibtaea** can be obtained by applying the leftmost derivation as shown below along with parse.

### Leftmost derivation

$$\begin{aligned} S &\Rightarrow iCtS \\ &\Rightarrow ibtS \\ &\Rightarrow ibtiCtSeS \\ &\Rightarrow ibtibtSeS \\ &\Rightarrow ibtibtaeS \\ &\Rightarrow ibtibtaea \end{aligned}$$



The string **ibtibtaea** can be obtained again by applying the leftmost derivation but using different sets of productions as shown below along with parse tree.

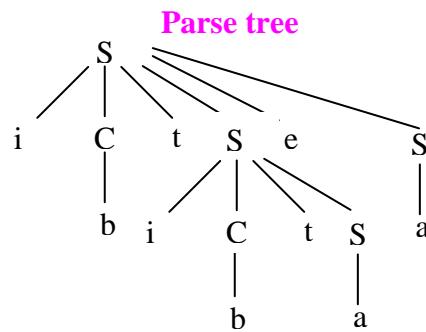
**Note:** i –if, t –then, e – else , b – other statement, a – other statement

## 2.10 □ Syntax Analyzer

---

### Leftmost derivation

$S \Rightarrow iCtSeS$   
 $\Rightarrow ibtSeS$   
 $\Rightarrow ibtiCtSeS$   
 $\Rightarrow ibtibtSeS$   
 $\Rightarrow ibtibtaeS$   
 $\Rightarrow ibtibtaea$



Since there are two different parse trees for the string ‘ibtibtaea’ by applying leftmost derivation the given grammar is ambiguous. The grammar has two interpretations or two meanings.

### 2.6 Eliminating ambiguity

Some grammars that are ambiguous can be converted into unambiguous grammars. This can be done using two methods:

- ♦ Dis-ambiguity rule
- ♦ Using precedence and associativity of operators

#### 2.6.1 Dis-ambiguity rule

We have already seen in the previous problem that the grammar corresponding to if-statement is ambiguous. This is due to dangling-else. The dangling else problem can be eliminated and thus ambiguity of the grammar can also be eliminated.

Now, let us see “What is dangling else problem?” Consider the following grammar:

$$\begin{aligned} S &\rightarrow iCtS \mid iCtSeS \mid a \\ C &\rightarrow b \end{aligned}$$

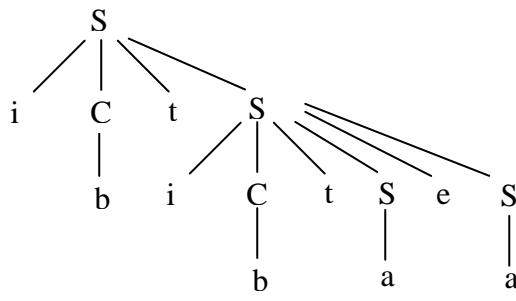
where

- ♦ i stands for keyword **if**
- ♦ C stands for **Condition** to be satisfied. Here C is a non-terminal
- ♦ t stands for keyword **then**
- ♦ S stands **statement** for non-terminal
- ♦ e stands for keyword **else**
- ♦ a stands for other statement
- ♦ b stands for other statement

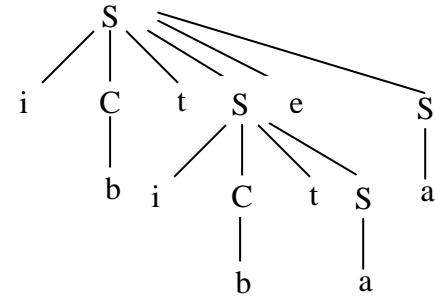
Since the above grammar is ambiguous, we get two different parse trees for the string **ibtibtaea** (Look at solution for previous problem for details) as shown below:

## □ Systematic approach to Compiler Design - 2.11

---



**Parse tree 1**



**Parse tree 2**

Since there are two parse trees for the same string **ibtibtaea** the given grammar is ambiguous. Observe the following points:

- ♦ The first parse tree associates **else** with 2<sup>nd</sup> if-statement
- ♦ The second parse tree associates **else** with first if-statement.

This ambiguity whether to associate **else** with first if-statement or second if-statement is called “**dangling else problem**”.

Now, let us see “**How dangling else problem can be solved?**” The dangling else problem can be solved by constructing unambiguous grammar as shown below:

---

**Example 2.5:** Eliminate ambiguity from the following ambiguous grammar:

$$\begin{aligned} S &\rightarrow iCtS \mid iCtSeS \mid a \\ C &\rightarrow b \end{aligned}$$


---

**Solution:** In all programming languages when if-statements are nested, the first parse tree is preferred. So, the general rule is “Match each **else** with closest unmatched **then**”. This rule can be directly incorporated into grammar and ambiguity can be eliminated as shown below:

**Step 1:** The matched statement **M** is an if-else statement where the statement **S** before **else** and after **else** keyword is matched. This can be expressed as:

$$M \rightarrow i C t M e M$$

**Step 2:** An unmatched statement **U** is the one consisting of:

- simple if-statement where the statement **S** is matched statement or unmatched statement. The equivalent production is:

$$U \rightarrow i C t S$$

- if-else statement where the statement before **else** is matched and statement after **else** is unmatched. The equivalent production is:

$$U \rightarrow i C t M e U$$

## 2.12 □ Syntax Analyzer

---

**Step 3:** The matched statement **M** and un-matched statement **U** can obtained using the statement **S** as shown below:

$$S \rightarrow M \mid U$$

So, the final grammar which is un-ambiguous is shown below:

$$\begin{array}{l} S \rightarrow M \mid U \\ M \rightarrow i C t M e M \\ U \rightarrow i C t S \\ U \rightarrow i C t M e U \end{array}$$

Observe that the above grammar associates **else** with closest **then** and eliminates ambiguity from the grammar.

### 2.6.2. Eliminating ambiguity using precedence and associativity

This method is explained using the following example:

---

**Example 2.6:** Convert the following ambiguous grammar into unambiguous grammar

$$\begin{array}{l} E \rightarrow E * E \mid E - E \\ E \rightarrow E ^ E \mid E / E \\ E \rightarrow E + E \\ E \rightarrow (E) \mid id \end{array}$$

The grammar can be converted into unambiguous grammar using the precedence of operators as well as associativity operators as shown below:

**Step 1:** Arrange the operators in increasing order of the precedence along with associativity as shown below:

Operators	Associativity	non-terminal used
+ , -	LEFT	E
*, /	LEFT	T
^	RIGHT	P

Since there are three levels of precedence, we associate three non-terminals: E, T and P. Also an extra non-terminal F, generating basic units in an arithmetic expression.

**Step 2:** The basic units in expression are **id** (identifier) and parenthesized expressions. The production corresponding to this can be written as:

$$F \rightarrow (E) \mid id$$

## ■ Systematic approach to Compiler Design - 2.13

---

**Step 3:** The next highest priority operator is  $\wedge$  and it is right associative. So, the production must start from the non-terminal P and it should have right recursion as shown below:

$$P \rightarrow F^\wedge P \mid F$$

**Step 4:** The next highest priority operators are  $*$  and  $/$  and they are left associative. So, the production must start from the non-terminal T and it should have left recursion as shown below:

$$T \rightarrow T * P \mid T / P \mid P$$

**Step 5:** The next highest priority operators are  $+$  and  $-$  and they are left associative. So, the production must start from the non-terminal E and it should have left recursion as shown below:

$$E \rightarrow E + T \mid E - T \mid T$$

**Step 6:** The final grammar which is unambiguous can be written as shown below:

$$\begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * P \mid T / P \mid P \\ P &\rightarrow F^\wedge P \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

---

**Example 2.7:** Convert the following ambiguous grammar into unambiguous grammar

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E - E \\ E &\rightarrow E ^ E \\ E &\rightarrow E * E \\ E &\rightarrow E / E \\ E &\rightarrow (E) \mid \text{id} \end{aligned}$$

by considering  $*$  and  $-$  operators lowest priority and they are left associative,  $/$  and  $+$  operators have the highest priority and are right associative and  $^$  operator has precedence in between and it is left associative.

---

The grammar can be converted into unambiguous grammar using the precedence of operators as well as associativity operators as shown below:

## 2.14 □ Syntax Analyzer

---

**Step 1:** Arrange the operators in increasing order of the precedence along with associativity as shown below:

Precedence	Operators	Associativity	non-terminal used
(lowest)	*	LEFT	E
	,	LEFT	P
(highest)	^	RIGHT	T

Since there are three levels of precedence we associate three non-terminals: E, P and T. Also use an extra non-terminal F generating basic units in an arithmetic expression.

**Step 2:** The basic units in expression are **id** (identifier) and parenthesized expressions. The production corresponding to this can be written as:

$$F \rightarrow (E) \mid \text{id}$$

**Step 3:** The next highest priority operators are + and / and they are right associative. So, the production must start from the non-terminal T and it should be right recursive in RHS of the production as shown below:

$$T \rightarrow F + T \mid F / T \mid F$$

**Step 4:** The next highest priority operator is ^ and it is left associative. So, the production must start from the non-terminal P and it should be left recursive in RHS of the production as shown below:

$$P \rightarrow P ^ T \mid T$$

**Step 5:** The next highest priority operators are \* and – and they are left associative. So, the production must start from the non-terminal E and it should be left recursive in RHS of the production as shown below:

$$E \rightarrow E + P \mid E - P \mid P$$

**Step 6:** The final grammar which is unambiguous can be written as shown below:

$$\boxed{\begin{aligned} E &\rightarrow E + P \mid E - P \mid P \\ P &\rightarrow P ^ T \mid T \\ T &\rightarrow F + T \mid F / T \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}}$$

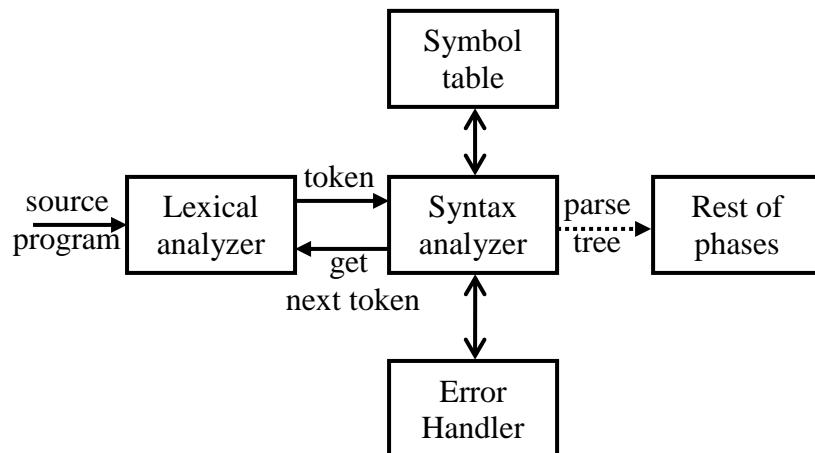
## ■ Systematic approach to Compiler Design - 2.15

### 2.7 The Role of the Parser

First, let us see “What is parsing?”

**Definition:** **Parsing** is the process of getting tokens from the lexical analyzer and obtains a derivation for the sequence of tokens and builds a parse tree. Thus, if the program is syntactically correct, the parse tree is generated. If a derivation for the sequence of tokens does not exist i.e., if the program is syntactically wrong, it results in syntax error and the parser displays the appropriate error messages. The parse trees are very important in figuring out the meaning of a program or part of the program. The parse tree is also called syntax tree. **Parser** also called syntax analyzer is the one which does parsing.

The block diagram that shows the interaction of parser with other modules and phases is shown below:



The role of the parser or the various activities that are performed by the parser are shown below:

- ♦ Parser reads sequence of tokens from the lexical analyzer
- ♦ The parser checks whether the tokens obtained from lexical analyzer can be successfully generated. This is done by obtaining a derivation for the sequence of tokens and builds the parse tree.
- ♦ If a derivation is obtained using the sequence of tokens it indicates that program is syntactically correct and the parse tree is generated.
- ♦ If a derivation is not obtained using the sequence of tokens it indicates that program is syntactically wrong and the parse tree is not generated. Now, the parser reports appropriate error messages clearly and accurately along with line numbers
- ♦ Parser also recovers from each error quickly so that subsequent errors can be detected and displayed so that the user can correct the programs.

## 2.16 □ Syntax Analyzer

---

### 2.7.1 Error Recovery strategies

The following activities are performed whenever errors are detected by the parser:

- ◆ Detect the syntax errors accurately and produce appropriate error messages so that the programmer can correct the program.
- ◆ It has to recover from the errors quickly and detect subsequent errors in the program.
- ◆ Error handler should take all the actions very fast and should not slowdown the compilation process.

Now, let us see “What are error recovery strategies of the parser (or syntax analyzer)?”

The various error recovery techniques are:

- ◆ Panic mode recovery
- ◆ Error productions
- ◆ Phrase level recovery
- ◆ Global correction

**Panic Mode Recovery:** It is the simplest and most popular error recovery method. When an error is detected, the parser discards symbols one at a time until next valid token (called synchronizing token) is found. The typical synchronizing tokens are:

- ◆ statement terminators such as semicolon
- ◆ Expression terminators such as \n

It often skips a considerable amount of input without checking it for additional errors. Once the synchronizing token found, the parser will continue from that point onwards to identify the subsequent errors. In situations where multiple errors are in the same statement, this method is not useful.

For example, consider the erroneous expression:

(5 \*\* 2) +8

- ◆ The parser scans the input from left to right and finds no mistake after reading (, 5 and \*.
- ◆ After reading the second \*, it knows that no expression has consecutive \* operators and it displays an error “Extra \* in the input”
- ◆ Now, it has to recover from the error. In panic mode recovery, it skips all input symbols till the next integer 2 is encountered. Here, 2 is the synchronizing token.
- ◆ Thus, error is detected and recovered from the error in panic mode recovery

**Error Productions:** In this type of error recovery strategy, we introduce error productions. The error productions specify commonly known mistakes in the grammar. When we implement the parser, when an error production is used, it displays the appropriate error message. For example, consider the expression  $10x$ . Mathematically it means multiply 10 with  $x$ . But, in a programming language we should write  $10*x$ . Such errors can be identified very easily by incorporating error productions and within the body of the function, display appropriate error messages.

### Disadvantages

- ♦ Can resolve many errors, but not all potential errors.
- ♦ The introduction of error productions will complicate the grammar

**Phrase Level Recovery:** It is an error correcting method. On discovering an error, a parser may perform local correction on the remaining input. This is normally done by inserting, deleting or/and replacing the input and enable the parser to continue parsing.

**Ex:** Replacing a comma by a semicolon, deleting an extraneous semicolon, or inserting a missing semicolon.

### Disadvantages

- ♦ Very difficult to implement
- ♦ Slows down the parsing of correct programs
- ♦ Proper care must be taken while choosing replacements as they may lead to infinite loops.

**Global Correction:** This is also one of the error correction strategies. The various points to remember in this error correction method are:

- ♦ These methods replace incorrect input with correct input using least-cost-correction algorithms.
- ♦ These algorithms take an incorrect input string  $x$  and grammar  $G$ , and find a parse tree for a related string  $y$ , such that the number of insertions, deletions and changes of tokens required to transform  $x$  into  $y$  is as small as possible.
- ♦ These methods are costly to implement in terms of time and space and hence are only of theoretical interest.

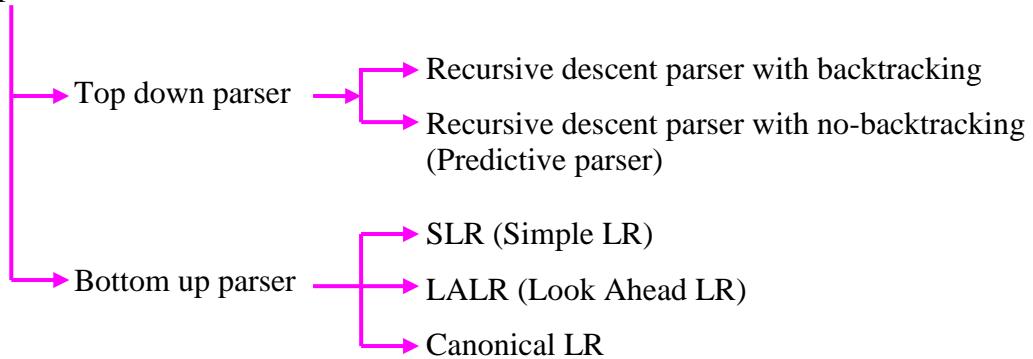
### 2.7.2 Parsing techniques

In this section, let us see “What are the different types of parsers?”

## 2.18 □ Syntax Analyzer

---

The parsers are classified as shown below:



### 2.8 Top-down Parsing

Now, let us see “What is top down parser?”

**Definition:** The process of constructing a parse tree for the string of tokens (obtained from the lexical analyzer) from top i.e., starting from the root node and creating the nodes of the parse tree in preorder in depth-first-search manner is called **top down parsing technique**. Thus, top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string and constructing the parse tree for that derivation. The parser that uses this approach is called **top down parser**. Since the parsing starts from top (i.e., root) down to the leaves, it is called **top down parser**.

---

**Example 2.8:** Show the top-down parsing process for the string **id + id \* id** for the grammar

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E * E \\ E &\rightarrow (E) \\ E &\rightarrow \text{id} \end{aligned}$$

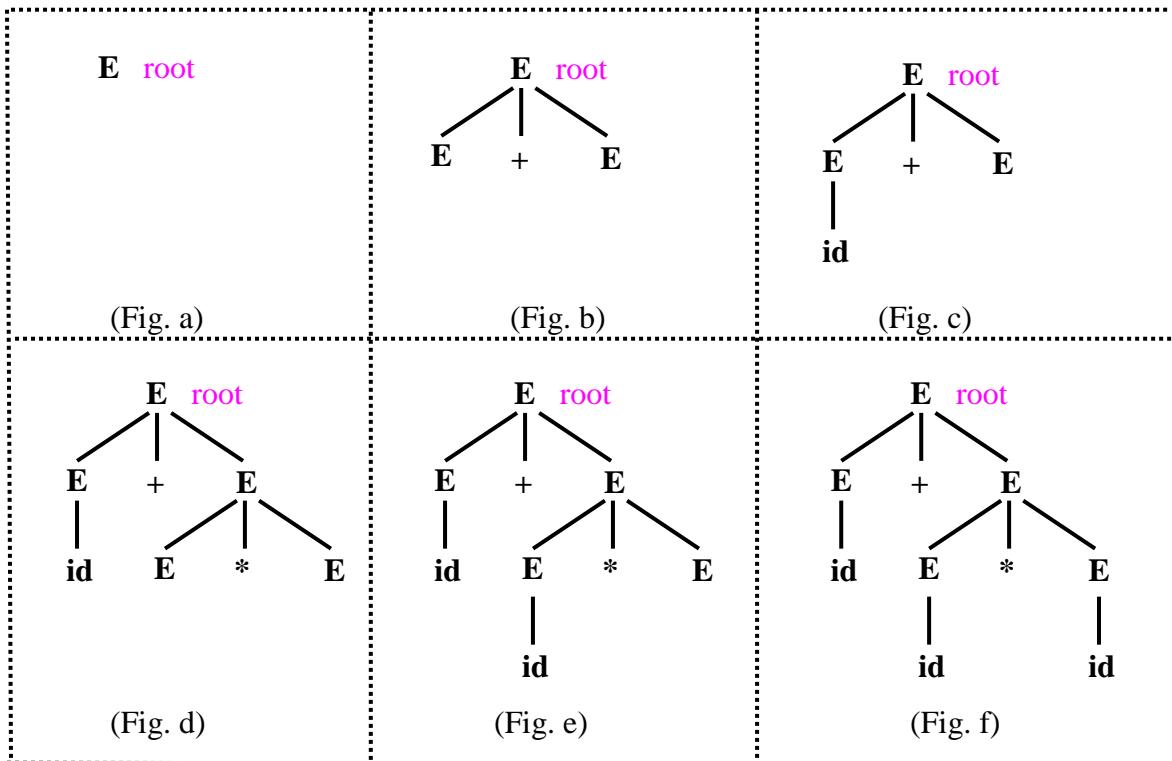
---

**Solution:** The string **id + id \* id** can be obtained by the grammar by applying leftmost derivation as shown below:

$$\begin{array}{ll} E & \xrightarrow[lm]{\Rightarrow} E + E & (\text{fig a}) \text{ step 1} \\ & \Rightarrow \text{id} + E & (\text{fig b}) \text{ step 2} \\ & \Rightarrow \text{id} + E * E & (\text{fig c}) \text{ step 3} \\ & \Rightarrow \text{id} + \text{id} * E & (\text{fig d}) \text{ step 4} \\ & \Rightarrow \text{id} + \text{id} * \text{id} & (\text{fig e}) \text{ step 5} \end{array}$$

## ❑ Systematic approach to Compiler Design - 2.19

The above derivation can be written in the form of a parse tree from the start symbol using top-down approach as shown below:



**Initial:** Start from root node E

**Step 1:** Replace E with E + E using  $E \rightarrow E + E$ . It is shown in figure (b)

**Step 2:** Replace E with **id** using  $E \rightarrow id$ . It is shown in figure (c).

**Step 3:** Replace E with E \* E using  $E \rightarrow E * E$ . It is shown in figure (d).

**Step 4:** Replace E with **id** using  $E \rightarrow id$ . It is shown in figure (e).

**Step 5:** Replace E with **id** using  $E \rightarrow id$ . It is shown in figure (f).

### 2.8.1 Recursive descent parser

Now, let us see “What is recursive descent parser?”

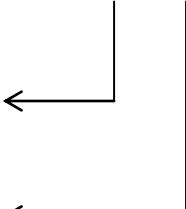
## 2.20 □ Syntax Analyzer

---

**Definition:** A recursive descent parser is a top down parser in which parse tree is constructed from the top starting from root node and selecting the productions from left to right (if two or more alternative productions exists). For every non-terminal there exists a recursive procedure and the right hand of the production of that non-terminal is implemented as the body of the procedure. The sequence of terminals and non-terminals on the right hand side of the production correspond to matching with input symbols and calls to other procedures while selecting the alternate production is implemented using switch or if-statements. Thus, the syntax or structure of the resulting program closely mirrors that of the grammar it recognizes.

For example, the procedure for the production  $A \rightarrow \alpha$  can be written as shown below:

```
procedure A ()// Function header
{
    .....
    .....
    .....
}
```



Observe the following points:

- ♦ For the variable  $A$  on the left hand side of the production, we write the function header
- ♦ For the string of grammar symbols denoted by  $\alpha$  on the right hand side of the production corresponds to the function body.
- ♦ Thus, for every non-terminal in the grammar we write the procedure or function as in previous two steps.

**Working:** The recursive descent parser works as shown below:

- ♦ Execution starts from the function that corresponds to the start symbol of the grammar
- ♦ If the body of the function scans the entire input string, then parsing is successful. Otherwise, the input string is not proper and parsing halts.
- ♦ Each non-terminal is associated with a parsing procedure or a function that can recognize any sequence of tokens generated by that non-terminal
- ♦ Within the function, both non-terminals and terminals are matched
- ♦ To match the non-terminal  $A$ , we call the function/procedure  $A$  which corresponds to non-terminal  $A$ . These calls may be recursive and hence the name recursive descent parser.

## ■ Systematic approach to Compiler Design - 2.21

- ♦ To match the terminal  $a$ , we compare current input symbol with  $a$ . If there is match, it is syntactically correct and we increment the input pointer and get the next token
- ♦ If the current input symbol is not  $a$ , it is syntactically wrong and appropriate error message is displayed
- ♦ Some error correction may be done to recover from each error quickly so that subsequent errors can be detected and displayed so that the user can correct the programs.

The general procedure for a recursive-descent parsing that uses top-down parser is shown below:

---

**Example 2.9:** Algorithm for recursive descent parser (backtracking is not supported)

---

```
procedure A()          //  $A \rightarrow X_1X_2X_3\dots\dots X_k$ 
{
    for i = 1 to k do
        if ( $X_i$  is a non-terminal)
            call procedure  $X_i()$ ;
        else if ( $X_i$  is same as current input symbol  $a$ )
            advance the input to the next symbol
        else
            error();
    end for
}
```

Now, let us write the recursive parsers for some of the grammars.

---

**Example 2.10:** Write the recursive descent parser for the following grammar

---

$E \rightarrow T$   
 $T \rightarrow F$   
 $F \rightarrow (E) \mid id$

```
// function corresponding to the production E → T
procedure E()
{
    T();
}
```

## 2.22 □ Syntax Analyzer

---

```
// function corresponding to the production T → F
procedure T()
{
    F();
}

// function corresponding to the production F → (E) | id
//                                         F → (   E   )   |   id
procedure F() <----->
{
    if (input_symbol == '(') <----->

        advance input pointer
        E();
        if (input_symbol == ')')
            advance input pointer
        else
            error()
        end if

    else if (input_symbol == id)
        advance input pointer
    else
        error();
    end if
}
```

Now, let us see “What are the different types of recursive descent parsers?” The recursive descent parser can be classified into two types:

- ♦ Recursive descent parser with backtracking
- ♦ Recursive descent parser without backtracking (predictive parser)

### 2.8.2 Recursive descent parser with backtracking

Now, let us see “What is the need for backtracking in recursive descent parser?” The backtracking is necessary for the following reasons:

- ♦ During parsing, the productions are applied one by one. But, if two or more alternative productions are there, they are applied in order from left to right one at a time.

## □ Systematic approach to Compiler Design - 2.23

- ♦ When a particular production applied fails to expand the non-terminal properly, we have to apply the alternate production. Before trying alternate production, it is necessary undo the activities done using the current production. This is possibly only using backtracking.

But, the recursive descent parsers with backtracking are not frequently used. So, we just concentrate on how they work with example.

---

**Example 2.11:** Show the steps involved in recursive descent parser with backtracking for the input string *cad* for the following grammar

$$\begin{aligned} S &\rightarrow cAd \\ A &\rightarrow ab \mid a \end{aligned}$$

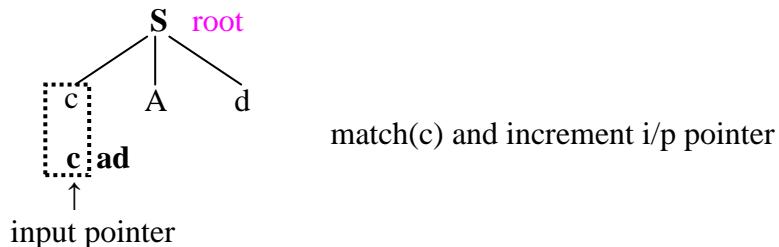

---

**Solution:** The three parts that are used while parsing the string are:

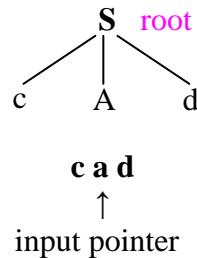
<u>Given grammar</u>	<u>String to be parsed</u>	<u>Parse tree</u>
$\begin{aligned} S &\rightarrow cAd \\ A &\rightarrow ab \mid a \end{aligned}$	$\begin{array}{c} \text{cad} \\ \uparrow \\ \text{input pointer} \end{array}$	$\begin{array}{c} \text{S } (\text{root}) \\   \\ \text{c} \quad \text{A} \quad \text{d} \end{array}$

Observe that input-pointer points to the next character to be read.

**Step 1:** The only unexplored node is *S* and we apply the production  $S \rightarrow cAd$  to expand the non-terminal *S* as shown below:

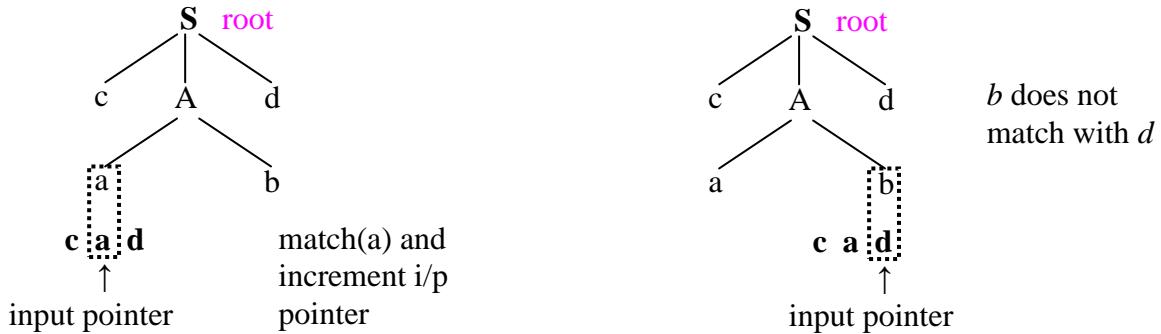


**Step 2:** Now, the next node to be expanded is *A* and input pointer points to *a* as shown below:

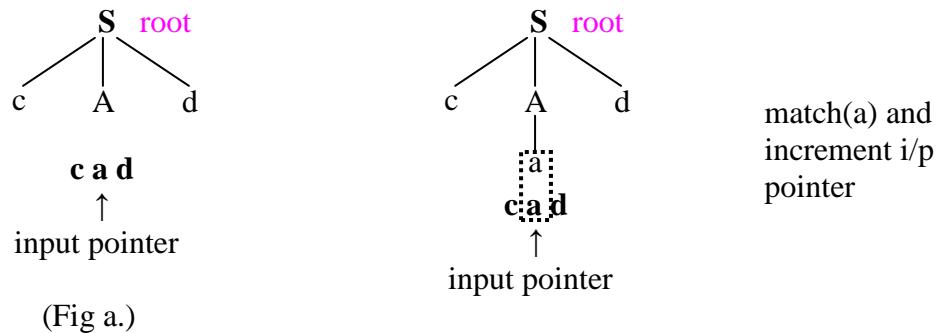


## 2.24 □ Syntax Analyzer

**Step 3:** Since two productions are there from A, the first production  $A \rightarrow ab$  is selected and non-terminal A is expanded as shown below:



**Step 4:** Observe that by selecting  $A \rightarrow ab$  the input string is not matched. So, we have to reset the pointer to input symbol  $a$  (so as to get the parse tree shown in step 2). This is done using backtracking and is shown in (fig a). After backtracking try expanding A using the second production  $A \rightarrow a$  and then proceed comparing as shown in (fig b).



**Step 5:** Now, the next symbol  $d$  in grammar is compared with  $d$  in the input and they match. Finally, we halt and announce successful completion of parsing.

Now, let us see “For what type of grammars recursive descent parser cannot be constructed? What is the solution?”

The recursive descent parser cannot be constructed for a grammars having:

- ♦ Ambiguity. The solution is to eliminate ambiguity from the grammar.
- ♦ Left recursion. The solution is to eliminate left recursion from the grammar.
- ♦ Two or more alternatives having a common prefix. The solution is to left factor the grammar.

### 2.8.3 Left recursion

Now, let us see “What is left recursion? What problems are encountered if a recursive descent parser is constructed for a grammar having left recursion?”

**Definition:** A grammar G is said to be left recursive if it has non-terminal A such that there is a derivation of the form:

$$A \xrightarrow{+} A\alpha \quad (\text{Obtained by applying one or more productions})$$

where  $\alpha$  is string of terminals and non-terminals. That is, whenever the first symbol in a partial derivation is same as the symbol from which this partial derivation is obtained, then the grammar is said to be **left-recursive** grammar. A grammar may have:

- ♦ immediate left recursion
- ♦ indirect left recursion

**Immediate left recursion:** A grammar G is said to have immediate left recursion if it has a production of the form:

$$A \rightarrow A\alpha$$

For example, consider the following grammar:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

In the above grammar consider the first two productions:

$$\begin{aligned} E &\rightarrow E + T \\ T &\rightarrow T * F \end{aligned}$$

Observe that in the above two productions, the first symbol on the right hand side of the production is same as the symbol on the left hand side of the production. So, the given grammar has immediate left recursion in two productions.

**Indirect left recursion:** A left recursion involving derivations of *two or more steps* so that the first symbol on the right hand side of the partial derivation is same as the symbol from which the derivation started is called **indirect left recursion**. For example, consider the following grammar:

$$\begin{aligned} E &\rightarrow T \\ T &\rightarrow F \\ F &\rightarrow E + T \mid \text{id} \end{aligned}$$

Consider the following derivation:

$$E \Rightarrow T \Rightarrow F \Rightarrow E + T$$

## 2.26 □ Syntax Analyzer

---

In the above derivation, note that in the partial derivation to get the string E + T, the first symbol is E which is same as the symbol from which the derivation started. But, the string E+T is obtained from E by applying two or more productions. So, the given grammar even though it is not having immediate left recursion, it has indirect left recursion and hence the grammar is left recursive.

Now, let us write the recursive descent parser for the grammar having left recursion.

---

**Example 2.12:** Consider the production  $E \rightarrow E + T$ . Write the recursive descent parser.

---

**Solution:** The recursive descent parser for the production  $E \rightarrow E + T$  can be written as shown below:

```
// function corresponding to the production E → E + T
procedure E()
{
    E();
    if (input_symbol = '+')
        advance input pointer
    else
        error
    end if
    T();
}
```

Now, let us see “What is the problem in constructing recursive descent parser for the grammar having left recursion?” Observe the following points (with respect to the above procedure which has left recursion)

- ♦ When a procedure is invoked, the parameter values along with return address will be pushed on to the stack and hence stack size decreases
- ♦ The procedure E() is called recursively infinitely without consuming any input and hence the size of the stack grows very fast and stack will be full soon.
- ♦ Since there is no space left on the stack to push parameter values and return address, the system crashes.
- ♦ So, the recursive descent parser that is built using left-recursive grammar can cause a parser to go into an infinite loop eventually crashing the system and hence the left recursive grammar is not suitable for recursive descent parser. Hence, we have to eliminate left recursion from the grammar and then parse the string.

## ❑ Systematic approach to Compiler Design - 2.27

### 2.8.4 Procedure to eliminate left recursion

Consider the production of the form:

$$A \rightarrow A\alpha \mid \beta$$

where  $\beta$  do not start with  $A$ . Note that the above grammar has left recursion. Now, let us see how to eliminate left recursion. The various strings that can be generated by above grammar are shown below:

Derivation: 1 <sup>st</sup>	2 <sup>nd</sup>	3 <sup>rd</sup>	4 <sup>th</sup> and so on.
$A \Rightarrow \beta$	$A \Rightarrow A\alpha$ $\Rightarrow \beta \alpha$	$A \Rightarrow A\alpha$ $\Rightarrow A\alpha\alpha$ $\Rightarrow \beta \alpha\alpha$	$A \Rightarrow A\alpha$ $\Rightarrow A\alpha\alpha$ $\Rightarrow A\alpha\alpha\alpha$ $\Rightarrow \beta \alpha\alpha\alpha \dots \}$
↓	↓	↓	↓
$\{ \beta, \beta \alpha, \beta \alpha\alpha, \beta \alpha\alpha\alpha \dots \}$			

Observe from above derivations that the language  $L$  consists of  $\beta$  followed zero or more  $\alpha$ 's. The same language can be represented and generated using different grammar as shown below:

$$\begin{aligned} L &= \{ \beta \alpha^i \mid i \geq 0 \} \\ &\quad \downarrow \downarrow \\ A &\rightarrow \beta A' \quad \text{where } A' \text{ should generate zero or more } \alpha \text{'s} \end{aligned}$$

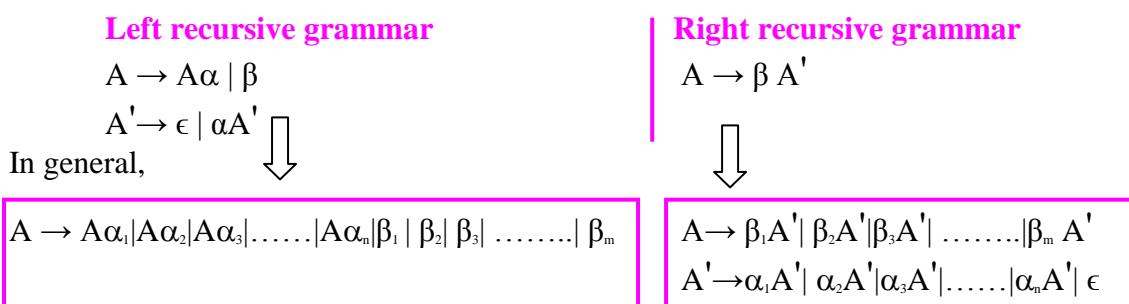
From  $A'$  we can get zero or more  $\alpha$ 's using the following productions:

$$A' \rightarrow \epsilon \mid \alpha A'$$

So the final grammar that generate  $\beta$  followed by zero or more  $\alpha$ 's which do not have left recursion is shown below:

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \epsilon \mid \alpha A' \end{aligned}$$

Thus, the grammar which has left recursion can be written in the form of another grammar that does not have left recursion as shown below:



## 2.28 □ Syntax Analyzer

**Example 2.13:** Eliminate left recursion from the following grammar

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$

**Solution:** The given grammar is shown below:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$

Since the first symbol on the right hand side of E-production and T-production is same as the symbol on the left hand side of the production, the grammar has immediate left recursion. Now, the immediate left recursion can be eliminated from the grammar as shown below:

Left recursive productions	Right recursive productions
$A \rightarrow A\alpha_1   A\alpha_2   A\alpha_3   \dots   A\alpha_n   \beta_1   \beta_2   \beta_3   \dots   \beta_m$	$A \rightarrow \beta_1 A'   \beta_2 A'   \beta_3 A'   \dots   \beta_m A'$ $A' \rightarrow \alpha_1 A'   \alpha_2 A'   \alpha_3 A'   \dots   \alpha_n A'   \epsilon$
1) $E \rightarrow E + \underbrace{T}_{\downarrow} \mid T$ $\downarrow \quad \downarrow \quad \downarrow \quad \downarrow$ $A \rightarrow A \alpha_1 \mid \beta_1$	$E \rightarrow TE'$ $E' \rightarrow +TE' \mid \epsilon$
2) $T \rightarrow T * \underbrace{F}_{\downarrow} \mid F$ $\downarrow \quad \downarrow \quad \downarrow \quad \downarrow$ $A \rightarrow A \alpha_1 \mid \beta_1$	$T \rightarrow FT'$ $T' \rightarrow *FT' \mid \epsilon$
3) $F \rightarrow (E) \mid id$	$F \rightarrow (E) \mid id$

The final grammar obtained after eliminating left recursion can be written as shown below:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid id \end{aligned}$$

## ■ Systematic approach to Compiler Design - 2.29

Now, we can write the recursive descent parser for the above grammar which is obtained after eliminating left recursion.

**Example 2.14:** Write the recursive descent parser for the following grammar:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid id \end{aligned}$$

The recursive descent parser for the above grammar is shown below. Note that for each non-terminal there is a procedure and the right hand side of the production is implemented as the body of the procedure as shown below:

```
// function corresponding to the production: E → T E'  
procedure E()  
{  
    T();  
    EDASH();  
}  
  
// function corresponding to the production: E' → + T E' | ε  
procedure EDASH()  
{  
    if ( inputsymbol == '+' )  
    {  
        Advance input pointer  
        T();  
        EDASH();  
    }  
}  
  
// function corresponding to the production: T → F T'  
procedure T()  
{  
    F();  
    TDASH();  
}
```

## 2.30 □ Syntax Analyzer

---

```

// function corresponding to the production: T' → * F T' | ε
procedure TDASH() ←-----[ ]-----[ ]-----[ ]
{
    if( inputsymbol == '*' ) ←-----[ ]-----[ ]
    {
        advance input pointer
        F(); ←-----[ ]-----[ ]
        TDASH(); ←-----[ ]-----[ ]
    }
}

// function corresponding to the production: F → ( E ) | id
procedure F() ←-----[ ]-----[ ]-----[ ]
{
    if( inputsymbol == '(' ) ←-----[ ]-----[ ]
    {
        advance input pointer
        E(); ←-----[ ]-----[ ]
        if( inputsymbol == ')' ) ←-----[ ]-----[ ]
            advance input pointer
        else error();
    }
    else
    {
        if( inputsymbol == id ) ←-----[ ]-----[ ]
            advance input pointer
        else error();
    }
}

```

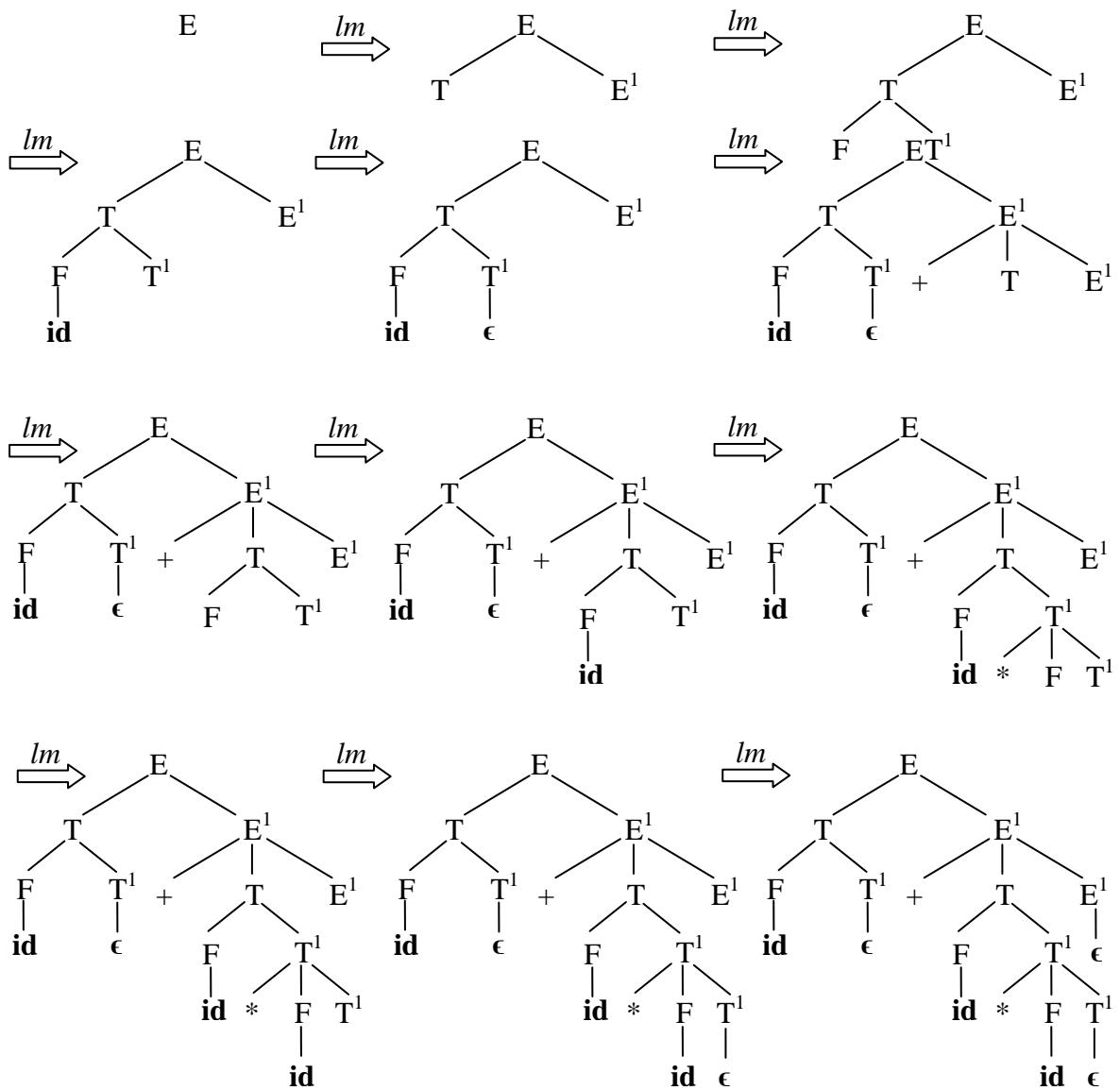
**Example 2.15:** Obtain top-down parse for the string **id+id\*id** for the following grammar

$$\begin{aligned}
 E &\rightarrow TE' \\
 E' &\rightarrow + TE' \mid \epsilon \\
 T &\rightarrow FT' \\
 T' &\rightarrow *FT' \mid \epsilon \\
 F &\rightarrow (E) \mid id
 \end{aligned}$$

The top-down parse for the string **id+id\*id** for the above grammar can be written as shown below:

## Systematic approach to Compiler Design - 2.31

---



**Example 2.16:** Eliminate left recursion from the following grammar:

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Sd \mid \epsilon \end{aligned}$$


---

**Solution:** Left recursion can be eliminated as shown below:

**Step 1:** The  $S$ -production does not have immediate left recursion. So, let us not consider the production  $S \rightarrow Aa \mid b$

## 2.32 □ Syntax Analyzer

**Step 2:** Consider the production:  $A \rightarrow Ac \mid Sd \mid \epsilon$ . Replacing the non-terminal  $S$  by the production  $S \rightarrow Aa \mid b$  we get following A-production:

$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$

**Step 3:** Now, the grammar obtained after eliminating indirect left recursion is shown below:

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Aad \mid bd \mid \epsilon \end{aligned}$$

Now, immediate left recursion can be eliminated as shown below:

Left recursive productions	Right recursive productions
$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \dots \mid \beta_m$	$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \dots \mid \beta_m A'$ $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \dots \mid \alpha_n A' \mid \epsilon$
1) $S \rightarrow Aa \mid b$	$S \rightarrow Aa \mid b$
2) $A \rightarrow A \overset{c}{\underset{\downarrow}{\mid}} c \mid A \overset{ad}{\underset{\downarrow}{\mid}} ad \mid \overset{bd}{\underset{\downarrow}{\mid}} bd \mid \overset{\epsilon}{\underset{\downarrow}{\mid}} \epsilon$ $A \rightarrow A \mid \alpha_1 \mid A \mid \alpha_2 \mid \mid \beta_1 \mid \beta_2$	$A \rightarrow bd A' \mid \epsilon A'$ $A' \rightarrow cA' \mid adA' \mid \epsilon$

So, the final grammar obtained after eliminating left recursion is shown below:

$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow bd A' \mid \epsilon A' \\ A' &\rightarrow cA' \mid adA' \mid \epsilon \end{aligned}$	$S \rightarrow Aa \mid b$ $A \rightarrow bd A' \mid A'$ $A' \rightarrow cA' \mid adA' \mid \epsilon$
$\left. \begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow bd A' \mid \epsilon A' \\ A' &\rightarrow cA' \mid adA' \mid \epsilon \end{aligned} \right\}$ can be written as	$A \rightarrow bd A' \mid A'$ $A' \rightarrow cA' \mid adA' \mid \epsilon$

Now, let us “Write the algorithm to eliminate left recursion” The algorithm to eliminate left recursion is shown below:

**Example 2.17:** Algorithm to eliminate left recursion (including indirect left recursion)

**Input :** Grammar  $G$  without  $\epsilon$ -productions and with no cycles

**Output:** Grammar without left recursion. It may have  $\epsilon$ -productions

Arrange the non-terminals in the order  $A_1, A_2, A_3, \dots, A_n$

## ■ Systematic approach to Compiler Design - 2.33

```
for i = 1 to n do
    for j = 1 to i-1 do
        Let Aj → β1 | β2 | β3 |..... βk
        Replace Ai → Ajα by Ai → β1α | β2α | β3α |..... βkα
    end for
    Eliminate immediate left recursion among Ai productions
end for
```

### 2.8.5 Left factoring

Now, let us see “What is left factoring? What is the need for left factoring?”

**Definition:** A grammar in which two or more productions from a non-terminal A do not have a common prefix of symbols on the right hand side of the A-productions is called **left factored grammar**. The left-factored grammar is suitable for top-down parser such as recursive descent parser with or without backtracking.

**Ex 1:** The grammar to generate string consisting of at least one ‘a’ followed by at least one ‘b’ can be written as shown below:

$$\begin{aligned} S &\rightarrow aAbB \\ A &\rightarrow aA \mid \epsilon \quad \text{Left factored grammar} \\ B &\rightarrow bB \mid \epsilon \end{aligned}$$

Observe the following points:

- ♦ The S-production has only one production and it cannot have common prefix on the right side of the production.
- ♦ The two A-productions do not have any common prefix on the right side.
- ♦ Finally two B-productions do not have any common prefix on the right side of that production

**Ex 2:** The grammar that generates string consisting of at least one ‘a’ followed by at least one ‘b’ can also be written as shown below:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aA \mid a \quad \text{Non Left factored grammar} \\ B &\rightarrow bB \mid b \end{aligned}$$

Observe the following points:

- ♦ The S-production has only one production and it is not having common prefix on the right side of the production.
- ♦ The two A-productions have a common prefix “a” on the right side of the production.

## 2.34 □ Syntax Analyzer

---

- ♦ The two B-productions have a common prefix “b” on the right side of the production.
- ♦ Since common prefix is present in both A-productions and B-productions, it is not left-factored grammar.

**Note:** If two or more productions starting from same non-terminal have a common prefix, the grammar is not left-factored.

Now, let us see “**What is the use of left factoring?**” Left factoring is must for top down parser such as recursive descent parser with backtracking or predictive parser which is also recursive descent parser without backtracking. This is because, if A-production has two or more alternate productions and they have a common prefix, then the parser has some confusion in selecting the appropriate production for expanding the non-terminal A

**Ex 1:** Consider the following grammar that recognizes the if-statement:

$$S \rightarrow \text{if } E \text{ then } S \text{ else } S \mid \text{if } E \text{ then } S$$

Observe the following points:

- Both productions starts with keyword **if**.
- So, when we get the input ‘**if**’ from the lexical analyzer, we cannot tell whether to use the first production or to use the second production to expand the non-terminal S.
- So, we have to transform the grammar so that they do not have any common prefix. That is, left factoring is must for parsing using top-down parser.

Now, the question is “**How to do left factoring?**” The left-factoring can be done as shown below:

- 1) Consider two A-productions with common prefix  $\alpha$ :

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

- 2) Let the input begins with string derived from  $\alpha$ . Since  $\alpha$  is the common prefix, we retain  $\alpha$  and we replace either  $\beta_1$  or  $\beta_2$  by the non-terminal  $A'$ . So, we can write the above production as:

$$A \rightarrow \alpha A'$$

where  $A'$  can produce either  $\beta_1$  or  $\beta_2$  using the production:

$$A' \rightarrow \beta_1 \mid \beta_2$$

## ■ Systematic approach to Compiler Design - 2.35

Now, after seeing the input derived from  $\alpha$ , we can expand  $A'$  either to  $\beta_1$  or to  $\beta_2$ . So, the given grammar is converted into left-factored grammar as shown below:

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2$$

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 | \beta_2$$

**Non left-factored grammar**

**Left-factored grammar**

Now, let us “**Write the algorithm for doing left-factoring**” The algorithm for doing left-factoring is shown below:

---

**Example 2.18:** The algorithm for left-factoring

---

**Algorithm** LEFT\_FACTOR(G)

**Input:** Grammar G

**Output:** An equivalent left-factored grammar

**Method:** The following procedure is used:

- 1) For each non-terminal  $A$ , find the longest prefix  $\alpha$  which is common to two or more of its alternatives.
- 2) If there is a production of the form:

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3 | \dots | \alpha\beta_n | \gamma$$

where  $\gamma$  do not start with  $\alpha$ , then the above A-production can be written as shown below:

$$\begin{aligned} A &\rightarrow \alpha A' | \gamma \\ A' &\rightarrow \beta_1 | \beta_2 | \beta_3 | \dots | \beta_n \end{aligned}$$

Here,  $A'$  is a new non-terminal.

- 3) Repeatedly apply the transformation in step 2 as long as two alternatives for a non-terminal have a common prefix
- 4) Return the final grammar which is left-factored

## 2.36 □ Syntax Analyzer

**Example 2.19:** Do the left-factoring for the following grammar:

$$\begin{aligned} S &\rightarrow iCtS \mid iCtSeS \mid a \\ C &\rightarrow b \end{aligned}$$

**Solution:** The given grammar is shown below:

$$\begin{aligned} S &\rightarrow iCtS \mid iCtSeS \mid a \\ C &\rightarrow b \end{aligned}$$

Since S-production has common prefix **iCtS** in more than one production, left factoring is necessary. Left factoring the above grammar can be done using the algorithm shown below:

Given productions	Left-factored productions
$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \dots \mid \alpha\beta_n \mid \gamma$	$A \rightarrow \alpha A' \mid \gamma$ $A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \dots \mid \beta_n$
1) $S \rightarrow iCtS \in \mid iCtS eS \mid a$ $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \gamma$	$S \rightarrow iCtSS' \mid a$ $S' \rightarrow \epsilon \mid eS$
2) $C \rightarrow b$	$C \rightarrow b$

So, the final grammar which is obtained after doing left-factoring is shown below:

$$\begin{aligned} S &\rightarrow iCtSS' \mid a \\ S' &\rightarrow \epsilon \mid eS \\ C &\rightarrow b \end{aligned}$$

## 2.8.6 Problems with top down parser

Now, let us “Briefly explain the problems associated with top-down parser?” (JUY-AUG-2009) The various problems associated with top down parser are:

- Ambiguity in the grammar
- Left recursion
- Non-left factored grammar
- Backtracking

## □ Systematic approach to Compiler Design - 2.37

- ♦ **Ambiguity in the grammar:** A grammar having two or more left most derivations or two or more right most derivations is called ambiguous grammar. For example, the following grammar is ambiguous:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid ( E ) \mid id$$

The ambiguous grammar is not suitable for top-down parser. So, ambiguity has to be eliminated from the grammar. (For details refer section 2.6)

- ♦ **Left-recursion:** A grammar G is said to be left recursive if it has non-terminal A such that there is a derivation of the form:

$$A \stackrel{+}{\Rightarrow} A\alpha \quad (\text{Obtained by applying one or more productions})$$

where  $\alpha$  is string of terminals and non-terminals. That is, whenever the first symbol in a partial derivation is same as the symbol from which this partial derivation is obtained, then the grammar is said to be **left-recursive** grammar. For example, consider the following grammar:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow ( E ) \mid is \end{aligned}$$

The above grammar is unambiguous but, it is having left recursion and hence, it is not suitable for top down parser. So, left recursion has to be eliminated (For details refer section 2.8.3 and 2.8.4)

- ♦ **Non-left factored grammar:** If A-production has two or more alternate productions and they have a common prefix, then the parser has some confusion in selecting the appropriate production for expanding the non-terminal A. For example, consider the following grammar that recognizes the if-statement:

$$S \rightarrow \text{if } E \text{ then } S \mid \text{else } S \mid \text{if } E \text{ then } S$$

Observe the following points:

- Both productions starts with keyword **if**.
- So, when we get the input ‘if’ from the lexical analyzer, we cannot tell whether to use the first production or to use the second production to expand the non-terminal S.
- So, we have to transform the grammar so that they do not have any common prefix. That is, left factoring is must for parsing using top-down parser.

## 2.38 □ Syntax Analyzer

---

A grammar in which two or more productions from every non-terminal A do not have a common prefix of symbols on the right hand side of the A-productions is called **left factored grammar**. (*Refer previous section for doing left-factoring*)

- ♦ **Backtracking:** The backtracking is necessary for top down parser for following reasons:
  - 1) During parsing, the productions are applied one by one. But, if two or more alternative productions are there, they are applied in order from left to right one at a time.
  - 2) When a particular production applied fails to expand the non-terminal properly, we have to apply the alternate production. Before trying alternate production, it is necessary undo the activities done using the current production. This is possibly only using backtracking.

Even though backtracking parsers are more powerful than predictive parsers, they are also much slower, requiring exponential time in general and therefore, backtracking parsers are not suitable for practical compilers.

### 2.8.7 Recursive descent parser with no-backtracking (Predictive parser)

Now, let us see “What is a predictive parser? Explain the working of predictive parser.”

**Definition:** Predictive parser is a top down parser. It is an efficient way of implementing a recursive descent parser by maintaining a stack explicitly rather than implicitly via recursive calls. The predictive parser can correctly guess or predict which production to use if two or more alternative productions are there. This is done using two ways:

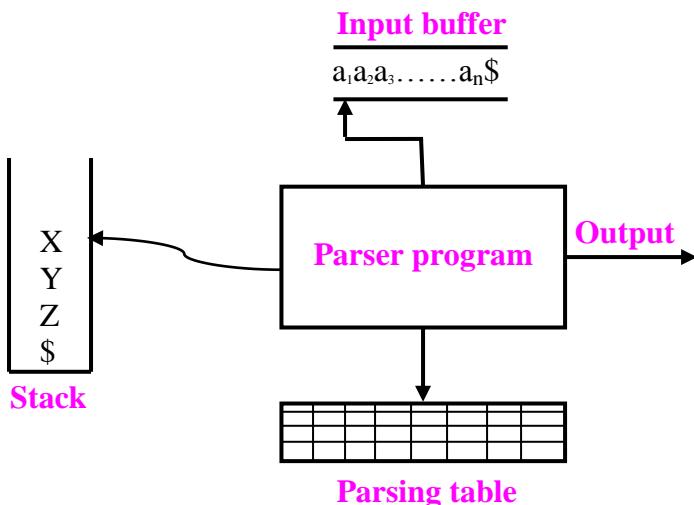
- ♦ By looking at the next few tokens (often called lookahead) it selects the correct production out of two or more alternatives productions and expand the non-terminal
- ♦ Without backtracking. So, there is no question of undoing bad choices using backtracking. In fact, bad choices will never occur.

Since, it can predict which production to use while parsing, it is called **predictive parser**. The predictive parsers accepts a restricted grammar called LL(k) grammars (defined in section 2.10)

Now, let us see “What are the various components of predictive parser? How it works?” The working of predictive parser can be explained easily by knowing the various components of the predictive parser. The block diagram showing the various parts of predictive parser are shown below:

## ❑ Systematic approach to Compiler Design - 2.39

---



The predictive parser has four components namely:

- ♦ Input
- ♦ Stack
- ♦ Parsing table
- ♦ Parsing program
- ♦ Output

- ♦ **Input :** The input buffer contains the string to be parsed and the input string ends with '\$'. Here, \$ indicates the end of the input.
- ♦ **Stack :** It contains sequence of grammar symbols and '\$' is placed initially on top of the stack. When \$ is on top of the stack, it indicates that stack is empty.
- ♦ **Parsing table :** It is a two dimensional array  $M[A, a]$  where  $A$  is a non-terminal and  $a$  is terminal or \$. The non-terminal  $A$  represent the row index and terminal  $a$  represent the column index. The entry in  $M[A, a]$  contains either a production or blank entry.
- ♦ **Parser :** It is a program which takes different actions based on  $X$  which is the symbol on top of the stack and the current input symbol  $a$ .
- ♦ **Output:** As output, the productions that are used are displayed using which the parse tree can be constructed.

**Working of the parser:** The various actions performed by the parser are shown below:

- 1) If  $X = a = \$$ , that is, if the symbol on top of the stack and the current input symbol is \$, then parsing is successful.
- 2) If  $X = a \neq \$$ , that is, if the symbol on top of the stack is same as the current input symbol but not equal to \$, then pop  $X$  from the stack and advance the input pointer to point to next symbol.

## 2.40 □ Syntax Analyzer

---

- 3) If  $X$  is a terminal and  $a \neq a$ , that is, the symbol on top of the stack is not equal to the current input symbol, then error()
- 4) If  $X$  is a non-terminal and  $a$  is the input symbol, the parser consults the parsing table  $M[X, a]$  which contains either an  $X$  production or an error entry. If  $X \rightarrow UVW$  is the corresponding production, the parser pops  $X$  from the stack and pushes  $U, V$  and  $W$  in reverse order onto the stack.

Now, before seeing how the parser parses the string, let us “[Explain parsing table and how to use the parsing table?](#) or “[What information is given in the predictive parsing table?](#)” The parsing table details and how it can be used can be explained using the example.

---

**Example 2.20:** Consider the following grammar and the corresponding predictive parsing table:

$E \rightarrow TE'$       **GRAMMAR**  
 $E' \rightarrow +TE' | \epsilon$   
 $T \rightarrow FT'$   
 $T' \rightarrow *FT' | \epsilon$   
 $F \rightarrow (E) | id$

<b><math>M \Rightarrow 2d</math> parsing table</b>						
	<b>id</b>	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
$E'$		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
$T'$		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

↑      **leftmost variable**

↑      **lookahead tokens**

The various information that we get from the above parsing table are shown below:

- ♦ The symbols present in the first column of table  $M$  i.e.,  $E, E', T, T'$  and  $F$  represent left most non-terminals in the derivation. Let us denote the non-terminal in general by  $A$

## ■ Systematic approach to Compiler Design - 2.41

---

- ♦ The symbols present in the first row such as **id**, +, \*, (, ) and \$ represent next input tokens obtained from the lexical analyzer. Let us denote the terminal in general by ‘**a**’.
- ♦ The entry in a particular row **A** and column ‘**a**’ denoted by  $M[A, a]$  may be either blank or a production. This is the production predicted for a variable **A** when the input symbol is ‘**a**’. Now, parsing is done as shown below:
  - 1) If **E** is on top of the stack and **id** is the input symbol, the parser consults the parsing table  $M[E, \text{id}]$ , gets the production  $E \rightarrow TE'$ . Now, the parser removes **E** from the stack and push  $TE'$  in reverse order. So, the entry  $M[E, \text{id}] = E \rightarrow TE'$  indicates that in the current leftmost derivation, **E** is the left most non-terminal. When the token **id** is read from the input, we expand the non-terminal **E** using the production  $E \rightarrow TE'$ .
  - 2) If  $E'$  is on top of the stack and input is ‘**)**’, the parser consults the parsing table  $M[E', \text{)}]$  and gets the production  $E' \rightarrow \epsilon$ . Now, the parser removes  $E'$  from the stack. But, nothing is there on the right side of the production to push. That is, the entry  $M[E', \text{)}] = E' \rightarrow \epsilon$  indicates that in the current leftmost derivation,  $E'$  is the leftmost non-terminal and it is replaced by  $\epsilon$ . Thus, only the leftmost variable is replaced at each step when the input symbol (lookahead token) is read from the input buffer which results in leftmost derivation. Thus, we say that predictive parsing will mimic the leftmost derivation.
  - 3) The entry in row **E** and column ‘+’ is blank. This indicates an error entry and the parser should display appropriate error messages.

Now, the various actions performed by the parser are can be implemented using algorithm. The complete algorithm to parse the string using predictive parser is shown below:

---

**Example 2.21:** The predicative parsing algorithm

---

**Input:** The string  $w$  ending with \$ (end of the input) and the parsing table

**Output:** If  $w \in L(G)$  i.e., if the input string is generated successfully from the parser, the parse tree using leftmost derivation is constructed. Otherwise, the parser displays error message.

## 2.42 □ Syntax Analyzer

**Method:** Initially the \$ and S are placed on the stack and the input buffer contains input string  $w$  ending with \$. The algorithm shown below uses the parsing table and produce the parse tree. But, instead of displaying the parse tree, we generate the productions that are used to generate the parse tree.

Let input pointer points to the first symbol of  $w$

Let  $X = S$  be the symbol on top of the stack.

```
while (X ≠ $)                                // Stack is not empty
    If ( X == a)                                // stack symbol = input symbol
        Pop X from the stack
        Advance the input pointer.
    else if X is a terminal
        Error()
    else if M[X, a] is blank
        Error()
    else if M[X, a] = X → Y1Y2Y3.....Yk
        Output the production X → Y1Y2Y3.....Yk
        Remove X from the stack
        Push Y1, Y2,Y3,.....Yk in reverse order
    endif
    Let X = top stack symbol
end while
```

The initial configuration of the parser

<b>Stack</b>	<b>Input</b>
\$S	w\$

Final configuration of parser, if parsing is successful

<b>Stack</b>	<b>Input</b>
\$	\$

## ■ Systematic approach to Compiler Design - 2.43

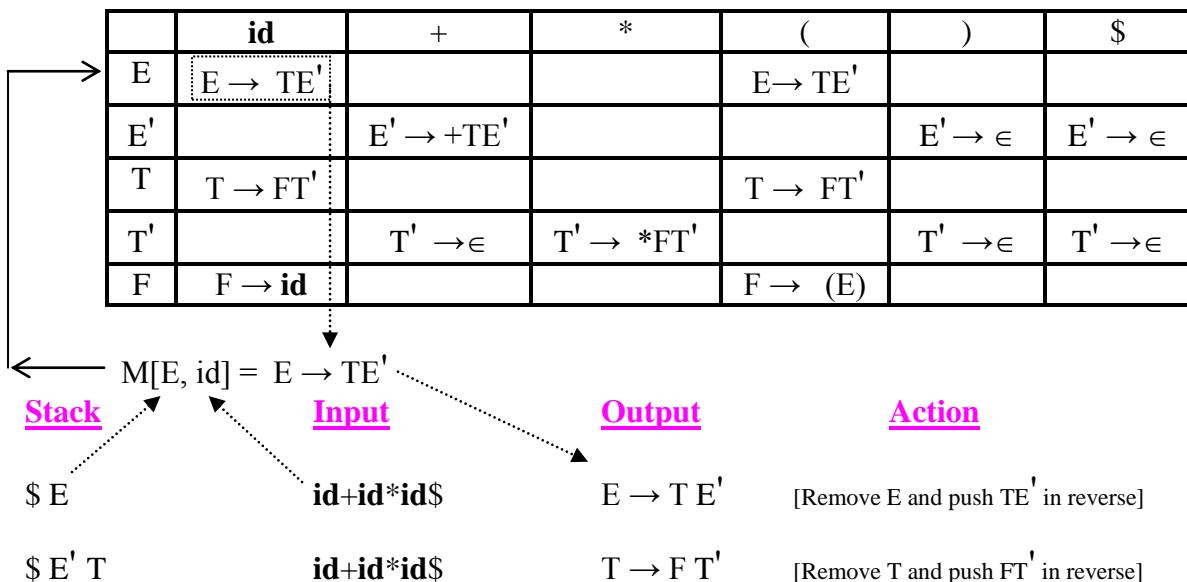
**Example 2.22:** Consider the following grammar and the corresponding predictive parsing table:

<b>GRAMMAR</b>	
$E \rightarrow TE'$	
$E' \rightarrow +TE' \mid \epsilon$	
$T \rightarrow FT'$	
$T' \rightarrow *FT' \mid \epsilon$	
$F \rightarrow (E) \mid id$	

<b>M</b>		<b>Parsing Table</b>				
	<b>id</b>	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
$E'$		$E^1 \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
$T'$		$T' \rightarrow \epsilon$	$T^1 \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Show the sequence of moves made by the predictive parser for the string **id+id\*id** during parsing.

**Solution:** The sequence of moves made by the parser for the string **id+id\*id** is shown below:



## 2.44 □ Syntax Analyzer

---

$\$ E' T' F$	$\mathbf{id+id*id\$}$	$F \rightarrow \mathbf{id}$	[Remove F and push <b>id</b> ]
$\$ E' T' \mathbf{id}$	$\mathbf{id+id*id\$}$	match(id)	[Remove <b>id</b> increment i/p ptr]
$\$ E' T'$	$+id*id\$$	$T' \rightarrow \in$	[Remove $T'$ from stack]
$\$ E'$	$+id*id\$$	$E' \rightarrow +TE'$	[Remove $E'$ and push $+TE'$ in reverse]
$\$ E' T+$	$+id*id\$$	match(+)	[Remove + increment i/p ptr]
$\$ E' T$	$id*id\$$	$T \rightarrow FT'$	[Remove T and push $FT'$ in reverse]
$\$ E' T' F$	$id*id\$$	$F \rightarrow id$	[Remove F and push <b>id</b> ]
$\$ E' T' \mathbf{id}$	$id*id\$$	match(id)	[Remove id and increment i/p ptr]
$\$ E' T'$	$*id\$$	$T' \rightarrow *FT'$	[Remove $T'$ and push $*FT'$ in reverse]
$\$ E' T' F *$	$*id\$$	match (*)	[Remove * increment i/p ptr]
$\$ E' T' F$	$id\$$	$F \rightarrow id$	[Remove F and push <b>id</b> ]
$\$ E' T' \mathbf{id}$	$id\$$	match(id)	[Remove id and increment i/p ptr]
$\$ E' T'$	$\$$	$T' \rightarrow \in$	[Remove $T'$ from stack]
$\$ E'$	$\$$	$E' \rightarrow \in$	[Remove $E'$ from stack]
$\$$	$\$$	<b>ACCEPT</b>	

Since the stack contains \$ and the input pointer points to \$, the string **id+id\*id** is parsed successfully.

## 2.9 FIRST and FOLLOW

The predictive parser can be easily constructed once we know FIRST and FOLLOW sets. These sets of symbols help us to construct the predictive parsing table very easily.

### 2.9.1 Computing FIRST symbols

Now, let us “Define FIRST( $\alpha$ )”

## □ Systematic approach to Compiler Design - 2.45

**Definition:** FIRST( $\alpha$ ) is defined as set of terminals that appear in the beginning of derivation derived from  $\alpha$ . Formally, FIRST( $\alpha$ ) is defined as shown below:

$$\text{FIRST}(\alpha) = \begin{cases} \epsilon & \text{if } \alpha = \epsilon \\ \epsilon & \text{if } \alpha \xrightarrow{*} \epsilon \\ a & \text{if } \alpha \xrightarrow{*} a\beta \end{cases} \quad \begin{array}{l} \text{Definition 1} \\ \text{Definition 2} \\ \text{Definition 3} \end{array}$$

---

**Example 2.23:** Compute FIRST sets for each non-terminal in the following grammar

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow + TE' | \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' | \epsilon \\ F &\rightarrow (E) | \text{id} \end{aligned}$$


---

**Solution:** The FIRST sets for the given grammar can be computed by obtaining various derivations as shown below:

$$\begin{array}{c} E \Rightarrow T E' \Rightarrow F T' E' \Rightarrow ((E) T'E') \\ \downarrow \qquad \downarrow \qquad \downarrow \qquad \downarrow \\ E \Rightarrow T E' \Rightarrow F T' E' \Rightarrow id T'E' \\ \downarrow \qquad \downarrow \qquad \downarrow \qquad \downarrow \\ \text{FIRST}(E) = \{ (, \text{id}) \} \\ \text{FIRST}(T) = \{ (, \text{id}) \} \\ \text{FIRST}(F) = \{ (, \text{id}) \} \end{array}$$

Consider the derivations not used in previous derivation:

$$\begin{array}{ll} E' \Rightarrow (+ T E') & T' \Rightarrow (* F T') \\ E' \Rightarrow \epsilon & T' \Rightarrow \epsilon \\ \downarrow & \downarrow \\ \text{So, FIRST}(E') = \{ \epsilon, + \} & \text{So, FIRST}(T') = \{ \epsilon, * \} \end{array}$$

Now, the final FIRST sets are written as shown below:

	E	E'	T	T'	F
FIRST	(, id	$\epsilon, +$	(, id	$\epsilon, *$	(, id

## 2.46 □ Syntax Analyzer

Now, the question is “What is the use of FIRST sets?” The FIRST sets can be used during predictive parsing while creating the predictive parsing table as shown below:

- ♦ Consider the A-production  $A \rightarrow \alpha | \beta$  and assume  $\text{FIRST}(\alpha)$  and  $\text{FIRST}(\beta)$  are disjoint i.e.,  $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \{\}$  which is an empty set.
- ♦ If the input symbol obtained from lexical analyzer is  $a$  and if  $a$  is in  $\text{FIRST}(\alpha)$  then use the production  $A \rightarrow \alpha$  during parsing.
- ♦ If the input symbol obtained from lexical analyzer is  $b$  and if  $b$  is in  $\text{FIRST}(\beta)$  then use the production  $A \rightarrow \beta$  during parsing.

Thus, using FIRST sets we can choose what production to use between the two productions  $A \rightarrow \alpha | \beta$  when input symbol is  $a$  or  $b$ .

Now, let us see “What are the rules to be followed to compute  $\text{FIRST}(X)$ ?” or “What is the algorithm to compute  $\text{FIRST}(X)$ ?” The algorithm or the rules to compute  $\text{FIRST}(X)$  are shown below:

### ALGORITHM FIRST(X)

- Rule 1:** If  $X \rightarrow a\alpha$  where  $a$  is a terminal, then  $\text{FIRST}(X) \leftarrow a$
- Rule 2:** If  $X \rightarrow \epsilon$ , then  $\text{FIRST}(X) \leftarrow \epsilon$
- Rule 3:** If  $X \rightarrow Y_1 Y_2 Y_3 \dots Y_n$  and if  $Y_1 Y_2 Y_3 \dots Y_{i-1} \not\Rightarrow \epsilon$ , then  $\text{FIRST}(X) \leftarrow \text{non-}\epsilon \text{ symbols in } \text{FIRST}(Y_i)$ .
- Rule 4:** If  $X \rightarrow Y_1 Y_2 Y_3 \dots Y_n$  and  $Y_1 Y_2 Y_3 \dots Y_n \not\Rightarrow \epsilon$ , then  $\text{FIRST}(X) \leftarrow \epsilon$
- Rule 5:** If  $X$  is a terminal or  $\epsilon$  then  $\text{FIRST}(X) \leftarrow X$

Now, let us see how FIRST sets are computed by taking some specific examples:

- 1) Rule 1 is applied if the first symbol on the right hand side of the production is a terminal. If so, then add only the first symbol.
  - ♦ Ex 1: if  $A \rightarrow aBC$ , then  $\text{FIRST}(A) = \{a\}$
  - ♦ Ex 2: if  $E \rightarrow +TE^1$  then  $\text{FIRST}(E) = \{+\}$
  - ♦ Ex 3: if  $A \rightarrow abc$ , then  $\text{FIRST}(A) = \{a\}$
- 2) Rule 2 is applied only for  $\epsilon$ -productions
  - ♦ Ex 1: if  $A \rightarrow \epsilon$ , then  $\text{FIRST}(A) = \{\epsilon\}$
  - ♦ Ex 2: if  $E^1 \rightarrow \epsilon$ , then  $\text{FIRST}(E^1) = \{\epsilon\}$
- 3) Rule 3 is applied for all productions not considered in first two steps

## ❑ Systematic approach to Compiler Design - 2.47

Ex : Consider the productions:

$$\begin{aligned} S &\rightarrow ABCd \\ A &\rightarrow \epsilon \mid +B \\ B &\rightarrow \epsilon \mid *B \\ C &\rightarrow \epsilon \mid \%B \end{aligned}$$

FIRST(A), FIRST(B), FIRST(C) are computed using rules 1 and 2 as shown below:

	S	A	B	C
FIRST		$\epsilon, +$	$\epsilon, *$	$\epsilon, \%$

To compute FIRST(S) consider the production  $S \rightarrow ABCd$  and apply rule 3 as shown below:

- a)  $S \rightarrow \boxed{ABC}d$       Add non-  $\epsilon$  symbols of FIRST(A) to FIRST(S)
- b)  $S \rightarrow \boxed{A} BCd$       Since  $A \not\Rightarrow \epsilon$ , add non-  $\epsilon$  symbols of FIRST(B) to FIRST(S)
- c)  $S \rightarrow \boxed{AB} Cd$       Since  $AB \not\Rightarrow \epsilon$ , add non- $\epsilon$  symbols of FIRST(C) to FIRST(S)
- d)  $S \rightarrow \boxed{ABC} d$       Since  $ABC \not\Rightarrow \epsilon$ , add non- $\epsilon$  symbols of FIRST(d) to FIRST(S)

So, all the above actions are pictorially represented as shown below:

	S	A	B	C
FIRST	$\%, *, +, d$	$\epsilon, +$	$\epsilon, *$	$\epsilon, \%$
	↑ ↑ ↑ ↑	step (d) step (a)	step (b)	step (c)

- 4) Rule 4 is applied for all productions whose RHS gives  $\epsilon$

Ex : Consider the productions:

$$\begin{aligned} S &\rightarrow ABC \\ A &\rightarrow \epsilon \mid +B \\ B &\rightarrow \epsilon \mid *B \\ C &\rightarrow \epsilon \mid \%B \end{aligned}$$

FIRST(A), FIRST(B), FIRST(C) are computed using rules 1 and 2 as shown below:

## 2.48 □ Syntax Analyzer

	S	A	B	C
FIRST		$\epsilon, +$	$\epsilon, *$	$\epsilon, %$

To compute FIRST(S) consider the production  $S \rightarrow ABC$  and apply rule 3 as shown below:

- a)  $S \rightarrow ABC$       Add non-  $\epsilon$  symbols of FIRST(A) to FIRST(S)
- b)  $S \rightarrow [A] BC$       Since  $A \not\Rightarrow \epsilon$ , add non-  $\epsilon$  symbols of FIRST(B) to FIRST(S)
- c)  $S \rightarrow [AB] C$       Since  $AB \not\Rightarrow \epsilon$ , add non- $\epsilon$  symbols of FIRST(C) to FIRST(S)
- d)  $S \rightarrow [ABC]$       Since  $ABC \not\Rightarrow \epsilon$ , add  $\epsilon$  to FIRST(S)

So, all the above actions are pictorially represented as shown below:

	S	A	B	C
FIRST	$\%, *, +, \epsilon$	$\epsilon, +$	$\epsilon, *$	$\epsilon, %$
	step (d)	step (a)	step (b)	step (c)

- 5) Rule 5 is applied only for terminals.
  - ♦ Ex 1:  $+$  is terminal. So,  $\text{FIRST}(+) = \{ + \}$
  - ♦ Ex 2:  $a$  is a terminal. So,  $\text{FIRST}(a) = \{a\}$
  - ♦ Ex 3: **id** is a terminal. So,  $\text{FIRST}(\text{id}) = \{\text{id}\}$

**Note:**  $\text{FIRST}(X_1 X_2 X_3 \dots X_n)$  can be computed as follows :

- 1)  $\text{FIRST}(X_1 X_2 X_3 \dots X_n) \leftarrow \text{Non- } \epsilon \text{ symbols of FIRST}(X_1)$
- 2) if  $\text{FIRST}(X_1) = \epsilon$ , then  $\text{FIRST}(X_1 X_2 X_3 \dots X_n) \leftarrow \text{FIRST}(X_2) - \epsilon$
- 3) if  $\text{FIRST}(X_1)$  and  $\text{FIRST}(X_2) = \epsilon$  then  $\text{FIRST}(X_1 X_2 X_3 \dots X_n) \leftarrow \text{FIRST}(X_3) - \epsilon$   
.....  
.....
- 4) If  $\text{FIRST}(X_1), \text{FIRST}(X_2), \dots$  and  $\text{FIRST}(X_n) = \epsilon$ , then  $\text{FIRST}(X_1 X_2 \dots X_n) \leftarrow \epsilon$

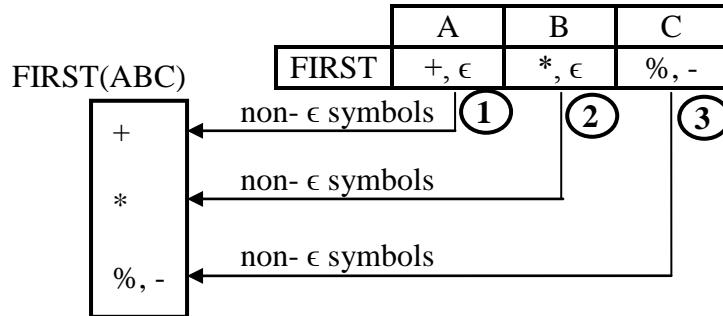
## □ Systematic approach to Compiler Design - 2.49

---

**Example 2.24:** Let  $\text{FIRST}(A) = \{+, \epsilon\}$ ,  $\text{FIRST}(B) = \{*, \epsilon\}$  and  $\text{FIRST}(C) = \{%, -, \epsilon\}$   
Compute  $\text{FIRST}(ABC)$

---

**Solution:** Using  $\text{FIRST}(A)$ ,  $\text{FIRST}(B)$  and  $\text{FIRST}(C)$ , the  $\text{FIRST}(ABC)$  can be obtained as shown below:



- ① Add non- $\epsilon$  symbols of  $\text{FIRST}(A)$
- ② Since  $\text{FIRST}(A)$  contains  $\epsilon$ , we add non- $\epsilon$  symbols of  $\text{FIRST}(B)$
- ③ Since  $\text{FIRST}(A)$  and  $\text{FIRST}(B)$  contains  $\epsilon$ , we add non- $\epsilon$  symbols of  $\text{FIRST}(C)$

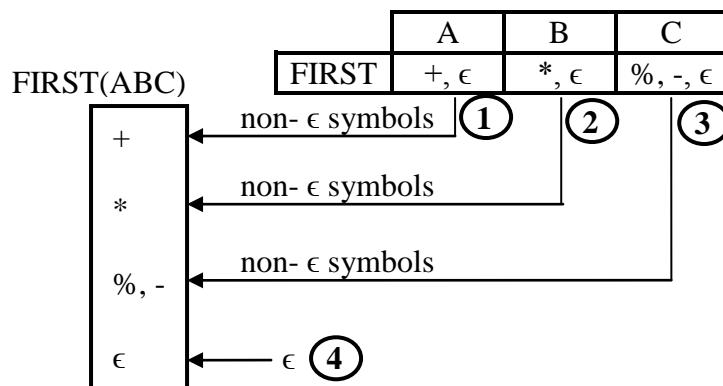
So,  $\boxed{\text{FIRST}(ABC) = \{+, *, %, -, \epsilon\}}$

---

**Example 2.25:** Let  $\text{FIRST}(A) = \{+, \epsilon\}$ ,  $\text{FIRST}(B) = \{*, \epsilon\}$  and  $\text{FIRST}(C) = \{%, -, \epsilon\}$   
Compute  $\text{FIRST}(ABC)$

---

**Solution:** Using  $\text{FIRST}(A)$ ,  $\text{FIRST}(B)$  and  $\text{FIRST}(C)$ , the  $\text{FIRST}(ABC)$  can be obtained as shown below:



## 2.50 □ Syntax Analyzer

- ① Add non-  $\epsilon$  symbols of FIRST(A)
- ② Since FIRST(A) contains  $\epsilon$ , we add non-  $\epsilon$  symbols of FIRST(B)
- ③ Since FIRST(A) and FIRST(B) contains  $\epsilon$ , we add non-  $\epsilon$  symbols of FIRST(C)
- ④ Since FIRST(A), FIRST(B) and FIRST(C) contains  $\epsilon$ , we add  $\epsilon$  symbol

So,  $\boxed{\text{FIRST(ABC)} = \{+, *, %, -, \epsilon\}}$

### 2.9.2 Computing FOLLOW symbols

Once we know, how to compute FIRST sets, let us concentrate on how to compute FOLLOW sets. Before proceeding further, let us “Define FOLLOW(A)?”

**Definition:** The FOLLOW(A) for a non-terminal A is defined as the set of terminals  $a$  that will appear immediately to the right of A in some sentential form. That is, the set of terminals  $a$  such that there exists a derivation of the form:

$$S \xrightarrow{*} \alpha A \alpha \beta$$

for some  $\alpha$  and  $\beta$ . If A is appeared as the last symbol in some sentential form, then place \$ into FOLLOW(A) where the symbol \$ is treated as “endmarker” symbol.

Now, let us see “What is the algorithm to compute FOLLOW(A)?” The algorithm to compute FOLLOW(A) is shown below:

#### ALGORITHM FOLLOW(A)

**Rule 1:** FOLLOW(S)  $\leftarrow \$$  where S is the start symbol.

**Rule 2:** If  $A \rightarrow \alpha B \beta$  is a production and  $\beta \neq \epsilon$  then  $\text{FOLLOW}(B) \leftarrow \text{non-}\epsilon \text{ symbols in FIRST}(\beta)$

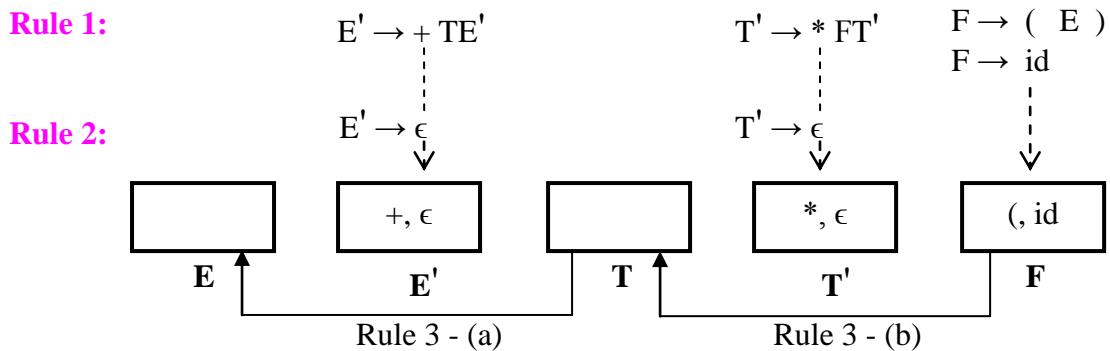
**Rule 3:** If  $A \rightarrow \alpha B \beta$  is a production and  $\beta \xrightarrow{*} \epsilon$ , then  $\text{FOLLOW}(B) \leftarrow \text{FOLLOW}(A)$

**Example 2.26:** Compute FIRST and FOLLOW sets for the following grammar:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow + TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

## ❑ Systematic approach to Compiler Design - 2.51

a) Computing FIRST sets: The FIRST sets can be computed as shown below:

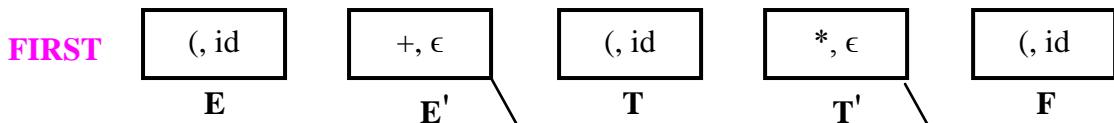


**Rule 3:** Consider the productions not considered earlier and obtain FIRST sets as shown below:

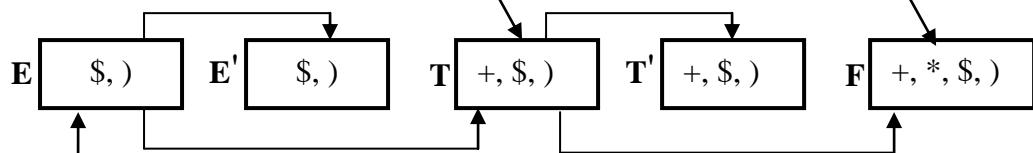
a)  $E \rightarrow T E'$  Add “FIRST(T) -  $\epsilon$ ” to FIRST(E) i.e., draw an edge from T to E in above figure.

b)  $T \rightarrow F T'$  Add “FIRST(F) -  $\epsilon$ ” to FIRST(T) i.e., draw an edge from F to T in above figure.

In the above figure, transfer FIRST(T) to FIRST(E) and from FIRST(F) to FIRST(T). So, the final FIRST sets are shown below:



b) Computing FOLLOW sets:



**Rule 1:** \$ is placed in FOLLOW(E) since E is the start symbol.

**Rule 2 &3 :** Apply rule 2 and 3 for every production of the form  $A \rightarrow \alpha B \beta$  where B is a non-terminal. In the first column shown below, copy from FIRST( $\beta$ ) to FOLLOW(B) and in the second column copy from FOLLOW(A) to FOLLOW(B).

## 2.52 □ Syntax Analyzer

<b>Rule 2</b> ( $\beta \neq \epsilon$ ) $\text{FOLLOW}(B) \leftarrow \text{FIRST}(\beta) - \epsilon$ Copy from right to left (Put arrow from right left on RHS of the production)	<b>Rule 3</b> ( $\beta \neq \epsilon$ ) $\text{FOLLOW}(A) \rightarrow \text{FOLLOW}(B)$ Copy from left to right (Put arrow from LHS of production to RHS)
$E \rightarrow T E'$ $A \rightarrow \alpha B \beta$	 $E \rightarrow T E'$ $A \rightarrow \alpha B \beta$
$E \rightarrow T E'$ Rule 2 not applicable $A \rightarrow \alpha B \beta$	 $E \rightarrow T E'$ $A \rightarrow \alpha B \beta$
$E' \rightarrow + T E'$ $A \rightarrow \alpha B \beta$	 $E' \rightarrow + T E'$ $A \rightarrow \alpha B \beta$
$E' \rightarrow [+] T E'$ Rule 2 not applicable $A \rightarrow \alpha B \beta$	 $E' \rightarrow [+] T E'$ $A \rightarrow \alpha B \beta$
$T \rightarrow F T'$ $A \rightarrow \alpha B \beta$	 $T \rightarrow F T'$ $A \rightarrow \alpha B \beta$
$T \rightarrow F T'$ Rule 2 not applicable $A \rightarrow \alpha B \beta$	 $T \rightarrow F T'$ $A \rightarrow \alpha B \beta$
$T' \rightarrow * F T'$ $A \rightarrow \alpha B \beta$	 $T' \rightarrow * F T'$ $A \rightarrow \alpha B \beta$
$T' \rightarrow [*] F T'$ Rule 2 not applicable $A \rightarrow \alpha B \beta$	 $T' \rightarrow [*] F T'$ $A \rightarrow \alpha B \beta$
$F \rightarrow ( E )$ $A \rightarrow \alpha B \beta$	$F \rightarrow ( E )$ $A \rightarrow \alpha B \beta$ Rule 3 not applicable

## □ Systematic approach to Compiler Design - 2.53

Now, let us see “What are the steps to be followed while constructing the predictive parser?” The various steps to be followed while constructing the predictive parser are shown below:

- ♦ If the grammar is ambiguous, eliminate ambiguity from the grammar
- ♦ If the grammar has left recursion, eliminate left recursion
- ♦ If the grammar has two or more alternatives having common prefix, then do left-factoring
- ♦ The resulting grammar is suitable for constructing predictive parsing table

### 2.9.3 Constructing predictive parsing table

Now, using FIRST and FOLLOW sets, we can easily construct the predictive parsing table and the productions are entered into the table  $M[A, a]$  where

- ♦  $M$  is a 2-dimensional array representing the predictive parsing table
- ♦  $A$  is a non-terminal which represent the row values
- ♦  $a$  is a terminal or  $\$$  which is endmarker and represent the column values

Now, let us “Write the algorithm to construct the predictive parsing table” The complete algorithm is shown below:

**ALGORITHM** Predictive\_Parsing\_Table( $G, M$ )

**Input** : Grammar  $G$

**Output** : Predictive parsing table  $M$

**Procedure** : For each production  $A \rightarrow \alpha$  of grammar  $G$  apply the following rules

- 1) For each terminal  $a$  in  $FIRST(\alpha)$ , add  $A \rightarrow \alpha$  to  $M[A, a]$
- 2) If  $FIRST(\alpha)$  contains  $\epsilon$ , for each symbol  $b$  in  $FOLLOW(A)$ , add  $A \rightarrow \alpha$  to  $M[A, b]$

---

**Example 2.27:** Obtain the predictive parsing table for the following grammar

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow + TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid id \end{aligned}$$

---

**Solution:** The FIRST and FOLLOW sets of each non-terminal of the given grammar are shown below: (See example 2.26 for details)

	E	E'	T	T'	F
FIRST	(, id	+ , $\epsilon$	(, id	* , $\epsilon$	(, id
FOLLOW	), \$	), \$	+ , ), \$	+ , ), \$	+ , *, ), \$

## 2.54 □ Syntax Analyzer

For every production of the form  $A \rightarrow \alpha$ , we compute  $\text{FIRST}(\alpha)$  and entries of the parsing table can be done as shown below:

Productions $A \rightarrow \alpha$	$a = \text{FIRST}(\alpha)$	$M[A, a] = A \rightarrow \alpha$	Rule
$E \rightarrow TE'$ A $\alpha$	(, id	$M[E, ()] = E \rightarrow TE'$ $M[E, id] = E \rightarrow TE'$	1
$E' \rightarrow +TE'$ A $\alpha$	+	$M[E', +] = E' \rightarrow +TE'$	1
$E' \rightarrow \epsilon$ A $\alpha$	$\epsilon$	$M[E', ()] = E' \rightarrow \epsilon$ $M[E', \$] = E' \rightarrow \epsilon$	2
$T \rightarrow FT'$ A $\alpha$	(, id	$M[T, ()] = T \rightarrow FT'$ $M[T, id] = T \rightarrow FT'$	2
$T' \rightarrow *FT'$ A $\alpha$	*	$M[T', *] = T' \rightarrow *FT'$	2
$T' \rightarrow \epsilon$ A $\alpha$	$\epsilon$	$M[T', +] = T' \rightarrow \epsilon$ $M[T', ()] = T' \rightarrow \epsilon$ $M[T', \$] = T' \rightarrow \epsilon$	3
$F \rightarrow (E)$ A $\alpha$	(	$M[F, ()] = F \rightarrow (E)$	2
$F \rightarrow \text{id}$ A $\alpha$	id	$M[F, \text{id}] = F \rightarrow \text{id}$	2

The parsing table is shown below:

	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

## ■ Systematic approach to Compiler Design - 2.55

---

**Note:** Since there are no multiple entries in the parsing table, the given grammar is called **LL(1) grammar**. If multiple entries are present in the parsing table, the grammar is not LL(1). The predictive parser accepts only the language generated from LL(1) grammar.

### 2.10 LL (1) Grammars

In this section, let us see “**What is LL (1) grammar?**”

**Definition:** The grammar from which a predictive parser, that is, recursive descent parser without backtracking is constructed is called **LL(1) grammar** where

- ♦ The first L stands for left-to-right scan of the input
- ♦ The second L stands for leftmost derivation. So, the predictive parsers always mimic the leftmost derivation.
- ♦ The digit 1 indicates number of tokens to lookahead.

In LL(1) parsing technique or predictive parsing if two or more alternative productions are there, the predictive parser also called LL(1) parser chooses the correct production by guessing using one lookahead token.

Now, let us see “**What grammars are not LL(1)?**” The following grammars are not LL(1) grammars:

- ♦ Ambiguous grammar is not LL(1)
- ♦ Left recursive grammar is not LL(1)
- ♦ The grammar which is not left factored (that is, if two or more alternative productions have common prefix), the grammar is not LL(1)
- ♦ The grammar that results in multiple entries in the parsing table is not LL(1).

Now, the question is “**How to check whether a given grammar is LL(1) or not without constructing the predictive parser?**” The grammar is said to be LL(1) if following two conditions are satisfied:

- ♦ For every production of the form  $A \rightarrow \alpha_1 | \alpha_2 | \alpha_3 | \dots | \alpha_n$ :

$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j)$  must be empty for all  $i, j \geq n$  where  $i \neq j$

- ♦ For every non-terminal A such that  $\text{FIRST}(A)$  contains  $\epsilon$ :

$\text{FIRST}(A) \cap \text{FOLLOW}(A)$  must be empty

---

**Example 2.28:** Compute FIRST and FOLLOW symbols and predictive parsing table for the following grammar:

$$\begin{aligned} S &\rightarrow iCtS \mid iCtSeS \mid a \\ C &\rightarrow b \end{aligned}$$

Is the following grammar LL(1)?

---

## 2.56 □ Syntax Analyzer

---

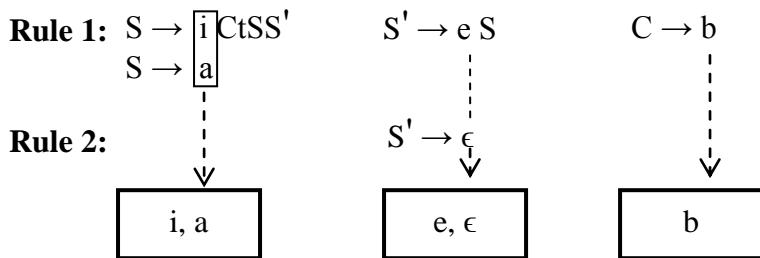
**Solution:** We know that the grammar is not left-factored since, two productions have common prefix “iCtS”. So, it is necessary to do the left-factoring for the given grammar. The left-factored grammar (for details refer section 2.8.5, example 2.19) is shown below:

$$\begin{aligned} S &\rightarrow iCtSS' \mid a \\ S' &\rightarrow \epsilon \mid eS \\ C &\rightarrow b \end{aligned}$$

The following procedure is used:

- ♦ Compute FIRST sets and FOLLOW sets
- ♦ Check whether the grammar is LL(1) or not
- ♦ Obtain the parsing table

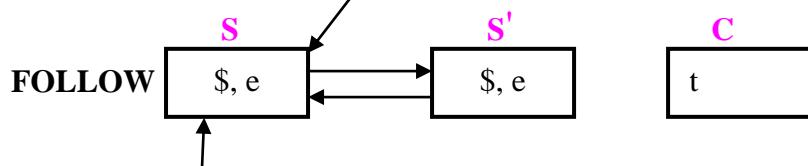
**Step 1:** The first symbols can be computed as shown below:



**Rule 3:** This rule is not applied, since all productions are already considered when we apply first two rules. So, the final FIRST sets are shown below:



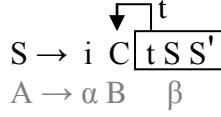
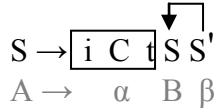
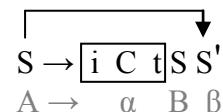
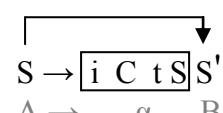
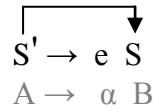
**FOLLOW sets:**



**Rule 1:**  $\$$  is placed in FOLLOW( $S$ ) since  $S$  is the start symbol.

**Rule 2 &3 :** Apply rule 2 and 3 for every production of the form  $A \rightarrow \alpha B \beta$  where  $B$  is a non-terminal. In the first column shown below, copy from  $\text{FIRST}(\beta)$  to  $\text{FOLLOW}(B)$  and in the second column copy from  $\text{FOLLOW}(A)$  to  $\text{FOLLOW}(B)$ .

## ■ Systematic approach to Compiler Design - 2.57

<b>Rule 2</b> ( $\beta \neq \epsilon$ ) $\text{FOLLOW}(B) \leftarrow \text{FIRST}(\beta) - \epsilon$	<b>Rule 3</b> ( $\beta = \epsilon$ ) $\text{FOLLOW}(A) \rightarrow \text{FOLLOW}(B)$
$S \rightarrow i \ C \ t \ S \ S'$ $A \rightarrow \alpha \ B \ \beta$ 	rule 3 not applicable
$S \rightarrow [i \ C \ t] S \ S'$ $A \rightarrow \alpha \ B \ \beta$ 	$S \rightarrow [i \ C \ t] S \ S'$ $A \rightarrow \alpha \ B \ \beta$ 
$S \rightarrow [i \ C \ t \ S] S'$ rule 2 not applicable $A \rightarrow \alpha \ B \ \beta$	$S \rightarrow [i \ C \ t \ S] S'$ $A \rightarrow \alpha \ B \ \beta$ 
$S' \rightarrow e \ S$ rule 2 not applicable $A \rightarrow \alpha \ B \ \beta$	$S' \rightarrow e \ S$ $A \rightarrow \alpha \ B \ \beta$ 

**Note:** The productions  $S \rightarrow a$  and  $C \rightarrow b$  are not considered while computing FOLLOW since there are no variables in those productions. So, the FIRST and FOLLOW sets for the left-factored grammar are shown below:

	S	S'	C
FIRST	a, i	e, $\epsilon$	b
FOLLOW	\$, e	\$, e	t

**To check whether the grammar is LL(1) or not:** Without constructing the predictive parser also we can check whether the grammar is LL(1) or not. If the grammar is LL(1), the following two conditions must be satisfied:

<b>Condition 1:</b> For a given production $A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \dots \alpha_n$	Condition to be satisfied $\text{FIRST}(\alpha_1) \cap \text{FIRST}(\alpha_2) \cap \dots \cap \text{FIRST}(\alpha_n) = \emptyset$
$S \rightarrow i \ C \ t \ S \ S' \mid a$	$\text{FIRST}(i \ C \ t \ S \ S') \cap \text{FIRST}(a)$ $\{i\} \cap \{a\} = \emptyset$
$S' \rightarrow \epsilon \mid e \ S$	$\text{FIRST}(\epsilon) \cap \text{FIRST}(e \ S)$ $\{\epsilon\} \cap \{e\} = \emptyset$

**Note:** Condition 1 is satisfied

## 2.58 □ Syntax Analyzer

<u>Condition 2:</u> If FIRST(A) contains $\epsilon$	Condition to be satisfied is $FIRST(A) \cap FOLLOW(A) = \emptyset$
FIRST(S') has $\epsilon$	$FIRST(S') \cap FOLLOW(S')$ $\{ \epsilon, \epsilon \} \cap \{ \$, \epsilon \} = \{ \epsilon \}$

**Note:** Condition 2 is not satisfied:

Since one of the condition fails, the given grammar is not LL(1). For a grammar to be LL(1), the both the conditions must be satisfied.

**Construction of predictive parsing table:** For every production of the form  $A \rightarrow \alpha$ , we compute  $FIRST(\alpha)$  and entries of the parsing table can be done as shown below:

Productions $A \rightarrow \alpha$	$a = FIRST(\alpha)$	$M[A, a] = A \rightarrow \alpha$	Rule
$S \rightarrow \underbrace{iCtSS'}_{A \quad \alpha}$	i	$M[S, i] = S \rightarrow iCtSS'$	1
$S \rightarrow a$ $A \quad \alpha$	a	$M[S, a] = S \rightarrow a$	1
$S' \rightarrow \underbrace{eS}_{A \quad \alpha}$	e	$M[S', e] = S' \rightarrow eS$	1
$S' \rightarrow \epsilon$ $A \quad \alpha$	$\epsilon$	$M[S', \epsilon] = S' \rightarrow \epsilon$ $M[S', \$] = S' \rightarrow \epsilon$	2
$C \rightarrow b$ $A \quad \alpha$	b	$M[C, b] = C \rightarrow b$	1

The parsing table is shown below:

	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iCtSS'$		
$S'$			$S' \rightarrow eS$ $S' \rightarrow \epsilon$			$S' \rightarrow \epsilon$
C		$C \rightarrow b$				

## ■ Systematic approach to Compiler Design - 2.59

---

**Example 2.29:** Given the following grammar:

$$\begin{aligned} S &\rightarrow a \mid (L) \\ L &\rightarrow L, S \mid S \end{aligned}$$

- a) Is the grammar suitable for predictive parser?
  - b) Do the necessary changes to make it suitable for LL(1) parser
  - c) Compute FIRST and FOLLOW sets for each non-terminal
  - d) Obtain the parsing table and check whether the resulting grammar is LL(1) or not.
  - e) Show the moves made by the predictive parser on the input “( a , ( a , a ) )”
- 

**Solution:** The given grammar is shown below:

$$\begin{aligned} S &\rightarrow a \mid (L) \\ L &\rightarrow L, S \mid S \end{aligned}$$

- a) Consider the production:  $L \rightarrow L, S$

Since the first symbol on RHS of the production is same as the symbol on LHS of the production, the given grammar is having left-recursion and hence, *it is not suitable for predictive parser.*

- b) To make it suitable for LL(1) parser or predictive parser, we need to eliminate left-recursion as shown below: (For details refer section 2.8.4)

### Left recursive productions

$$A \rightarrow A\alpha_1 | A\alpha_2 | A\alpha_3 | \dots | A\alpha_n | \beta_1 | \beta_2 | \beta_3 | \dots | \beta_m$$

### Right recursive productions

$$A \rightarrow \beta_1 A' | \beta_2 A' | \beta_3 A' | \dots | \beta_m A'$$

$$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \alpha_3 A' | \dots | \alpha_n A' | \epsilon$$

1)  $S \rightarrow a \mid (L)$

$S \rightarrow a \mid (L)$

2)  $L \rightarrow L, S \mid S$

$L \rightarrow SL'$

$$\begin{array}{cccc} \downarrow & \downarrow & \downarrow & \downarrow \\ A & \rightarrow & A \alpha_1 & | \beta_1 \end{array}$$

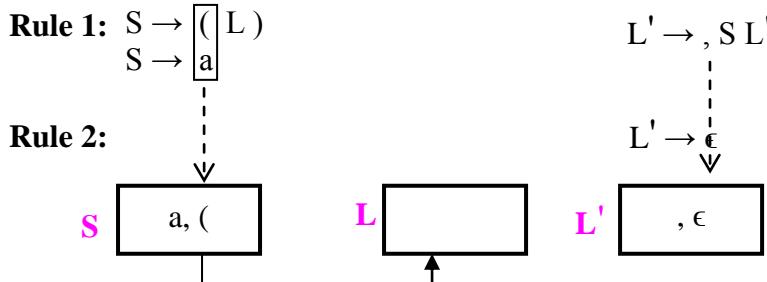
$L' \rightarrow , S L' | \epsilon$

The final grammar obtained after eliminating left recursion can be written as shown below:

$$\begin{aligned} S &\rightarrow a \mid (L) \\ L &\rightarrow SL' \\ L' &\rightarrow , S L' | \epsilon \end{aligned}$$

## 2.60 □ Syntax Analyzer

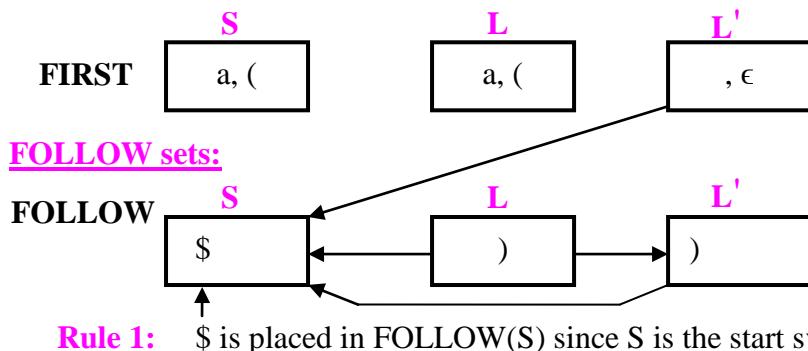
c) Computing FIRST and FOLLOW: The first set can be computed as shown below:



**Rule 3:** Consider the productions not considered earlier and obtain FIRST sets as shown below:

a)  $L \rightarrow S L'$       Add "FIRST(S) -  $\epsilon$ " to FIRST(L)

In the above figure, transfer FIRST(S) to FIRST(L). So, the final FIRST sets are shown below:



**Rule 1:** \$ is placed in FOLLOW(S) since S is the start symbol.

**Rule 2 &3 :** Apply rule 2 and 3 for every production of the form  $A \rightarrow \alpha B \beta$  where B is a non-terminal. In the first column shown below, copy from FIRST( $\beta$ ) to FOLLOW(B) and in the second column copy from FOLLOW(A) to FOLLOW(B).

Rule 2 ( $\beta \neq \epsilon$ ) $\text{FOLLOW}(B) \leftarrow \text{FIRST}(\beta) - \epsilon$	Rule 3 ( $\beta \neq \epsilon$ ) $\text{FOLLOW}(A) \rightarrow \text{FOLLOW}(B)$
$S \rightarrow ( L )$ $A \rightarrow \alpha B \beta$	rule 3 not applicable
$L \rightarrow S L'$ $A \rightarrow \alpha B \beta$	$L \rightarrow S L'$ $A \rightarrow \alpha B \beta$

## ■ Systematic approach to Compiler Design - 2.61

$L \rightarrow S L'$ $A \rightarrow \alpha B \beta$	$L \rightarrow S L'$ $A \rightarrow \alpha B \beta$
$L' \rightarrow , S L'$ $A \rightarrow \alpha B \beta$	$L' \rightarrow , S L'$ $A \rightarrow \alpha B \beta$
$L' \rightarrow [ , S ] L'$ rule 2 not applicable $A \rightarrow \alpha B \beta$	$L' \rightarrow [ , S ] L'$ results in self-loop $A \rightarrow \alpha B \beta$ and hence discard

**Note:** The productions  $S \rightarrow a$  and  $L' \rightarrow \epsilon$  are not considered while computing FOLLOW since they do not have non-terminals in those productions. So, the FIRST and FOLLOW sets for the left-factored grammar are shown below:

	S	L	L'
FIRST	a (	a (	, $\epsilon$
FOLLOW	, \$)	)	)

- d) **Construction of parsing table:** For every production of the form  $A \rightarrow \alpha$ , we compute FIRST( $\alpha$ ) and entries of the parsing table can be done as shown below:

Productions $A \rightarrow \alpha$	$a = \text{FIRST}(\alpha)$	$M[A, a] = A \rightarrow \alpha$	Rule
$S \rightarrow a$	a	$M[S, a] = S \rightarrow a$	1
$A \quad \alpha$			
$S \rightarrow ( L )$	(	$M[S, ()] = S \rightarrow ( L )$	1
$A \quad \alpha$			
$L \rightarrow S L'$	a (	$M[L, a] = L \rightarrow S L'$	1
$A \quad \alpha$		$M[L, ()] = L \rightarrow S L'$	
$L' \rightarrow \epsilon$	$\epsilon$	$M[L', ()] = L' \rightarrow \epsilon$	2
$A \quad \alpha$			
$L' \rightarrow , S L'$	,	$M[L', ',] = L' \rightarrow , S L'$	1
$A \quad \alpha$			

## 2.62 □ Syntax Analyzer

---

The above entries can be entered into parsing table as shown below:

	(	)	a	,	\$
S	$S \rightarrow (L)$		$S \rightarrow a$		
L	$L \rightarrow SL'$		$L \rightarrow SL'$		
$L'$		$L' \rightarrow \epsilon$		$L' \rightarrow , SL'$	

Since there are no multiple entries in the parse table, the resulting grammar obtained after eliminating left recursion is LL(1).

- e) The moves made by the predictive parser on the input “( a , ( a , a ) )” is shown below:

<u>Stack</u>	<u>Input</u>	<u>Output</u>	<u>Action</u>
\$ S	( a , ( a , a ) ) \$	$S \rightarrow (L)$	[Remove S and push (L) in reverse]
\$ ) L (	( a , ( a , a ) ) \$	Match (	Pop ( and increment i/p pointer
\$ ) L	a , ( a , a ) ) \$	$L \rightarrow SL'$	[Remove L and push $SL'$ in reverse]
\$ ) L' S	a , ( a , a ) ) \$	$S \rightarrow a$	[Remove S and push a in reverse]
\$ ) L' a	a , ( a , a ) ) \$	Match a	Pop a and increment i/p pointer
\$ ) L'	, ( a , a ) ) \$	$L' \rightarrow , SL'$	Remove $L'$ and push $, SL'$ in reverse
\$ ) L' S ,	, ( a , a ) ) \$	Match ,	Pop ‘,’ and increment i/p pointer
\$ ) L' S	( a , a ) ) \$	$S \rightarrow (L)$	Remove S and push (L) in reverse
\$ ) L' ) L (	( a , a ) ) \$	Match (	Pop ( and increment i/p pointer
\$ ) L' ) L	a , a ) ) \$	$L \rightarrow SL'$	Remove L and push $SL'$ in reverse
\$ ) L' ) L'S	a , a ) ) \$	$S \rightarrow a$	Remove S and push a in reverse
\$ ) L' ) L' a	a , a ) ) \$	$S \rightarrow a$	Pop a and increment i/p pointer
\$ ) L' ) L'	, a ) ) \$	$L' \rightarrow , SL'$	Remove $L'$ and push $, SL'$ in reverse

## ❑ Systematic approach to Compiler Design - 2.63

---

$\$) L') L' S ,$	$, a)) \$$	Match ,	Pop ‘,’ and increment i/p pointer
$\$) L') L' S$	$a)) \$$	$S \rightarrow a$	Remove S and push a
$\$) L') L' a$	$a)) \$$	Match a	Pop a and increment i/p pointer
$\$) L') L'$	$)) \$$	$L' \rightarrow \epsilon$	Pop L'
$\$) L')$	$)) \$$	Match )	Pop ) and increment i/p pointer
$\$) L'$	$) \$$	$L' \rightarrow \epsilon$	Pop L'
$\$)$	$) \$$	Match )	Pop ) and increment i/p pointer
$\$$	$\$$	<b>Accept</b>	

**Note:** Since stack is empty and i/p pointer also points to \$ which is endmarker, parsing is successful

---

**Example 2.30:** Given the following grammar:

$$\begin{aligned} E &\rightarrow 5 + T \mid 3 - T \\ T &\rightarrow V \mid V^*V \mid V+V \\ V &\rightarrow a \mid b \end{aligned}$$

- a) Is the grammar suitable for predictive parser?
  - b) What is the use of left-factoring? Do the left factoring for the above grammar
  - c) Compute FIRST and FOLLOW sets for each non-terminal
  - d) Without constructing the parsing table, check whether the grammar is LL(1) or not.
  - e) By constructing the parsing table, check whether the grammar is LL(1) or not.
- 

**Solution:** The given grammar is shown below:

$$\begin{aligned} E &\rightarrow 5 + T \mid 3 - T \\ T &\rightarrow V \mid V^*V \mid V+V \\ V &\rightarrow a \mid b \end{aligned}$$

- a) The E-productions and V productions are suitable for parsing. But, consider the production:

$$T \rightarrow V \mid V^*V \mid V+V$$

In the T-production, one or more productions have a common prefix V and hence the given grammar is not left-factored grammar. So, *the given grammar is not suitable for predictive parser.*

## 2.64 □ Syntax Analyzer

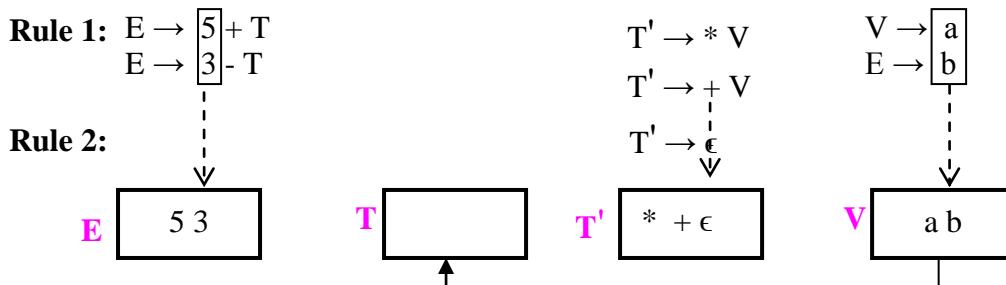
- b) To make it suitable for LL(1) parser or predictive parser, we need to do left factoring (For details refer section 2.8.5). If an A-production has two or more alternate productions and they have a common prefix, then the parser has some confusion in selecting the appropriate production for expanding the non-terminal A. So, left factoring is must for top down parser. This can be done as shown below:

Given productions	Left-factored productions
$A \rightarrow \alpha\beta_1   \alpha\beta_2   \alpha\beta_3   \dots   \alpha\beta_n   \gamma$	$A \rightarrow \alpha A'   \gamma$ $A' \rightarrow \beta_1   \beta_2   \beta_3   \dots   \beta_n$
1) $E \rightarrow 5 + T   3 - T$	$E \rightarrow 5 + T   3 - T$
2) $\begin{array}{l} T \rightarrow V \epsilon   V * V   V + V \\ A \rightarrow \alpha \beta_1   \alpha \beta_2   \alpha \beta_3 \end{array}$	$T \rightarrow V T'$ $T' \rightarrow \epsilon   * V   + V$
3) $V \rightarrow a   b$	$V \rightarrow a   b$

So, the final grammar which is obtained after doing left-factoring is shown below:

$$\begin{aligned} E &\rightarrow 5 + T | 3 - T \\ T &\rightarrow V T' \\ T' &\rightarrow \epsilon | * V | + V \\ V &\rightarrow a | b \end{aligned}$$

- c) Computing FIRST and FOLLOW: The first set can be computed as shown below:

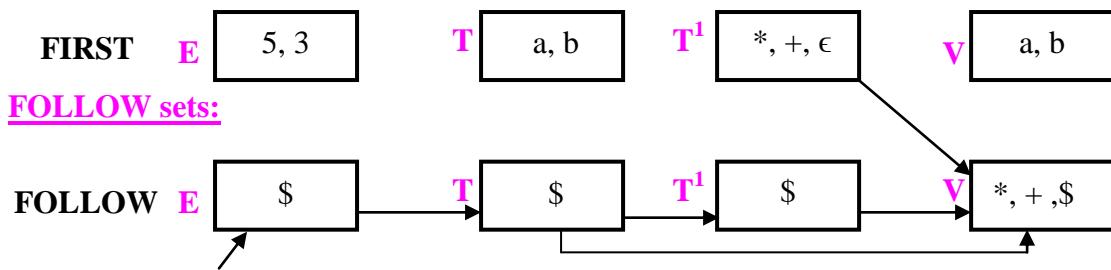


**Rule 3:** Consider the productions not considered earlier and obtain FIRST sets as shown below:

b)  $T \rightarrow V T'$       Add "FIRST(V) -  $\epsilon$ " to FIRST(T)

In the above figure, transfer FIRST(V) to FIRST(T). So, the final FIRST sets are shown below:

## Systematic approach to Compiler Design - 2.65



**Rule 1:** Place \$ into FOLLOW(S) since S is the start symbol.

**Rule 2 &3 :** Apply rule 2 and 3 for every production of the form  $A \rightarrow \alpha B \beta$  where B is a non-terminal. In the first column shown below, copy from FIRST( $\beta$ ) to FOLLOW(B) and in the second column copy from FOLLOW(A) to FOLLOW(B).

Rule 2 ( $\beta \neq \epsilon$ ) $\text{FOLLOW}(B) \leftarrow \text{FIRST}(\beta) - \epsilon$	Rule 3 ( $\beta \neq \epsilon$ ) $\text{FOLLOW}(A) \rightarrow \text{FOLLOW}(B)$
$E \rightarrow [5+] T$ rule 2 not applicable $A \rightarrow \alpha B \beta$	 $E \rightarrow [5+] T$ $A \rightarrow \alpha B \beta$
$E \rightarrow [3-] T$ rule 2 not applicable $A \rightarrow \alpha B \beta$	 $E \rightarrow [3-] T$ $A \rightarrow \alpha B \beta$
$T \rightarrow V T^1$ $A \rightarrow \alpha B \beta$	 $T \rightarrow V T^1$ $A \rightarrow \alpha B \beta$
$T \rightarrow V T^1$ rule 2 not applicable $A \rightarrow \alpha B \beta$	 $T \rightarrow V T^1$ $A \rightarrow \alpha B \beta$
$T^1 \rightarrow * V$ rule 2 not applicable $A \rightarrow \alpha B \beta$	 $T^1 \rightarrow * V$ $A \rightarrow \alpha B \beta$
$T^1 \rightarrow + V$ rule 2 not applicable $A \rightarrow \alpha B \beta$	 $T^1 \rightarrow + V$ $A \rightarrow \alpha B \beta$

## 2.66 □ Syntax Analyzer

**Note:** The productions  $T^1 \rightarrow \epsilon$  and  $V \rightarrow a | b$  are not considered while computing FOLLOW since there are no non-terminals in those productions.

So, the FIRST and FOLLOW sets for the left-factored grammar are shown below:

	E	T	$T^1$	V
FIRST	5, 3	a, b	$*, +, \epsilon$	a, b
FOLLOW	\$	\$	\$	$*, +, \$$

d) Now, for the grammar to be LL(1) the following two conditions must be satisfied:

a. The first condition has to be satisfied:

Production $A \rightarrow \alpha_1   \alpha_2   \alpha_3   \dots$	Condition to be satisfied $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$
$E \rightarrow 5 + T   3 - T$	$\text{FIRST}(5 + T) \cap \text{FIRST}(3 - T) = \emptyset$
$V \rightarrow a   b$	$\text{FIRST}(a) \cap \text{FIRST}(b) = \emptyset$

*Observe that the first condition is satisfied*

b. The second condition has to be satisfied:

If $\text{FIRST}(A) = \epsilon$	Condition to be satisfied $\text{FIRST}(A) \cap \text{FOLLOW}(A) = \emptyset$
If $\text{FIRST}(T^1) = \epsilon$	$\text{FIRST}(T^1) \cap \text{FOLLOW}(T^1)$ $\{*, +, \epsilon\} \cap \{\$\} = \emptyset$

*Observe that the second condition is satisfied*

Since, both conditions are satisfied, **the resulting grammar is LL(1)**

e) **Construction of Parsing table:** For every production of the form  $A \rightarrow \alpha$ , we compute  $\text{FIRST}(\alpha)$  and entries of the parsing table can be done as shown below:

Productions $A \rightarrow \alpha$	$a = \text{FIRST}(\alpha)$	$M[A, a] = A \rightarrow \alpha$	Rule
$E \rightarrow 5 + T$ $A \quad \alpha$	5	$M[E, 5] = E \rightarrow 5 + T$	1

## ■ Systematic approach to Compiler Design - 2.67

$E \rightarrow 3 - T$	3	$M [ E, 3 ] = E \rightarrow 3 - T$	1
$A \alpha$			
$T \rightarrow \underbrace{V T^1}_{\alpha}$	a, b	$M [ T, a ] = T \rightarrow V T^1$ $M [ T, b ] = T \rightarrow V T^1$	1
$T^1 \rightarrow \epsilon$	$\epsilon$	$M [ T^1, \$ ] = T^1 \rightarrow \epsilon$	2
$A \alpha$			
$T^1 \rightarrow *V$	*	$M [ T^1, * ] = T^1 \rightarrow *V$	1
$A \alpha$			
$T^1 \rightarrow +V$	+	$M [ T^1, + ] = T^1 \rightarrow +V$	1
$A \alpha$			
$V \rightarrow a$	a	$M [ V, a ] = V \rightarrow a$	1
$A \alpha$			
$V \rightarrow b$	b	$M [ V, b ] = V \rightarrow b$	1
$A \alpha$			

The parsing table is shown below:

	5	3	a	b	*	+	\$
E	$E \rightarrow 5 + T$	$E \rightarrow 3 - T$					
T			$T \rightarrow V T^1$	$T \rightarrow V T^1$			
$T^1$					$T^1 \rightarrow *V$	$T^1 \rightarrow +V$	$T^1 \rightarrow \epsilon$
V			$V \rightarrow a$	$V \rightarrow b$			

Since there are no multiple entries in the parse table, the resulting grammar obtained after doing left factoring is LL(1).

**Example 2.31:** Given the following grammar:

$$\begin{aligned} Z &\rightarrow d \mid XYZ \\ Y &\rightarrow \epsilon \mid c \\ X &\rightarrow Y \mid a \end{aligned}$$

- Compute FIRST and FOLLOW sets for each non-terminal
- Without constructing the parsing table, check whether the grammar is LL(1) or not.
- By constructing the parsing table, check whether the grammar is LL(1) or not.

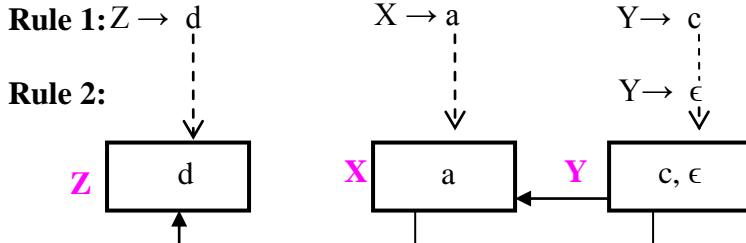
**Solution:** The given grammar is shown below:

$$\begin{aligned} Z &\rightarrow d \mid XYZ \\ Y &\rightarrow \epsilon \mid c \\ X &\rightarrow Y \mid a \end{aligned}$$

## 2.68 □ Syntax Analyzer

- a) The FIRST and FOLLOW sets are computed as shown below:

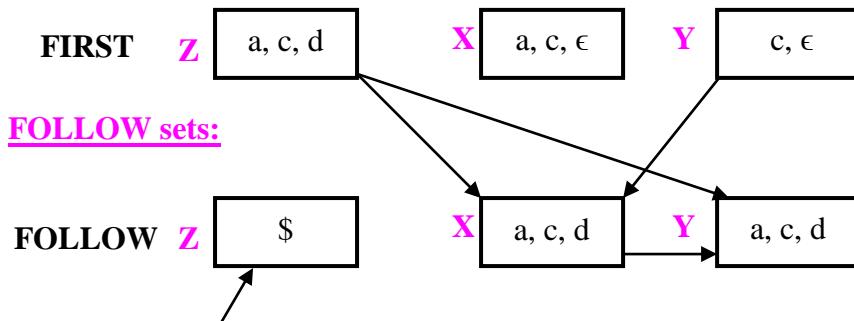
**Step 1:** The first symbols can be computed as shown below:



**Rule 3:** Consider the productions not considered earlier and obtain FIRST sets as shown below:

- 
- 1)  $Z \rightarrow XYZ$  Add “ $\text{FIRST}(X) - \epsilon$ ” to  $\text{FIRST}(Z)$
- 
- 2)  $Z \rightarrow XYZ$  Since  $\text{FIRST}(X)$  has  $\epsilon$ , add “ $\text{FIRST}(Y) - \epsilon$ ” to  $\text{FIRST}(Z)$
- 
- 3)  $Z \rightarrow XYZ$  Since  $\text{FIRST}(X)$  and  $\text{FIRST}(Y)$  has  $\epsilon$ , add “ $\text{FIRST}(Z) - \epsilon$ ” to  $\text{FIRST}(Z)$
- 
- 4)  $X \rightarrow Y$  Add “ $\text{FIRST}(Y) - \epsilon$ ” to  $\text{FIRST}(X)$
- 
- 5)  $X \rightarrow Y$  Since  $Y \not\Rightarrow \epsilon$ , add  $\epsilon$  to  $\text{FIRST}(X)$
- 

So, the final FIRST sets are shown below:



**Rule 1:** Place  $\$$  into  $\text{FOLLOW}(Z)$  since  $Z$  is the start symbol.

## □ Systematic approach to Compiler Design - 2.69

**Rule 2 &3 :** Apply rule 2 and 3 for every production of the form  $A \rightarrow \alpha B \beta$  where B is a non-terminal. In the first column shown below, copy from FIRST( $\beta$ ) to FOLLOW(B) and in the second column copy from FOLLOW(A) to FOLLOW(B).

Rule 2 ( $\beta \neq \epsilon$ ) FOLLOW(B) $\leftarrow$ FIRST( $\beta$ ) - $\epsilon$	Rule 3 ( $\beta \neq \epsilon$ ) FOLLOW(A) $\rightarrow$ FOLLOW(B)
$Z \rightarrow X \boxed{Y} Z$ $\beta = \text{FIRST}(Y) - \epsilon +$ $A \rightarrow \alpha B \beta$ $\text{FIRST}(Z) - \epsilon$	Rule 3 is not applicable
$Z \rightarrow X Y \boxed{Z}$ $\beta = \text{FIRST}(Z) - \epsilon$ $A \rightarrow \alpha B \beta$	Rule 3 is not applicable
$Z \rightarrow \boxed{X Y} Z$ $\text{rule 2 not applicable}$ $A \rightarrow \alpha B \beta$	$Z \rightarrow \boxed{X Y} Z$ $\downarrow$ $A \rightarrow \alpha B \beta$
$X \rightarrow Y$ $\text{rule 2 not applicable}$ $A \rightarrow \alpha B \beta$	$X \rightarrow Y$ $\downarrow$ $A \rightarrow \alpha B \beta$

**Note:** The productions  $Z \rightarrow d$ ,  $Y \rightarrow \epsilon | c$  and  $X \rightarrow a$  are not considered while computing FOLLOW since there are no non-terminals in those productions. So, the FIRST and FOLLOW sets for the left-factored grammar are shown below:

	Z	X	Y
FIRST	a,c,d	a,c, $\epsilon$	c, $\epsilon$
FOLLOW	\$	a,c,d	a,c,d

b) Now, for the grammar to be LL(1) the following two conditions must be satisfied:

a. The first condition to be satisfied:

Production	Condition to be satisfied
$A \rightarrow \alpha_1   \alpha_2   \alpha_3   \dots$	$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$
$Z \rightarrow d   XYZ$	$\text{FIRST}(d) \cap \text{FIRST}(XYZ) = \emptyset$ $\{d\} \cap \{a,c,d\} = d$

Condition 2 is not satisfied. Hence, the grammar is not LL(1).

## 2.70 □ Syntax Analyzer

---

- c) **Construction of Parsing table:** It can be constructed as shown below:

Productions $A \rightarrow \alpha$	$a = \text{FIRST}(\alpha)$	$M[A, a] = A \rightarrow \alpha$	Rule
$Z \rightarrow d$ $A \quad \alpha$	d	$M[Z, d] = Z \rightarrow d$	1
$Z \rightarrow XYZ$ $A \quad \alpha$	a, c, d	$M[Z, a] = Z \rightarrow XYZ$ $M[Z, c] = Z \rightarrow XYZ$ $M[Z, d] = Z \rightarrow XYZ$	1
$Y \rightarrow c$ $A \quad \alpha$	c	$M[Y, c] = Y \rightarrow c$	1
$Y \rightarrow \epsilon$ $A \quad \alpha$	$\epsilon$	$M[Y, a] = Y \rightarrow \epsilon$ $M[Y, c] = Y \rightarrow \epsilon$ $M[Y, d] = Y \rightarrow \epsilon$	2
		↑ FOLLOW(X)	
$X \rightarrow a$ $A \quad \alpha$	a	$M[X, a] = X \rightarrow a$	1
$X \rightarrow Y$ $A \quad \alpha$	c, $\epsilon$	$M[X, c] = X \rightarrow Y$ $M[X, a] = X \rightarrow Y$ $M[X, c] = X \rightarrow Y$ $M[X, d] = X \rightarrow Y$	1
		↑ FOLLOW(X)	2

The parsing table is shown below:

	a	c	d	\$
Z	$Z \rightarrow XYZ$	$Z \rightarrow XYZ$	$Z \rightarrow d$ $Z \rightarrow XYZ$	
X	$X \rightarrow a$ $X \rightarrow Y$	$X \rightarrow Y$	$X \rightarrow Y$	
Y	$Y \rightarrow \epsilon$	$Y \rightarrow c$ $Y \rightarrow \epsilon$	$Y \rightarrow \epsilon$	

Since there are multiple entries in the parse table, the given grammar is not LL(1).

---

**Example 2.32 :** Left factor the following grammar and obtain LL(1) parsing table

$$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow \text{float} \mid \text{float} * T \mid (E) \end{aligned}$$


---

## ■ Systematic approach to Compiler Design - 2.71

**Solution:** Since the right hand side of E-production and T-production has common prefixes, this grammar is not suitable for parsing. So, we have to do left factoring and see that two or more productions do not have common prefix. Left-factoring can be done as shown below:

The left factoring can be done to the given grammar as shown below:

Given productions	Left-factored productions
$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \dots \mid \alpha\beta_n \mid \gamma$	$A \rightarrow \alpha A^1 \mid \gamma$ $A^1 \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \dots \mid \beta_n$
1) $E \rightarrow T [+ E] \mid T$ $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$	$E \rightarrow T E^1$ $E^1 \rightarrow + E \mid \epsilon$
2) $T \rightarrow \text{float} \mid \text{float} * [T] (E)$ $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \gamma$	$T \rightarrow \text{float } T^1 \mid ( E )$ $T^1 \rightarrow \epsilon \mid * T$

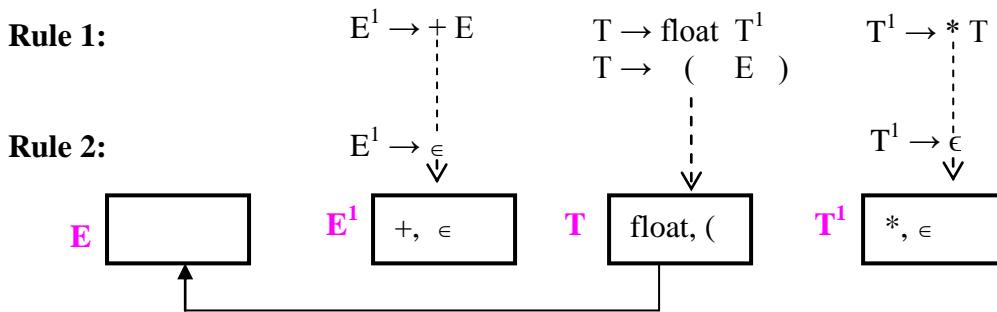
So, the grammar obtained after doing left factoring is shown below:

$$\begin{aligned} E &\rightarrow T E^1 \\ E^1 &\rightarrow + E \mid \epsilon \\ T &\rightarrow \text{float } T^1 \mid ( E ) \\ T^1 &\rightarrow \epsilon \mid * T \end{aligned}$$

- a) The FIRST and FOLLOW sets can be computed as shown below:

**FIRST sets:** are computed as shown below:

**Step 1:** The first symbols can be computed as shown below:

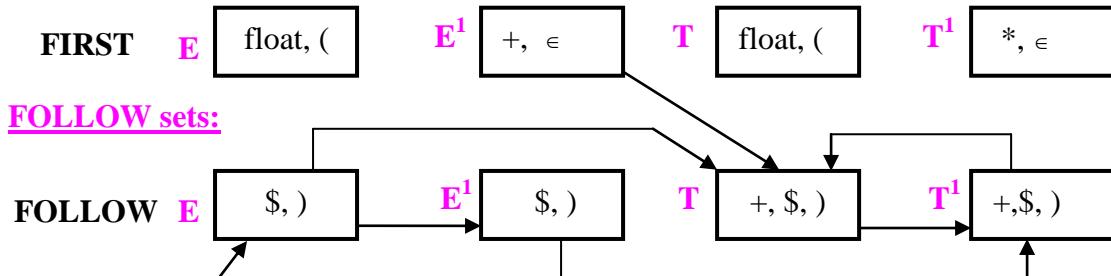


**Rule 3:** Consider the productions not considered earlier and obtain FIRST sets as shown below:

$$E \rightarrow T E^1 \quad \text{Add "FIRST}(T) - \epsilon\text{" to FIRST}(E)$$

## 2.72 □ Syntax Analyzer

In the above figure, transfer FIRST(T) to FIRST(E). So, the final FIRST sets are shown below:



**Rule 1:** Place \$ into FOLLOW(S) since S is the start symbol.

**Rule 2 &3 :** Apply rule 2 and 3 for every production of the form  $A \rightarrow \alpha B \beta$  where B is a non-terminal. In the first column shown below, copy from  $\text{FIRST}(\beta)$  to  $\text{FOLLOW}(B)$  and in the second column copy from  $\text{FOLLOW}(A)$  to  $\text{FOLLOW}(B)$ .

Rule 2 ( $\beta \neq \epsilon$ ) $\text{FOLLOW}(B) \leftarrow \text{FIRST}(\beta) - \epsilon$	Rule 3 ( $\beta = \epsilon$ ) $\text{FOLLOW}(A) \rightarrow \text{FOLLOW}(B)$
$\begin{array}{l} \downarrow \\ E \rightarrow T E^1 \\ A \rightarrow \alpha B \beta \end{array}$	$\begin{array}{l} \downarrow \\ E \rightarrow T E^1 \\ A \rightarrow \alpha B \beta \end{array}$
$\begin{array}{ll} E \rightarrow T E^1 & \text{rule 2 not applicable} \\ A \rightarrow \alpha B \beta & \end{array}$	$\begin{array}{l} \downarrow \\ E \rightarrow T E^1 \\ A \rightarrow \alpha B \beta \end{array}$
$\begin{array}{ll} E \rightarrow + E^1 & \text{rule 2 not applicable} \\ A \rightarrow \alpha B \beta & \end{array}$	$\begin{array}{l} \downarrow \\ E \rightarrow + E^1 \\ A \rightarrow \alpha B \beta \end{array}$
$\begin{array}{ll} T \rightarrow \text{float } T^1 & \text{rule 2 not applicable} \\ A \rightarrow \alpha B \beta & \end{array}$	$\begin{array}{l} \downarrow \\ T \rightarrow \text{float } T^1 \\ A \rightarrow \alpha B \beta \end{array}$
$\begin{array}{l} \downarrow \\ T \rightarrow ( E ) \\ A \rightarrow \alpha B \beta \end{array}$	$\begin{array}{ll} T \rightarrow ( E ) & \text{Rule 3 not applicable} \\ A \rightarrow \alpha B \beta & \end{array}$
$\begin{array}{ll} T^1 \rightarrow * T & \text{rule 2 not applicable} \\ A \rightarrow \alpha B \beta & \end{array}$	$\begin{array}{l} \downarrow \\ T^1 \rightarrow * T \\ A \rightarrow \alpha B \beta \end{array}$

## □ Systematic approach to Compiler Design - 2.73

**Note:** The productions  $T^1 \rightarrow \epsilon$  and  $E^1 \rightarrow \epsilon$  are not considered while computing FOLLOW since there are no non-terminals in those productions. So, the FIRST and FOLLOW sets for the left-factored grammar are shown below:

	E	$E^1$	T	$T^1$
FIRST	float, (	+ , $\epsilon$	float, (	* , $\epsilon$
FOLLOW	\$, )	\$, )	+,\$, )	+,\$, )

- b) **Construction of Parsing table:** For every production of the form  $A \rightarrow \alpha$ , we compute  $\text{FIRST}(\alpha)$  and entries of the parsing table can be done as shown below:

Productions $A \rightarrow \alpha$	$a = \text{FIRST}(\alpha)$	$M[A, a] = A \rightarrow \alpha$	Rule
$E \rightarrow T E^1$ A $\alpha$	float, (	$M[E, \text{float}] = E \rightarrow T E^1$ $M[E, '('] = E \rightarrow T E^1$	1
$E^1 \rightarrow + E$ A $\alpha$	+	$M[E^1, '+'] = E^1 \rightarrow + E$	1
$E^1 \rightarrow \epsilon$ A $\alpha$	$\epsilon$	$M[E^1, '$'] = E^1 \rightarrow \epsilon$ $M[E^1, ')'] = E^1 \rightarrow \epsilon$	2
$T \rightarrow \text{float } T^1$ A $\alpha$	float	$M[T, \text{float}] = T \rightarrow \text{float } T^1$	1
$T \rightarrow ( E )$ A $\alpha$	(	$M[T, '('] = T \rightarrow ( E )$	1
$T^1 \rightarrow \epsilon$ A $\alpha$	$\epsilon$	$M[T^1, '$'] = T^1 \rightarrow \epsilon$ $M[T^1, ')'] = T^1 \rightarrow \epsilon$ $M[T^1, '+'] = T^1 \rightarrow \epsilon$	2
$T^1 \rightarrow * T$ A $\alpha$	*	$M[T^1, '*'] = T^1 \rightarrow * T$	1

The parsing table is shown below:

	float	*	+	(	)	\$
E	$E \rightarrow T E^1$			$E \rightarrow T E^1$		
$E^1$			$E^1 \rightarrow + E$		$E^1 \rightarrow \epsilon$	$E^1 \rightarrow \epsilon$
T	$T \rightarrow \text{float } T^1$			$T \rightarrow ( E )$		
$T^1$		$T^1 \rightarrow * T$	$T^1 \rightarrow \epsilon$		$T^1 \rightarrow \epsilon$	$T^1 \rightarrow \epsilon$

### 2.11 Error recovery in predictive parsing

Now, let us see “How error recovery is done in predictive parsing?” An error is detected during predictive parsing when the following two situations occur:

## 2.74 □ Syntax Analyzer

---

- ♦ The terminal on top of the stack does not match with the next input symbol
- ♦ When non-terminal A is on top of the stack,  $a$  is the next input symbol and  $M[A, a]$  has blank entry (blank denote an error)

The error recovery is done using *panic mode* and *phrase-level recovery* as shown below:

- ♦ **Panic mode:** In this approach, error recovery is done by skipping symbols from the input until a token matches with synchronizing tokens. The synchronizing tokens are selected such that the parser should quickly recover from the errors that are likely to occur in practice. Some of the recovery techniques are shown below:
  - 1) For a non-terminal A, consider the symbols in  $\text{FOLLOW}(A)$ . These symbols can be considered as synchronizing tokens and are added into parsing table replacing only blank entries. Now, whenever there is a mismatch, keep skipping the tokens till we get one of the synchronizing character and remove A from the stack. It is likely that parsing can continue.
  - 2) For a non-terminal A, consider the symbols in  $\text{FIRST}(A)$ . These symbols can also be considered as synchronizing characters and add to the parsing table replacing only blank entries. Now, whenever there is a mismatch, keep skipping the tokens till we get one of the synchronizing character and remove A from the stack. It is also likely that parsing can continue.
  - 3) If a terminal on top of the stack cannot be matched, pop the terminal from the stack and issue “Error message” and insert the corresponding terminal and continue parsing.

For example, consider the parsing table containing synchronizing tokens and sequence of moves made by the parser in example 2.33 given later in this section.

**Phrase level recovery:** This recovery method is implemented by filling the blank entries in the predictive parsing table with pointers to error routines. These routines may change, insert, replace or delete symbols from the input and issue appropriate error messages. They may also pop from the stack.

---

**Example 2.33:** Consider the following grammar

$$\begin{aligned} E &\rightarrow TE^1 \\ E^1 &\rightarrow + TE^1 \mid \epsilon \\ T &\rightarrow FT^1 \\ T^1 &\rightarrow *FT^1 \mid \epsilon \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

and the parsing table (Refer example 2.27 for details)

## ■ Systematic approach to Compiler Design - 2.75

	<b>id</b>	+	*	(	)	\$
E	$E \rightarrow TE^1$			$E \rightarrow TE^1$		
$E^1$		$E^1 \rightarrow +TE^1$			$E^1 \rightarrow \epsilon$	$E^1 \rightarrow \epsilon$
T	$T \rightarrow FT^1$			$T \rightarrow FT^1$		
$T^1$		$T^1 \rightarrow \epsilon$	$T^1 \rightarrow *FT^1$		$T^1 \rightarrow \epsilon$	$T^1 \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Add the synchronizing tokens for the above parsing table and show the sequence of moves made by parser for the string “ ) id \* + id”

**Solution:** The synchronizing characters are the characters present in FIRST or FOLLOW sets of each non-terminal. In our example, let us add synchronizing characters by considering FOLLOW of each non-terminal replacing each blank entry in the parsing table. The FOLLOW sets of each non-terminal are shown below (Refer example 2.26 for details):

	E	$E^1$	T	$T^1$	F
FOLLOW	\$, )	\$, )	+, \$, )	+, \$, )	+, *, \$, )

Now, FOLLOW(E) = { \$, ) }. So, M[E, \$] = M[E, )] = synch only for blank entries. Similarly, FOLLOW(F) = {+, \*, \$, ) }. So, M[F,+] = M[F,\*] = M[F, \$] = M[F,) = synch. On similar lines we add synchronizing characters to the parsing table as shown below:

	<b>id</b>	+	*	(	)	\$
E	$E \rightarrow TE^1$			$E \rightarrow TE^1$	synch	synch
$E^1$		$E^1 \rightarrow +TE^1$			$E^1 \rightarrow \epsilon$	$E^1 \rightarrow \epsilon$
T	$T \rightarrow FT^1$	synch		$T \rightarrow FT^1$	synch	synch
$T^1$		$T^1 \rightarrow \epsilon$	$T^1 \rightarrow *FT^1$		$T^1 \rightarrow \epsilon$	$T^1 \rightarrow \epsilon$
F	$F \rightarrow id$	synch	synch	$F \rightarrow (E)$	synch	synch

Now, the sequence of moves made by the parser for the string “ ) id \* + id” is shown below:

<u>Stack</u>	<u>Input</u>	<u>Output</u>	<u>Action</u>
\$ E	) id * + id\$	<b>error, skip</b>	Remove ) from the input
\$ E	id * + id\$	$E \rightarrow TE^1$	Remove E and push $TE^1$ in reverse
\$ $E^1 T$	id * + id\$	$T \rightarrow FT^1$	Remove T and push $FT^1$ in reverse

## 2.76 □ Syntax Analyzer

---

$\$ E^1 T^1 F$	$id * + id \$$	$F \rightarrow id$	Remove F and push id in reverse
$\$ E^1 T^1 id$	$id * + id \$$	Match id	Pop id and increment i/p pointer
$\$ E^1 T^1$	$* + id \$$	$T^1 \rightarrow *FT^1$	Remove $T^1$ and push $*FT^1$ in reverse
$\$ E^1 T^1 F *$	$* + id \$$	Match *	Pop * and increment i/p pointer
$\$ E^1 T^1 F$	$+ id \$$	<b>error, skip</b>	Pop + from the input
$\$ E^1 T^1 F$	$id \$$	$F \rightarrow id$	Remove F and push id in reverse
$\$ E^1 T^1 id$	$id \$$	Match id	Pop id and increment i/p pointer
$\$ E^1 T^1$	$\$$	Match id	Pop id and increment i/p pointer
$\$ E^1 T^1$	$\$$	$T^1 \rightarrow \in$	Remove $T^1$ from the stack
$\$ E^1$	$\$$	$E^1 \rightarrow \in$	Remove $E^1$ from the stack
$\$$	$\$$	<b>ACCEPT</b>	

**Note:** Observe that parsing is successful and the parser has also recognized two errors. By looking at these errors if the programmer corrects the program, parsing action is successful without any errors.

**Computing FIRST sets:** The first sets of LHS of the production is nothing but the terminals obtained from the first symbols on the RHS of the production.

So,  $\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = (, id$

**Computing FOLLOW sets:** The FOLLOW sets of any non-terminal A on RHS of the production are obtained the following rules:

- 1) Sets of terminals immediately following A or sets of first symbols obtained from the non-terminals immediately following A
- 2) If A is on LHS of the production and B is right most symbol on RHS of the production then  $\text{FOLLOW}(B) = \text{FOLLOW}(A)$

### Exercises

- 1) What is a context free grammar? What is derivation? What are the two types of derivations?
- 2) Define the terms: leftmost derivation, rightmost derivation, sentence

## □ Systematic approach to Compiler Design - 2.77

---

- 3) What are the different sentential forms? What is left sentential form? What is right sentential form?
- 4) Define the terms: Language, derivation tree, yield of a tree, ambiguous grammar
- 5) Show that the following grammar is ambiguous

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E - E \\ E &\rightarrow E * E \\ E &\rightarrow E / E \\ E &\rightarrow (E) | I \\ I &\rightarrow \text{id} \end{aligned}$$

- 6) Is the following grammar ambiguous? (if-statement or if-then-else)

$$\begin{aligned} S &\rightarrow iCtS | iCtSeS | a \\ C &\rightarrow b \end{aligned}$$

- 7) What is dangling else problem? How dangling else problem can be solved
- 8) Eliminate ambiguity from the following ambiguous grammar:

$$\begin{aligned} S &\rightarrow iCtS | iCtSeS | a \\ C &\rightarrow b \end{aligned}$$

- 9) Convert the following ambiguous grammar into unambiguous grammar using normal precedence and associativity of the operators

$$\begin{aligned} E &\rightarrow E * E | E - E \\ E &\rightarrow E ^ E | E / E \\ E &\rightarrow E + E \\ E &\rightarrow (E) | \text{id} \end{aligned}$$

- 10) Convert the following ambiguous grammar into unambiguous grammar

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E - E \\ E &\rightarrow E ^ E \\ E &\rightarrow E * E \\ E &\rightarrow E / E \\ E &\rightarrow (E) | \text{id} \end{aligned}$$

by considering \* and – operators lowest priority and they are left associative, / and + operators have the highest priority and are right associative and ^ operator has precedence in between and it is left associative.

- 11) What is parsing? What are the different types of parsers?

## 2.78 □ Syntax Analyzer

---

- 12) What are error recovery strategies of the parser (or syntax analyzer)?"
- 13) What is top down parser? Show the top-down parsing process for the string id + id \* id for the grammar
- i.  $E \rightarrow E + E$
  - ii.  $E \rightarrow E * E$
  - iii.  $E \rightarrow (E)$
  - iv.  $E \rightarrow id$
- 14) What is recursive descent parser? Write the algorithm for recursive descent parser
- 15) Write the recursive descent parser for the following grammar
- $$\begin{aligned} E &\rightarrow T \\ T &\rightarrow F \\ F &\rightarrow (E) \mid id \end{aligned}$$
- 16) What are the different types of recursive descent parsers? What is the need for backtracking in recursive descent parser
- 17) Show the steps involved in recursive descent parser with backtracking for the input string *cad* for the following grammar
- $$\begin{aligned} S &\rightarrow cAd \\ A &\rightarrow ab \mid a \end{aligned}$$
- 18) For what type of grammars recursive descent parser cannot be constructed? What is the solution?
- 19) What is left recursion? What problems are encountered if a recursive descent parser is constructed for a grammar having left recursion?
- 20) Write the procedure to eliminate left recursion
- 21) Eliminate left recursion from the following grammar
- $$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$
- 22) Write the recursive descent parser for the following grammar:
- $$\begin{aligned} E &\rightarrow TE^1 \\ E^1 &\rightarrow +TE^1 \mid \epsilon \\ T &\rightarrow FT^1 \\ T^1 &\rightarrow *FT^1 \mid \epsilon \\ F &\rightarrow (E) \mid id \end{aligned}$$
- 23) Obtain top-down parse for the string id+id\*id for the following grammar
- $$E \rightarrow TE^1$$

## ■ Systematic approach to Compiler Design - 2.79

---

$$\begin{aligned} E^1 &\rightarrow + TE^1 | \epsilon \\ T &\rightarrow FT^1 \\ T^1 &\rightarrow *FT^1 | \epsilon \\ F &\rightarrow (E) | id \end{aligned}$$

24) Eliminate left recursion from the following grammar:

$$\begin{aligned} S &\rightarrow Aa | b \\ A &\rightarrow Ac | Sd | \epsilon \end{aligned}$$

25) Write the algorithm to eliminate left recursion

26) What is left factoring? What is the need for left factoring? How to do left factoring?  
Write the algorithm for doing left-factoring

27) Do the left-factoring for the following grammar:

$$\begin{aligned} S &\rightarrow iCtS | iCtSeS | a \\ C &\rightarrow b \end{aligned}$$

28) Briefly explain the problems associated with top-down parser?

29) What is a predictive parser? Explain the working of predictive parser.

30) What are the various components of predictive parser? How it works?

31) Define FIRST and FOLLOW sets and write the rules to compute FIRST and FOLLOW sets

32) Consider the following grammar:

$$\begin{aligned} E &\rightarrow TE^1 \\ E^1 &\rightarrow + TE^1 | \epsilon \\ T &\rightarrow FT^1 \\ T^1 &\rightarrow *FT^1 | \epsilon \\ F &\rightarrow (E) | id \end{aligned}$$

- Compute FIRST and FOLLOW sets for the following grammar:
- Obtain the predictive parsing table
- Show the sequence of moves made by the parser for the string **id+id\*id**
- Add the synchronizing tokens for the above parsing table and show the sequence of moves made by parser for the string “ ) id \* + id”

33) What is LL (1) grammar? How to check whether a given grammar is LL(1) or not without constructing the predictive parser

## 2.80 □ Syntax Analyzer

---

34) Compute FIRST and FOLLOW symbols and predictive parsing table for the following grammar and check whether the grammar is LL(1) or not.

$$\begin{aligned} S &\rightarrow iCtS \mid iCtSeS \mid a \\ C &\rightarrow b \end{aligned}$$

35) Given the following grammar:

$$\begin{aligned} S &\rightarrow a \mid (L) \\ L &\rightarrow L, S \mid S \end{aligned}$$

- Is the grammar suitable for predictive parser?
- Do the necessary changes to make it suitable for LL(1) parser
- Compute FIRST and FOLLOW sets for each non-terminal
- Obtain the parsing table and check whether the resulting grammar is LL(1) or not.
- Show the moves made by the predictive parser on the input “( a , ( a , a ) )”

36) Given the following grammar:

$$\begin{aligned} E &\rightarrow 5 + T \mid 3 - T \\ T &\rightarrow V \mid V^*V \mid V+V \\ V &\rightarrow a \mid b \end{aligned}$$

- Is the grammar suitable for predictive parser?
- What is the use of left-factoring? Do the left factoring for the above grammar
- Compute FIRST and FOLLOW sets for each non-terminal
- Without constructing the parsing table, check whether the grammar is LL(1)
- By constructing the parsing table, check whether the grammar is LL(1).

37) Given the following grammar:

$$\begin{aligned} Z &\rightarrow d \mid XYZ \\ Y &\rightarrow \epsilon \mid c \\ X &\rightarrow Y \mid a \end{aligned}$$

- Compute FIRST and FOLLOW sets for each non-terminal
- Without constructing the parsing table, check whether the grammar is LL(1).
- By constructing the parsing table, check whether the grammar is LL(1).

38) Left factor the following grammar and obtain LL(1) parsing table

$$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow \text{float} \mid \text{float} * T \mid (E) \end{aligned}$$

39) How error recovery is done in predictive parsing

## Module-4

### LEX AND YACC

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex.

#### Lex and Yacc

The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

Lex turns the user's expressions and actions (called source in this memo) into the host general-purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream (called input in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.

+.....+  
Source -> | Lex | -> yylex  
+.....+

+.....+  
Input -> | yylex | -> Output  
+.....+

## A YACC Parser

The structure of a lex file is intentionally similar to that of a yacc file; files are divided up into three sections, separated by lines that contain only two percent signs, as follows:

*Definition section*

```
%%
```

*Rules section*

```
%%
```

*C code section*

- The **definition** section is the place to define macros and to import header files written in [C](#). It is also possible to write any C code here, which will be copied verbatim into the generated source file.
- The **rules** section is the most important section; it associates patterns with C statements. Patterns are simply regular expressions. When the lexer sees some text in the input matching a given pattern, it executes the associated C code. This is the basis of how lex operates.
- The **C code** section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file and link it in at compile time.

Example:

```
/** Definition section **/  
  
%{  
/* C code to be copied verbatim */  
#include <stdio.h>  
%}  
  
/* This tells lex to read only one input file */
```

```
%%  
/** Rules section **/  
  
/* [0-9]+ matches a string of one or more digits */  
[0-9]+ {  
    /* yytext is a string containing the matched text. */  
    printf("Saw an integer: %s\n", yytext);  
}  
  
. { /* Ignore all other characters. */ }  
  
%%  
/** C Code section **/  
  
int main(void)  
{  
    /* Call the lexer, then quit. */  
    yylex();  
    return 0;  
}
```

## **REGULAR EXPRESSIONS:**

Regular expression specifies a set of strings to be matched. It contains text characters and operator characters. The letters of the alphabet and the digits are always text characters; thus the regular expression integer matches the string integer wherever it appears and the expression

a57D

looks for the string a57D.

### Operators:

The operator characters are

" \ [ ] ^ - ? . \* + | ( ) \$ / { } % <>

and if they are to be used as text characters, an escape should be used. The quotation mark operator ("") indicates that whatever is contained between a pair of quotes is to be taken as text characters.

Thus

`xyz"++"`

matches the string `xyz++` when it appears.

- Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

`"xyz++"`

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex lengthen the list.

- An operator character may also be turned into a text character by preceding it with \ as in

`xyz\+\+`

which is another, less readable, equivalent of the above expressions.

Another use of the quoting mechanism is to get a blank into an expression; blanks or tabs end a rule. Any blank character not contained within [] must be quoted.

- Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.
- Character classes. Classes of characters can be specified using the operator pair []. The construction [abc] matches a single character, which may be a, b, or c. Within

square brackets, most operator meanings are ignored. Only three characters are special: these are \ - and ^. The - character indicates ranges.

For example:

[a-z0-9<>\_] indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order.

- Using - between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. If it is desired to include the character - in a character class, it should be first or last; thus

[-+0-9]

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus, [^abc] matches all characters except a, b, or c, including all special or control characters

or [^a-zA-Z]

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

- Optional expressions.: The operator ? indicates an optional element of an expression.  
Thus                                        ab?c

matches either ac or abc.

- Repeated expressions: Repetitions of classes are indicated by the operators \* and +.

Ex:            a\*

# System Software & Compiler (18CS61)

---

is any number of consecutive a characters, including zero, while a+ is one or more instances of a.

For example [a-z]+

is all strings of lower case letters.

## Defining regular expressions in Lex :

Character	Meaning
A-Z, 0-9, a-z	Characters and numbers that form part of the pattern.
.	Matches any character except \n.
-	Used to denote range. Example: A-Z implies all characters from A to Z.
[ ]	A character class. Matches <i>any</i> character in the brackets. If the first character is ^ then it indicates a negation pattern. Example: [abC] matches either of a, b, and C.
*	Match <i>zero or more</i> occurrences of the preceding pattern.
+	Matches <i>one or more</i> occurrences of the preceding pattern.
?	Matches <i>zero or one</i> occurrences of the preceding pattern.
\$	Matches end of line as the last character of the pattern.
{ }	Indicates how many times a pattern can be present. Example: A{1,3} implies one or three occurrences of A may be present.
\	Used to escape meta characters. Also used to remove the special meaning of characters as defined in this table.
^	Negation.
	Logical OR between expressions.

"<some symbols>"	Literal meanings of characters. Meta characters hold.
/	Look ahead. Matches the preceding pattern only if followed by the succeeding expression. Example: A0/1 matches A0 only if A01 is the input.
( )	Groups a series of regular expressions.

Examples of regular expressions:

Regular expression	Meaning
joke[rs]	Matches either jokes or joker.
A{1,2}shis+	Matches AAshis, Ashis, AAshi, Ashi.
(A[b-e]) <sup>+</sup>	Matches zero or one occurrences of A followed by any character from b to e.

Tokens in Lex are declared like variable names in C. Every token has an associated expression. (Examples of tokens and expression are given in the following table.) Using the examples in our tables, we'll build a word-counting program. Our first task will be to show how tokens are declared.

### Examples of token declarations

Token	Associated expression	Meaning
number	([0-9]) <sup>+</sup>	1 or more occurrences of a digit
chars	[A-Za-z]	Any character
blank	" "	A blank space
word	(chars) <sup>+</sup>	1 or more occurrences of <i>chars</i>
variable	(chars)+(number)*(chars)*( number)*	

## 7.3. USING LEX:

If *lex.l* is the file containing the **lex** specification, the C source for the lexical analyzer is produced by running **lex** with the following command:

```
lex lex.l
```

**lex** produces a C file called *lex.yy.c*.

### Options

There are several options available with the **lex** command. If you use one or more of them, place them between the command name **lex** and the filename argument.

The **-t** option sends **lex**'s output to the standard output rather than to the file *lex.yy.c*.

The **-v** option prints out a small set of statistics describing the so-called finite automata that **lex** produces with the C program *lex.yy.c*.

### WORD COUNTING PROGRAM

In this section we can add C variable declarations. We will declare an integer variable here for our word-counting program that holds the number of words counted by the program. We'll also perform token declarations of Lex.

#### Declarations for the word-counting program

```
%{  
int wordCount = 0;  
%}  
chars [A-zA-Z\.\.\  
numbers ([0-9])  
delim [" "\n\t]  
whitespace {delim}+  
words {chars}+  
%%
```

The double percent sign implies the end of this section and the beginning of the second of the three sections in Lex programming.

### *Lex rules for matching patterns*

Let's look at the Lex rules for describing the token that we want to match. (We'll use C to define what to do when a token is matched.) Continuing with our word-counting program, here are the rules for matching tokens.

### **Lex rules for the word-counting program**

```
{words} { wordCount++; /*  
increase the word count by one*/ }  
{whitespace} { /* do  
nothing */ }  
{numbers} { /* one may  
want to add some processing here */ }  
%%
```

C

*code*

The third and final section of programming in Lex covers C function declarations (and occasionally the main function) Note that this section has to include the yywrap() function. Lex has a set of functions and variables that are available to the user. One of them is yywrap. Typically, yywrap() is defined as shown in the example below.

### **C code section for the word-counting program**

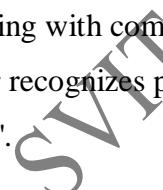
```
void main()  
{  
    yylex(); /* start the analysis*/  
    printf(" No of words:  
    %d\n", wordCount);  
}  
int yywrap()  
{  
    return 1;  
}
```

## LEXER

**lexical analysis** is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a **lexical analyzer**, **lexer** or **scanner**. A lexer often exists as a single function which is called by a parser or another function

## Token

A **token** is a string of characters, categorized according to the rules as a symbol (e.g. IDENTIFIER, NUMBER, COMMA, etc.). The process of forming tokens from an input stream of characters is called **tokenization** and the lexer categorizes them according to a symbol type. A token can look like anything that is useful for processing an input text stream or text file.

A lexical analyzer generally does nothing with combinations of tokens, a task left for a parser. For example, a typical lexical analyzer recognizes parenthesis as tokens, but does nothing to ensure that each '(' is matched with a ')'.  


Consider this expression in the C programming language:

sum=3+2;

Tokenized in the following table:

### lexeme token type

sum Identifier

= Assignment operator

3 Number

+ Addition operator

2 Number

; End of statement

Tokens are frequently defined by regular expressions, which are understood by a lexical analyzer generator such as [lex](#). The lexical analyzer (either generated automatically by a tool like lex, or hand-crafted) reads in a stream of characters, identifies the lexemes in the stream, and categorizes them into tokens. This is called "tokenizing." If the lexer finds an invalid token, it will report an error.

Following tokenizing is parsing. From there, the interpreted data may be loaded into data structures for general use, interpretation, or compiling.

### Examples:

#### 1. Write a Lex source program to copy an input file while adding 3 to every positive number divisible by 7.

```
% %

int k;

[0-9]+ {

    k = atoi(yytext);

    if (k%7 == 0)

        printf("%d", k+3);

    else

        printf("%d", k);

}
```

to do just that. The rule [0-9]+ recognizes strings of digits; atoi converts the digits to binary and stores the result in k. The operator % (remainder) is used to check whether k is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as 49.63 or X7. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```
%%

int k;

-[0-9]+      {

    k = atoi(yytext);

    printf("%d",

    k%7 == 0 ? k+3 : k);

}

-[0-9.]+      ECHO;

[A-Za-z][A-Za-z0-9]+ ECHO;
```

Numerical strings containing a “.” or preceded by a letter will be picked up by one of the last two rules, and not changed. The if-else has been replaced by a C conditional expression to save space; the form a?b:c means “if a then b else c”.

**2. Write a Lex program that histograms the lengths of words, where a word is defined as a string of letters.**

```
int lengs[100];

%%

[a-z]+ lengs[yyleng]++;

.      |

\n    ;

%%

yywrap()

{

int i;

printf("Length No. words\n");
```

```
for(i=0; i<100; i++)  
  
    if (lengs[i] > 0)  
  
        printf("%5d%10d\n",i,lengs[i]);  
  
    return(1);  
  
}
```

### **3.. Write a lex program to find the number of vowels and consonants.**

```
% {  
/* to find vowels and consonents*/  
int vowels = 0;  
int consonents = 0;  
% }  
%%  
[ \t\n]+  
[aeiouAEIOU] vowels++;  
[bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ] consonents++;  
. . .  
%%  
main()  
{  
yylex();  
printf(" The number of vowels = %d\n", vowels);  
printf(" number of consonents = %d \n", consonents);  
return(0);  
}
```

The same program can be executed by giving alternative grammar. It is as follows:  
Here a file is opened which is given as a argument and reads to text and counts the number of vowels and consonants.

```
% {  
    unsigned int vowelcount=0;  
    unsigned int consocount=0;  
}  
  
vowel [aeiouAEIOU]  
consonant [bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ]  
eol \n  
  
%%  
  
{vowel} { vowelcount++;}  
{consonant} { consocount++; }  
  
%%  
main(int argc,char *argv[]){  
if(argc > 1){  
FILE *fp;  
fp=fopen(argv[1],"r");  
if(!fp){  
    fprintf(stderr,"could not open %s\n",argv[1]);  
    exit(1);  
}  
yyin=fp;  
}  
yylex();  
printf(" vowelcount=%u consonantcount=%u\n ",vowelcount,consocount);  
return(0);  
}
```

SVIT

**4. Write a Lex program to count the number of words, characters, blanks and lines in a given text.**

```
% {  
    unsigned int charcount=0;  
    int wordcount=0;  
    int linecount=0;  
    int blankcount =0;  
}  
  
word[^ \t\n]+  
eol \n  
%%  
[ ] blankcount++;  
{word} { wordcount++; charcount+=yyleng;}  
{eol} {charcount++; linecount++;}  
. { ECHO; charcount++;}  
%%  
main(argc, argv)  
int argc;  
char **argv;  
{  
if(argc > 1)  
{  
    FILE *file;  
    file = fopen(argv[1],"r");  
    if(!file)  
    {  
        fprintf(stderr, "could not open %s\n", argv[1]);  
        exit(1);  
    }  
    yyin = file;
```

```
yylex();  
printf("\nThe number of characters = %u\n", charcount);  
printf("The number of wordcount = %u\n", wordcount);  
printf("The number of linecount = %u\n", linecount);  
printf("The number of blankcount = %u\n", blankcount);  
return(0);  
}  
else  
printf(" Enter the file name along with the program \n");  
}
```

**5. Write a lex program to find the number of positive integer, negative integer, positive floating positive number and negative floating point number.**

```
% {  
    int posnum = 0;  
    int negnum = 0;  
int posflo = 0;  
    int negflo = 0;  
}  
%%  
[\n\t];  
([0-9]+) {posnum++;}  
-?([0-9]+) {negnum++;}  
([0-9]*\.[0-9]+) { posflo++; }  
-?([0-9]*\.[0-9]+) { negflo++; }  
. ECHO;  
%%  
main()  
{  
    yylex();
```

```
printf("Number of positive numbers = %d\n", posnum);
printf("number of negative numbers = %d\n", negnum);
printf("number of floating positive number = %d\n", posflo);
printf("number of floating negative number = %d\n", negflo);
}
```

**6. Write a lex program to find the given c program has right number of brackets. Count the number of comments. Check for while loop.**

```
% {
/* find main, comments, {, (, ), } */
int comments=0;
int opbr=0;
int clbr=0;
int opfl=0;
int clfl=0;
int j=0;
int k=0;
%
%"main()" j=1;
"/**[\ \t].*[ \t]**/" comments++;
"while("[0-9a-zA-Z]*")"[ \t]*\n{ "[ \t]*.*"}" k=1;
^[\ \t]*{ "[ \t]*\n
^[\ \t]*" " k=1;
"( opbr++;
")" clbr++;
"{" opfl++;
"}" clfl++;
```

```
[^ \t\n]+
. ECHO;
%%%
main(argc, argv)
int argc;
char *argv[];
{
if (argc > 1)
{
    FILE *file;
    file = fopen(argv[1], "r");
    if (!file)
    {
        printf("error opeing a file \n");
exit(1);
    }
    yyin = file;
}
yylex();
if(opbr != clbr)
printf("open brackets is not equal to close brackets\n");
if(opfl != clfl)
    printf("open flower brackets is not equal to close flower brackets\n");
    printf(" the number of comments = %d\n",comments);
if(!j)
    printf("there is no main function \n");
if (k)
printf("there is loop\n");
else printf("there is no valid for loop\n");
return(0);
}
```

SVIT

**6. Write a lex program to replace scanf with READ and printf with WRITE statement also find the number of scanf and printf.**

```
% {  
int pc=0,sc=0;  
% }  
%%  
printf fprintf(yyout,"WRITE");pc++;  
scanf fprintf(yyout,"READ");sc++;  
. ECHO;  
%%  
main(int argc,char* argv[])  
{  
if(argc!=3)  
{  
printf("\nUsage: %s <src><dest>\n",argv[0]);  
return;  
}  
yyin=fopen(argv[1],"r");  
yyout=fopen(argv[2],"w");  
yylex();  
printf("\nno. of prints:%d\nno. of scans:%d\n",pc,sc);  
}
```

**7. Write a lex program to find whether the given expression is valid.**

```
% {  
#include <stdio.h>  
int valid=0,ctr=0,oc = 0;  
% }  
NUM [0-9]+
```

```
OP [+*/-]
%%
{NUM}({OP}{NUM})+ {
    valid = 1;
    for(ctr = 0;yytext[ctr];ctr++)
    {
        switch(yytext[ctr])
        {
            case '+':
            case '-':
            case '*':
            case '/': oc++;
        }
    }
    {NUM}\n {printf("\nOnly a number.");}
\n { if(valid) printf("valid \n operatorcount = %d",oc);
else printf("Invalid");
valid = oc = 0;ctr=0;
}
%%
main()
{
    yylex();
}

/*
     Another solution for the same problem
*/
%{
int oprc=0,digc=0,top=-1,flag=0;
char stack[20];
```

```
% }

digit [0-9] +
opr [+*/-]

%%

[ \n\t]+
['('] {stack[++top]='(';}
[')'] {flag=1;
if(stack[top]=='&&(top!=-1)
    top--;
else
{
    printf("\n Invalid expression\n");
    exit(0);
}
}

{digit} {digc++;}
{opr}/['('] { oprc++; printf("%s",yytext);}
{opr}/{digit} { oprc++; printf("%s",yytext);}
. {printf("Invalid "); exit(0);}

%%

main()
{
    yylex();
    if((digc==oprc+1||digc==oprc) && top==-1)
    {
        printf("VALID");
        printf("\n oprc=%d\n digc=%d\n",oprc,digc);
    }
    else
        printf("INVALID");
}
```

**8. Write a lex program to find the given sentence is simple or compound.**

```
%{  
int flag=0;  
%}  
%%  
( " [aA][nN][dD]" ")|(" "[oO][rR]" ")|(" "[bB][uU][tT]" ") flag=1;  
. ;  
%%  
main()  
{yylex();  
if (flag==1)  
    printf("COMPOUND SENTENCE \n");  
else  
    printf("SIMPLE SENTENCE \n");  
}
```

**9. Write a lex program to find the number of valid identifiers.**

```
%{  
int count=0;  
%}  
%%  
( " int ")|(" float ")|(" double ")|(" char ")  
  
{  
int ch; ch = input();  
for(;;)  
{  
if (ch==',') {count++;}  
else
```

```
if(ch==';') {break;}
ch = input();
}
count++;
}
%
main(int argc,char *argv[])
{
yyin=fopen(argv[1],"r");
yylex();
printf("the no of identifiers used is %d\n",count);
}
```

## RECOMMENDED QUESTIONS:

1. write the specification of lex with an example? (10)
2. what is regular expressions? With examples explain? (8)
3. write a lex program to count the no of words , lines , space, characters? (8)
4. write a lex program to count the no of vowels and consonants? (8)
5. what is lexer- parser communication? Explain? (5)
6. write a program to count no of words by the method of substitution? (7)

### USING YACC

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters or in terms of higher level constructs such as names and numbers. The user supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification. Yacc is written in portable C.

Yacc provides a general tool for imposing structure on the input to a computer program. User prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input.

#### Grammars:

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ',' year
```

Here, date, month\_name, day, and year represent structures of interest in the input process; presumably, month\_name, day, and year are defined elsewhere. The comma ``,'' is

enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

July 4, 1776

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal symbol. To avoid confusion, terminal symbols will usually be referred to as tokens.

### **Basic Specifications:**

Every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent ``%%'' marks. (The percent ``%'' is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

#### **declarations**

**%%**

#### **rules**

**%%**

#### **programs**

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

**%%**

## rules

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /\* . . . \*/, as in C and PL/I.

The rules section is made up of one or more grammar rules.

A grammar rule has the form:

A:BODY;

A represents a non terminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation. Names may be of arbitrary length, and may be made up of letters, dot ``.'', underscore ``\_'', and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

## AYACC PARSER

A literal consists of a character enclosed in single quotes ``''. As in C, the backslash ``\'' is an escape character within literals, and all the C escapes are recognized. Thus

**'\n' newline**  
**'\r' return**  
**'\' single quote ``''**  
**'\\' backslash ``\''**  
**'\t' tab**  
**'\b' backspace**  
**'\f' form feed**  
**'\xxx' ``xxx'' in octal**

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar ``|'' can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

**A** : **B C D** ;

**A** : **E F** ;

**A** : **G** ;

can be given to Yacc as

```
A : B C D  
| E F  
| G  
;
```

- It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.
- If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:
- **empty** : ;
- Names representing tokens must be declared; this is most simply done by writing
- **%token name1, name2 ...**

In the declarations section, Every name not defined in the declarations section is assumed to represent a non-terminal symbol. Every non-terminal symbol must appear on the left side of at least one rule.

- Of all the nonterminal symbols, one, called the start symbol, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section.
- It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the % start keyword:
- **%start symbol**
- The end of the input to the parser is signaled by a special token, called the endmarker. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the end-marker is seen; it accepts the input. If the endmarker is seen in any other context, it is an error.
- It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as ``end-of-file'' or ``end-of-record''.  


Actions:

- With each grammar rule, the user may associate actions to be Yacc: Yet Another Compiler-Compiler performed each time the rule is recognized in the input process.
- These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.
- An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces ``{'' and ``}'' . For example,

**A : '(' B ')'**

```
{ hello( 1, "abc" ); }
```

and

**XXX : YYY ZZZ**

```
{ printf("a message\n");
```

```
flag = 25; }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol ``dollar sign'' ``\$\$'' is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable ``\$\$'' to some value. For example, an action that does nothing but return the value 1 is

```
{ $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables \$1, \$2 . . . , which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

**A : B C D ;**

for example, then \$2 has the value returned by C, and \$3 the value returned by D.

As a more concrete example, consider the rule

```
expr : '(' expr ')' ;
```

The value returned by this rule is usually the value of the expr in parentheses. This can be indicated by

```
expr : '(' expr ')' { $$ = $2; }
```

By default, the value of a rule is the value of the first element in it (\$1). Thus, grammar rules of the form

A : B ;

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks ``%{" and ``%}" . These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

**%{ int variable = 0; %}**

could be placed in the declarations section, making variable accessible to all of the actions. The Yacc parser uses only names beginning in ``yy"; the user should avoid such names.

In these examples, all the values are integers.

## Lexer

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called yylex. The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called yylex. The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the ``# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

**yylex(){**

```
extern int yylval;  
  
int c;  
  
...  
  
c = getchar();  
  
...  
  
switch( c ) {  
  
    ...  
  
    case '0':  
  
    case '1':  
  
        ...  
  
    case '9':  
  
        yylval = c-'0';  
  
        return( DIGIT );  
  
    ...  
  
}
```

...

- The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

SVIT

- This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser;
- For example, the use of token names ‘if’ or ‘while’ will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name error is reserved for error handling, and should not be used naively.
- The token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc.
- The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

## Compiling and running a SimpleParser:

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls yylex to obtain the next token.

2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off the stack, and in the lookahead token being processed or left alone.

The shift action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

**IF shift 34**

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The reduce action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ``.'') is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

**reduce 18**

refers to grammar rule 18, while the action

**IF shift 34**

refers to state 34. Suppose the rule being reduced is

**A : x y z ;**

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule).

In effect, these states were the ones put on the stack while recognizing x, y, and z, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable `yylval` is copied onto the value stack. After the return from the user code, the reduction is carried out. When the `goto` action is done, the external variable `yyval` is copied onto the value stack. The pseudo-variables `$1`, `$2`, etc., refer to the value stack.

## **Arithmetic Expressions and Ambiguity:**

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule

**expr : expr '-' expr**

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

**expr - expr - expr**

the rule allows this input to be structured as either

**( expr - expr ) - expr**

or as

**expr - ( expr - expr )**

(The first is called **left association**, the second **right association**).

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

**expr - expr - expr**

When the parser has read the second expr, the input that it has seen:

**expr - expr**

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule; after applying the rule, the input is reduced to expr (the left side of the rule). The parser would then read the final part of the input:

- expr  
expr - expr

and again reduce. The effect of this is to take the left associative interpretation. Alternatively, when the parser has seen

**expr - expr - expr**

it could defer the immediate application of the rule, and continue reading the input until it had seen

**expr - expr**

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

**expr - expr**

The parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a shift / reduce conflict. It may also happen that the

parser has a choice of two legal reductions; this is called a reduce / reduce conflict. Note that there are never any ``Shift/shift" conflicts.

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a disambiguating rule.

Yacc invokes two **disambiguating** rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an ``if-then-else" construction:

```
stat : IF '(' cond ')' stat
      | IF '(' cond ')' stat ELSE stat
      ;
```

In these rules, IF and ELSE are tokens, cond is a nonterminal symbol describing conditional (logical) expressions, and stat is a nonterminal symbol describing statements. The first rule will be called the simple-if rule, and the second the if-else rule.

These two rules form an ambiguous construction, since input of the form

EXAMPLE:

IF ( C1 ) IF ( C2 ) S1 ELSE S2

can be structured according to these rules in two ways:

IF ( C1 ) {

IF ( C2 ) S1

}

ELSE S2

or

IF ( C1 ) {

IF ( C2 ) S1

ELSE S2

}

- The second interpretation is the one given in most programming languages having this construct. Each ELSE is associated with the last preceding ``un-ELSE'd'' IF. In this example, consider the situation where the parser has seen

IF ( C1 ) IF ( C2 ) S1

and is looking at the ELSE. It can immediately reduce by the simple-if rule to get

IF ( C1 ) stat

and then read the remaining input,

ELSE S2

and reduce

IF ( C1 ) stat ELSE S2

by the if-else rule. This leads to the first of the above groupings of the input.

- On the other hand, the ELSE may be shifted, S2 read, and then the right hand portion of

IF ( C1 ) IF ( C2 ) S1 ELSE S2

can be reduced by the if-else rule to get

IF ( C1 ) stat

which can be reduced by the simple-if rule.

- Once again the parser can do two valid things - there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.
- This shift/reduce conflict arises only when there is a particular current input symbol, ELSE, and particular inputs already seen, such as

**IF ( C1 ) IF ( C2 ) S1**

- In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

**stat : IF '(' cond ')' stat**

- Once again, notice that the numbers following ``shift'' commands refer to other states, while the numbers following ``reduce'' commands refer to grammar rule numbers. In the y.output file, the rule numbers are printed after those rules which can be reduced.

## Variables and Typed Tokens

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars.

- The basic notion is to write grammar rules of the form

**expr : expr OP expr**

and

**expr : UNARY expr**

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators.

- This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.
- The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens.
- All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

**%left '+' '-'**

**%left '\*' '/'**

## System Software & Compiler (18CS61)

---

- describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative.
- The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators

**%right '='**

**%left '+' '-'**

- **%left '\*' '/'**
- **%%**
- **expr : expr '=' expr**
  - | **expr '+' expr**
  - | **expr '-' expr**
  - | **expr '\*' expr**
  - | **expr '/' expr**
  - | **NAME**
  - ;

might be used to structure the input

**a = b = c\*d - e - f\*g**

as follows

**a = ( b = ( ((c\*d)-e) - (f\*g) ) )**

- When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences.
  - An example is unary and binary '-'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. %prec appears

immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal.

- It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left '+' '-'
%left '*' '/'
% %

expr : expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| '-' expr %prec '*'
| NAME
;
;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

The precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.

2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.



Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by Yacc. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially ``cookbook'' fashion, until some experience has been gained. The y.output file is very useful in deciding whether the parser is actually doing what was intended.

### **Recursive rules:**

The algorithm used by the Yacc parser encourages so called ``left recursive'' grammar rules: rules of the form

**name : name rest\_of\_rule ;**

These rules frequently arise when writing specifications of sequences and lists:

```
list  :  item
      |  list ',' item
      ;
;
```

and

```
seq  :  item
      |  seq item
      ;
;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq  :  item
      |  item seq
      ;
;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq  :  /* empty */
      |  seq item
      ;
;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read

## RUNNING BOTH LEXER AND PARSER:

The yacc program gets the tokens from the lex program. Hence a lex program has to be written to pass the tokens to the yacc. That means we have to follow different procedure to get the executable file.

- i. The lex program <lexfile.l> is first compiled using lex compiler to get **lex.yy.c**.
- ii. The yacc program <yaccfile.y> is compiled using yacc compiler to get **y.tab.c**.
- iii. Using c compiler both the lex and yacc intermediate files are compiled with the lex library function. **cc y.tab.c lex.yy.c -ll**.
- iv. If necessary output file name can be included during compiling with **-o** option.

## Examples

### 1. Write a Yacc program to test validity of a simple expression with +, -, ., and \*

```
/* Lex program that passes tokens */

%{

#include "y.tab.h"

extern int yyparse();

%}

%%

[0-9]+ { return NUM; }

[a-zA-Z_][a-zA-Z_0-9]* { return IDENTIFIER; }

[+-] {return ADDORSUB; }

[*/] {return PROORDIV; }

[]() {return yytext[0]; }

[\n] {return '\n'; }

%%

int main()

{
```

```
yyparse();  
}  
/* Yacc program to check for valid expression */  
  
% {  
#include<stdlib.h>  
extern int yyerror(char * s);  
extern int yylex();  
% }  
%token NUM  
%token ADDORSUB  
%token PROORDIV  
%token IDENTIFIER  
%%  
input :  
    | input line  
        ;  
line   : '\n'  
    | exp '\n' { printf("valid"); }  
    | error '\n' { yyerrok; }  
        ;  
exp    : exp ADDORSUB term  
    | term  
        ;  
term   : term PROORDIV factor  
    | factor  
        ;  
factor : NUM  
    | IDENTIFIER  
    | '(' exp ')'  
        ;  
%%
```

```
int yyerror(char *s)
{
    printf("%s","INVALID\n");
}

/* yacc program that gets token from the c porogram */

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

%token NUMBER LETTER

%left '+'
%left '*'
% %

line:line expr '\n' {printf("\nVALID\n");}
| line '\n'
|
|error '\n' { yyerror ("\n INVALID"); yyerrok;}
;

expr:expr '+' expr
|expr '-' expr
|expr '*'expr
|expr '/' expr
| NUMBER
| LETTER
;
% %

main()
{
    yyparse();
}
```

```
}

yylex()
{
char c;

while((c=getchar())==' ');

if(isdigit(c)) return NUMBER;
if(isalpha(c)) return LETTER;
return c;
}

yyerror(char *s)
{
printf("%s",s);
}
```

**2. Write a Yacc program to recognize validity of a nested ‘IF’ control statement and display levels of nesting in the nested if.**

```
/* Lex program to pass tokens */

%{

#include "y.tab.h"

%}

digit [0-9]

num {digit} + ("." {digit}+)?

binopr [+/*%0^=><&|=!=|>=|<=|>|<|~|!]

unopr [~!]

char [a-zA-Z_]

id {char}({digit} | {char})*

space [ \t]
```

```
% %

{space} ;

{num} return num;

{ binopr } return binopr;

{ unopr } return unopr;

{ id} return id

“if” return if

. return yytext[0];

%%

NUMBER {DIGIT}+
/* Yacc program to check for the valid expression */

%{

#include<stdio.h>

int cnt;

%}

%token binopr

%token unop

%token num

%token id

%token if

%%

foo: if_stat { printf("valid: count = %d\n", cnt); cnt = 0;
```

```
        exit(0);

    }

    | error { printf("Invalid \n"); }

if_stat: token_if '(' cond ')' comp_stat {cnt++;}

cond: expr

;

expr: sim_exp

| '(' expr ')'

| expr binop factor

| unop factor

;

factor: sim_exp

| '(' expr ')'

;

sim_exp: num

| id

;

sim_stat: expr ';'

| if

;

stat_list: sim_stat
```

```
| stat_list sim_stat  
;  
comp_stat:    sim_stat  
| '{' stat_list '}'  
;  
%%  
main()  
{  
    yyparse();  
}  
yyerror(char *s)  
{  
    printf("%s\n", s);  
    exit(0);  
}
```

SVIT

### **3. Write a Yacc program to recognize a valid arithmetic expression that uses +, -, /, \*.**

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

```
% token num

% left '+' '-'

% left '*' '/'

%%

st      : st expn '\n' {printf ("valid \n"); }

|
| st '\n'

| error '\n' { yyerror ("Invalid \n"); }

;

%%

void main()

{

    yyparse (); return 0 ;

}

yylex()

{

    char c;

    while (c = getch () ) == ' ')

        if (is digit (c))

            return num;
```

SVIT

```
        return c;  
  
    }  
  
    yyerror (char *s)  
  
    {  
  
        printf("%s", s);  
  
    }
```

**4. Write a yacc program to recognize an valid variable which starts with letter followed by a digit. The letter should be in lowercase only.**

```
/*      Lex program to send tokens to the yacc program      */
```

```
% {  
  
    #include "y.tab.h"  
  
}%
```

```
%%
```

```
[0-9] return digit;
```

```
[a-z] return letter;
```

```
[\n] return yytext[0];
```

```
. return 0;
```

```
%%
```

```
/*      Yacc program to validate the given variable      */
```

```
%{

#include<type.h>

%}

% token digit letter ;

%%

ident  : expn '\n' { printf ("valid\n"); exit (0); }

;

expn   : letter

| expn letter

| expn digit

| error { yyerror ("invalid \n"); exit (0); }

;

%%

main()

{

    yyparse();

}

yyerror (char *s)

{

    printf("%s", s);

}
```

}

/\* Yacc program which has c program to pass tokens \*/

```
%{  
#include <stdio.h>  
#include <ctype.h>  
%}  
%token LETTER DIGIT  
%%  
st:st LETTER DIGIT '\n' {printf("\nVALID");}  
| st '\n'  
|  
| error '\n' {yyerror("\nINVALID");yyerrok;}  
;  
%%  
main()  
{  
yyparse();  
}  
  
yylex()  
{  
char c;  
while((c=getchar())==' ');  
if(islower(c)) return LETTER;  
if(isdigit(c)) return DIGIT;  
return c;  
}
```

```
yyerror(char *s)
{
    printf("%s",s);
}
```

## **5. Write a yacc program to evaluate an expression (simple calculator program).**

```
/*      Lex program to send tokens to the Yacc program      */
%{
```

```
#include" y.tab.h"
```

```
expern int yylval;
```

```
%}
```

```
%%
```

```
[0-9]  digit
```

```
char[_a-zA-Z]
```

```
id      {char} ({ char } | {digit })*
```

```
%%
```

```
{digit}+ {yylval = atoi (yytext);
```

```
return num;
```

```
}
```

```
{id}   return name
```

```
[ \t]   ;
```

```
\n   return 0;
```

```
.   return yytext [0];
```

```
% %

/*      Yacc Program to work as a calculator      */

% {

#include<stdio.h>

#include <string.h>

#include <stdlib.h>

}

% token num name

% left '+' '-'

% left '*' '/'

% left unaryminus

% %

st      : name '=' expn

| expn { printf ("%d\n" $1); }

;

expn   : num { $$ = $1 ; }

| expn '+' num { $$ = $1 + $3; }

| expn '-' num { $$ = $1 - $3; }

| expn '*' num { $$ = $1 * $3; }

| expn '/' num { if (num == 0)

{ printf ("div by zero \n");
```

SVIT

```
        exit (0);

    }

    else

    { $$ = $1 / $3; }

| '(' expn ')' { $$ = $2; }

;

%%

main()

{

    yyparse();

}

yyerror (char *s)

{

    printf("%s", s);

}


```

## 5. Write a yacc program to recognize the grammar { a<sup>n</sup>b for n >= 0}.

```
/* Lex program to pass tokens to yacc program */

%
```

```
#include "y.tab.h"

%}

[a] { return a ; printf("returning A to yacc \n"); }
```

```
[b] return b  
  
[\n] return yytex[0];  
  
. return error;  
  
%%  
  
/*      Yacc program to check the given expression      */  
  
% {  
#include<stdio.h>  
  
% }  
  
% token a b error  
  
%%  
  
input  : line  
  
| error  
  
;  
  
line   : expn '\n' { printf(" valid new line char \n"); }  
  
;  
  
expn   : aa expn bb  
  
| aa  
  
;  
  
aa     : aa a
```

```
| a  
;  
bb    : bb b  
| b  
;  
error : error { yyerror( " " ); }  
  
%%  
main()  
{  
    yyparse();  
}  
yyerror (char *s)  
{  
    printf("%s", s);  
}  
  
/* Yacc to evaluate the expression and has c program for tokens */  
%{  
/* 6b.y {A^NB N >=0} */  
  
#include <stdio.h>
```

```
% }

%token A B

%%

st:st reca endb '\n'    {printf("String belongs to grammar\n");}
| st endb '\n'          {printf("String belongs to grammar\n");}

| st '\n'
| error '\n'           {yyerror ("\nDoes not belong to grammar\n");yyerrok;}
|
;

reca: reca enda | enda;
enda:A;
endb:B;

%%

main()
{
    yyparse();
}

yylex()
{
    char c;
    while((c=getchar())==' ');
    if(c=='a')
        return A;
    if(c=='b')
        return B;
    return c;
}

yyerror(char *s)
{
    fprintf(stdout,"%s",s);
}
```

SVIT

## 7. Write a program to recognize the grammar { a<sup>n</sup>b<sup>n</sup> | n >= 0 }

```
/*      Lex program to send tokens to yacc program      */
```

```
% {
```

```
#include "y.tab.h"
```

```
% }
```

```
[a] {return A ; printf("returning A to yacc \n"); }
```

```
[b] return B
```

```
[\\n] return yytext[0];
```

```
. return error;
```

```
%%
```

```
/*      yacc program that evaluates the expression  */
```

```
% {
```

```
#include<stdio.h>
```

```
% }
```

```
% token a b error
```

```
%%
```

```
input : line
```

```
| error
```

```
;  
line   : expn '\n' { printf(" valid new line char \n"); }  
;  
expn   : aa expn bb  
|  
;  
error  : error { yyerror ( " " ); }  
  
%%  
  
main()  
{  
    yyparse();  
}  
  
yyerror (char *s)  
{  
    printf("%s", s);  
}  
  
/*      Yacc program which has its own c program to send tokens */  
%{  
/* 7b.y  {A^NB^N N >=0}  */
```

SVIT

```
#include <stdio.h>
%
%token A B
%%
st:st reca endb '\n'    {printf("String belongs to grammar\n");}
| st '\n'          {printf("N value is 0,belongs to grammar\n");}
|
| error '\n'
            {yyerror ("\nDoes not belong to grammar\n");yyerrok;}
;
reca: enda reca endb | enda;
enda:A;
endb:B;
%%
main()
{
    yyparse();
}
yylex()
{
    char c;
    while((c=getchar())==' ');
    if(c=='a')
        return A;
    if(c=='b')
        return B;
    return c;
}
yyerror(char *s)
{
    fprintf(stdout,"%s",s);
}
```

SVIT

}

## 8. Write a Yacc program to identify a valid IF statement or IF-THEN-ELSE statement.

```
/*      Lex program to send tokens to yacc program      */
```

```
%{  
#include "y.tab.h"  
%}  
CHAR [a-zA-Z0-9]  
%x CONDSTART  
%%  
<*>[ ] ;  
<*>[ \t\n]+ ;  
<*><<EOF>> return 0;  
if return(IF);  
else return(ELSE);  
then return(THEN);  
\( {BEGIN(CONDSTART);return('(');}  
<CONDSTART>{CHAR}+ return COND;  
<CONDSTART>) {BEGIN(INITIAL);return(''))}  
{CHAR}+ return(STAT) ;  
%%
```

```
/* Yacc program to check for If and IF Then Else statement      */
```

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

```
%token IF COND THEN STAT ELSE
%%

Stat:IF '(' COND ')' THEN STAT {printf("\n VALID Statement");}
| IF '(' COND ')' THEN STAT ELSE STAT {printf("\n VALID Statement");}
|
;

%%

main()
{
    printf("\n enter statement ");
    yyparse();
}
yyerror (char *s)
{
    printf("%s",s);
}

/*      Yacc program that has c program to send tokens      */

%{

#include <stdio.h>

#include <ctype.h>

%}

%token if simple

% noassoc reduce

% noassoc else

%%
```

```
start    : start st '\n'
           |
           ;
st       : simple
           | if_st
           ;
if_st   : if st %prec reduce { printf("simple\n"); }
           | if st else st      {printf ("if_else \n"); }
           ;
%%%
int yylex()
{
    int c;
    c = getchar();
    switch ( c )
    {
        case 'i' : return if;
        case 's' : return simple;
        case 'e' : return else;
        default : return c;
    }
}
```

SVIT

```
}
```

```
main ()
```

```
{
```

```
    yy parse();
```

```
}
```

```
yyerror (char *s)
```

```
{
```

```
    printf("%s", s);
```

```
}
```

## RECOMMENDED QUESTIONS:

1. give the specification of yacc program? give an example? (8)
2. what is grammar? How does yacc parse a tree? (5)
3. how do you compile a yacc file? (5)
4. explain the ambiguity occurring in an grammar with an example? (6)
5. explain shift/reduce and reduce/reduce parsing ? (8)
6. write a yacc program to test the validity of an arithmetic expressions? (8)
7. write a yacc program to accept strings of the form  $a^n b^n$ ,  $n > 0$ ? (8)

# Chapter 5: Syntax Directed Translation

## What are we studying in this chapter?

- ◆ Syntax Directed Definitions
- ◆ Evaluation orders for SDD's
- ◆ Application of Syntax-Directed Translation
- ◆ Syntax directed translation schemes
- ◆ Implementing L-attributed SDD's

- 6 hours

### 5.1 Introduction

In the previous chapters, we have discussed first two phases of the compiler i.e., lexical analysis phase and syntax analysis phase. In this section, let us concentrate on the third phase of the compiler called *semantic analysis*. The main goal of the semantic analysis is to check for correctness of program and enable proper execution.

We know that the job of the parser is only to verify that the input program consists of tokens arranged in syntactically valid combination. In semantic analysis we check whether they form a sensible set of instructions in the programming language. For example,

- ◆ If an identifier is already declared, semantic analyzer checks whether the type of an identifier is respected in all expressions and statements used in the program. That is, it checks whether the type of RHS of an expression of an assignment statement match the type on LHS. It also checks whether the LHS needs to be properly declared and is it an assignable identifier or not.

**Ex 1:** int a = 10 + 20; // valid

**Ex 2:** char b[] = "Hello"; // valid

int a = 10 + b; // syntactically valid but, semantically invalid statement;  
// invalid usage of identifier b in the expression

**Ex 3:** int a[5][5]; // syntactically valid

a = 10 + 20; // syntactically valid but, semantically invalid statement;

- ◆ It checks whether the type and number of parameters in the function definition and the type and number of arguments in the function call are same or not. If not appropriate error messages are displayed
- ◆ It checks whether the type of operands are same in an arithmetic operation. If not it displays appropriate error messages
- ◆ The index variable used in an array must be integer type else it is a semantic error.
- ◆ Appropriate type conversion is also done by semantic analysis phase

## 5.2 □ Syntax Directed Translation

---

Now, let us see “[What is semantic analysis?](#)”

**Definition:** Semantic analysis is the third phase of the compiler which acts as an interface between syntax analysis phase and code generation phase. It accepts the parse tree from the syntax analysis phase and adds the semantic information to the parse tree and performs certain checks based on this information. It also helps constructing the symbol table with appropriate information. Some of the actions performed semantic analysis phase are:

- ◆ Type checking i.e., number and type of arguments in function call and in function header of function definition must be same. Otherwise, it results in semantic error.
- ◆ Object binding i.e., associating variables with respective function definitions
- ◆ Automatic type conversion of integers in mixed mode of operations
- ◆ Helps intermediate code generation.
- ◆ Display appropriate error messages

The semantics of a language can be described very easily using two notations namely:

- ◆ Syntax directed definition (SDD)
- ◆ Syntax directed translation (SDT)

First, we shall discuss about *syntax-directed definition* and next we shall discuss about *syntax-directed translation*.

**Note:** Consider the production  $E \rightarrow E_1 + T$ . To distinguish  $E$  on LHS of the production and  $E$  on RHS of the production, we use  $E_1$  on RHS of the production as shown below:

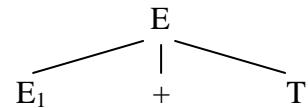
$$E \rightarrow E_1 + T$$

Now, let us consider the production, its derivation and corresponding parse tree as shown below:

<u>Production</u>	<u>Derivation</u>	<u>Parse tree</u>
-------------------	-------------------	-------------------

$$E \rightarrow E_1 + T$$

$$E \Rightarrow E_1 + T$$



- ◆ The non-terminal  $E$  on LHS of the production is called “*head of the production*”
- ◆ The string of grammar symbols “ $E + T$ ” on RHS of the production is called “*body of the production*”
- ◆ So, in the derivation, head of the production will be the parent node and the symbols that represent body of the production will be the children nodes.

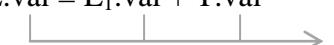
## Introduction to Compiler Design - 5.3

### 5.2 Syntax Directed Definition

Now, we shall see “What is syntax directed definition (SDD)?”

**Definition:** A *syntax-directed definition* (SDD) is a context free grammar with *attributes* and *semantic rules*. The attributes are associated with grammar symbols whereas the semantic rules are associated with productions. The semantic rules are used to compute the attribute values.

For example, a simple SDD for the production  $E \rightarrow E_1 + T$  can be written as shown below:

<u>Production</u>	<u>Semantic Rule</u>
$E \rightarrow E_1 + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$  attributes

Observe that a semantic rule is associated with production where the attribute name *val* is associated with each non-terminal used in the rule.

Now, let us see “What is an attribute? Explain with example”

**Definition:** An *attribute* is a property of a programming language construct. Attributes are always associated with grammar symbols. If  $X$  is a grammar symbol and  $a$  is the attribute, then  $X.a$  denote the value of attribute  $a$  at a particular node  $X$  in the parse tree. If we implement the nodes of the parse tree by records or using structures, then the attribute of  $X$  can be implemented as a field in the record or a structure.

- ◆ **Ex 1:** If *val* is the attribute associated with a non-terminal  $E$ , then  $E.\text{val}$  gives the value of attribute *val* at a node  $E$  in the parse tree.
- ◆ **Ex 2:** If *lexval* is the attribute associated with a terminal **digit**, then **digit.lexval** gives the value of attribute *lexval* at a node **digit** in the parse tree.
- ◆ **Ex 3:** If *syn* is the attribute associated with a non-terminal  $F$ , then  $F.\text{syn}$  gives the value of attribute *syn* at a node  $F$  in the parse tree.

Typical examples of attributes are:

- ◆ The data types associated with variable such as **int**, **float**, **char** etc
- ◆ The value of an expression
- ◆ The location of a variable in memory
- ◆ The object code of a function or a procedure
- ◆ The number of significant digits in a number and so on.

Now, let us see “What is a semantic rule?” Explain with example”

## 5.4 □ Syntax Directed Translation

---

**Definition:** The rule that describe how to compute the attribute values of the attributes associated with a grammar symbol using attribute values of other grammar symbols is called *semantic rule*.

For example, consider the production  $E \rightarrow E_1 + T$ . The attribute value of  $E$  which is on LHS of the production denoted by  $E.val$  can be calculated by adding the attribute values of variables  $E$  and  $T$  which are on RHS of the production denoted by  $E_1.val$  and  $T.val$  as shown below:

$$E.val = E_1.val + T.val \quad // \text{ Semantic rule}$$

### 5.3 Inherited and Synthesized attributes

The attribute value for a node in the parse tree may depend on information from its children nodes or its sibling nodes or parent nodes. Based on how the attribute values are obtained we can classify the attributes. Now, let us see “**What are the different types or classifications of attributes?**” There are two types of attributes namely:

- Synthesized attribute
- Inherited attribute

#### 5.3.1 Synthesized attribute

Now, let us see “**What is synthesized attribute? Explain with example**”

**Definition:** The attribute value of a non-terminal  $A$  derived from the attribute values of its children or itself is called *synthesized attribute*. Thus, the attribute values of synthesized attributes are passed up from children to the parent node in bottom-up manner.

For example, consider the production:  $E \rightarrow E_1 + T$ . Suppose, the attribute value  $val$  of  $E$  on LHS (head) of the production is obtained by adding the attribute values  $E_1.val$  and  $T.val$  appearing on the RHS (body) of the production as shown below:

<u>Production</u>	<u>Semantic Rule</u>	<u>Parse tree with attribute values</u>
$E \rightarrow E + T$	$E.val = E_1.val + T.val$	$\begin{array}{ccc} & E.val = 30 & \\ &   & \\ E_1.val = 10 & + & T.val = 20 \end{array}$

Now, attribute  $val$  with respect to  $E$  appearing on head of the production is called *synthesized attribute*. This is because, the value of  $E.val$  which is 30, is obtained from the children by adding the attribute values 10 and 20 as shown in above parse tree.

## Introduction to Compiler Design - 5.5

### 5.3.2 Inherited attribute

Now, let us see “What is inherited attribute? Explain with example”

**Definition:** The attribute value of a non-terminal A derived from the attribute values of its *siblings* or from its *parent* or *itself* is called *inherited attribute*. Thus, the attribute values of inherited attributes are passed from siblings or from parent to children in top-down manner. **For example**, consider the production:  $D \rightarrow T V$  which is used for a single declaration such as:

*int sum*

In the production, D stands for declaration, T stands for *type* such as *int* and V stands for the variable *sum* as in above declaration. The production, semantic rule and parse tree along with attribute values is shown below:

<u>Production</u>	<u>Parse tree with attribute values</u>	<u>Semantic Rule</u>
$D \rightarrow T V$	<pre>graph TD; D[D] --&gt; T1[T.type = int]; D --&gt; V1[V.inh = int]; T1 -.-&gt; V1; V1 --&gt; id[id.entry]</pre>	$V.inh = T.type$

Observe the following points from the above parse tree:

- ♦ The type **int** obtained from the lexical analyzer is already stored in  $T.type$  whose value is transferred to its sibling  $V$ . This can be done using:

$V.inh = T.type$

Since attribute value for  $V$  is obtained from its sibling, it is inherited attribute and its attribute is denoted by *inh*.

- ♦ On similar line, the value **int** stored in  $V.inh$  is transferred to its child  $id.entry$  and hence *entry* is inherited attribute of  $id$  and attribute value is denoted by  $id.entry$

**Note:** With the help of the annotated parse tree, it is very easy for us to construct SDD for a given grammar.

### 5.3.3 Annotated parse tree

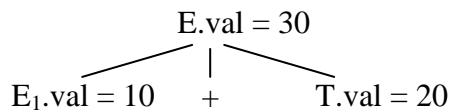
Before proceeding further, it is necessary to know the definition of annotated parse tree. Now, let us see “What is annotated parse tree? Explain with example”

## 5.6 □ Syntax Directed Translation

---

**Definition:** A parse tree showing the attribute values of each node is called *annotated parse tree*. The terminals in the annotated parse tree can have only synthesized attribute values and they are obtained directly from the lexical analyzer. So, there are no semantic rules in SDD (short form Syntax Directed Definition) to get the lexical values into terminals of the annotated parse tree. The other nodes in the annotated parse tree may be either synthesized or inherited attributes. **Note:** Terminals can never have inherited attributes

For example, consider the partial annotated tree shown below:



In the above partial annotated parse tree, the attribute values 10, 20 and 30 are stored in  $\text{E}_1.\text{val}$ ,  $\text{T}. \text{val}$  and  $\text{E}. \text{val}$  respectively.

---

**Example 5.1:** Write the SDD for the following grammar:

$$\begin{array}{ll} S \rightarrow En & \text{where } n \text{ represent end of file marker} \\ E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid \text{digit} \end{array}$$

---

**Solution:** The given grammar is shown below:

$$\begin{array}{l} S \rightarrow En \\ E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid \text{digit} \end{array}$$

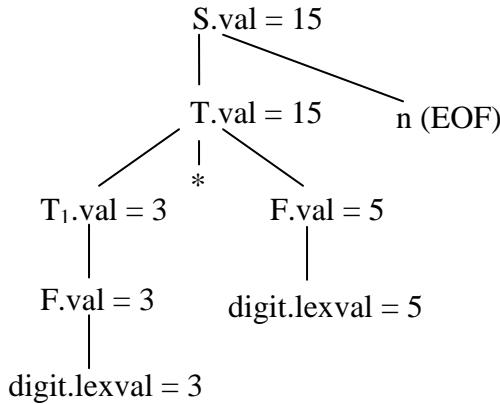
The above grammar generates an arithmetic expression consisting of parenthesized or un-parenthesized expression with operators  $+$  and  $*$ . For the sake of convenience, let us consider part of the grammar written as shown below:

$$\begin{array}{l} S \rightarrow Tn \\ T \rightarrow T * F \mid F \\ F \rightarrow \text{digit} \end{array}$$

Using the above productions we can generate an un-parenthesized expression consisting of only  $*$  operator such as:  $3*4$  or  $3*4*5$  etc. The annotated parse tree for evaluating the expression  $3*5$  is shown below:

## ■ Introduction to Compiler Design - 5.7

---



It is very easy to see how the values 3 and 5 are moved from bottom to top (propagated upwards) till we reach the root node to get the value 15. The rules to get the value 15 from the productions used are shown below:

<u>Productions</u>	<u>Semantic rules</u>
$F \rightarrow \text{digit}$	$F.\text{val} = \text{digit}.lexval$
$T \rightarrow F$	$T.\text{val} = F.\text{val}$
$T \rightarrow T_1 * F$	$T.\text{val} = T_1.\text{val} * F.\text{val}$
$S \rightarrow Tn$	$S.\text{val} = T.\text{val}$

On similar lines we can write the semantic rules for the following productions as shown below:

<u>Productions</u>	<u>Semantic rules</u>
$S \rightarrow En$	$S.\text{val} = E.\text{val}$
$E \rightarrow E_1 + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$
$E \rightarrow T$	$E.\text{val} = T.\text{val}$
$F \rightarrow (E)  $	$F.\text{val} = E.\text{val}$

Now, the final SDD along with productions and semantic rules is shown below:

<u>Productions</u>	<u>Semantic Rules</u>
$S \rightarrow En$	$S.\text{val} = E.\text{val}$
$E \rightarrow E + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$
$E \rightarrow T$	$E.\text{val} = T.\text{val}$
$T \rightarrow T * F$	$T.\text{val} = T_1.\text{val} * F.\text{val}$
$T \rightarrow F$	$T.\text{val} = F.\text{val}$
$F \rightarrow (E)$	$F.\text{val} = E.\text{val}$
$F \rightarrow \text{digit}$	$F.\text{val} = \text{digit}.lexval$

---

**Example 5.2:** Write the grammar and syntax directed definition for a simple desk calculator and show annotated parse tree for the expression  $(3+4)*(5+6)$

---

## 5.8 □ Syntax Directed Translation

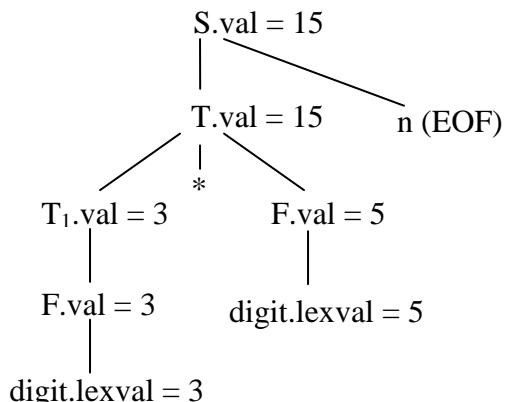
**Solution:** A simple desk calculator performs operations such as addition, subtraction, division and multiplication with or without parenthesis. The grammar for obtaining an arithmetic expression with or without parentheses can be written as shown below:

$$\begin{aligned} S &\rightarrow En \\ E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T / F \mid F \\ F &\rightarrow (E) \mid \text{digit} \end{aligned}$$

The above grammar generates an arithmetic expression consisting of parenthesized or un-parenthesized expression with operators +, -, \* and / operators. For the sake of convenience, let us consider part of the grammar written as shown below:

$$\begin{aligned} S &\rightarrow Tn \\ T &\rightarrow T * F \mid F \\ F &\rightarrow \text{digit} \end{aligned}$$

Using the above productions we can generate an un-parenthesized expression consisting of only \* operator such as:  $3*4$  or  $3*4*5$  etc. The annotated parse tree for evaluating the expression  $3*5$  is shown below:



It is very easy to see how the values 3 and 5 are moved from bottom to top (propagated upwards) till we reach the root node to get the value 15. The rules to get the value 15 from the productions used are shown below:

<u>Productions</u>	<u>Semantic rules</u>
$F \rightarrow \text{digit}$	$F.val = \text{digit}.lexval$
$T \rightarrow F$	$T.val = F.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$S \rightarrow Tn$	$S.val = T.val$

On similar lines we can write the semantic rules for the following productions as shown below:

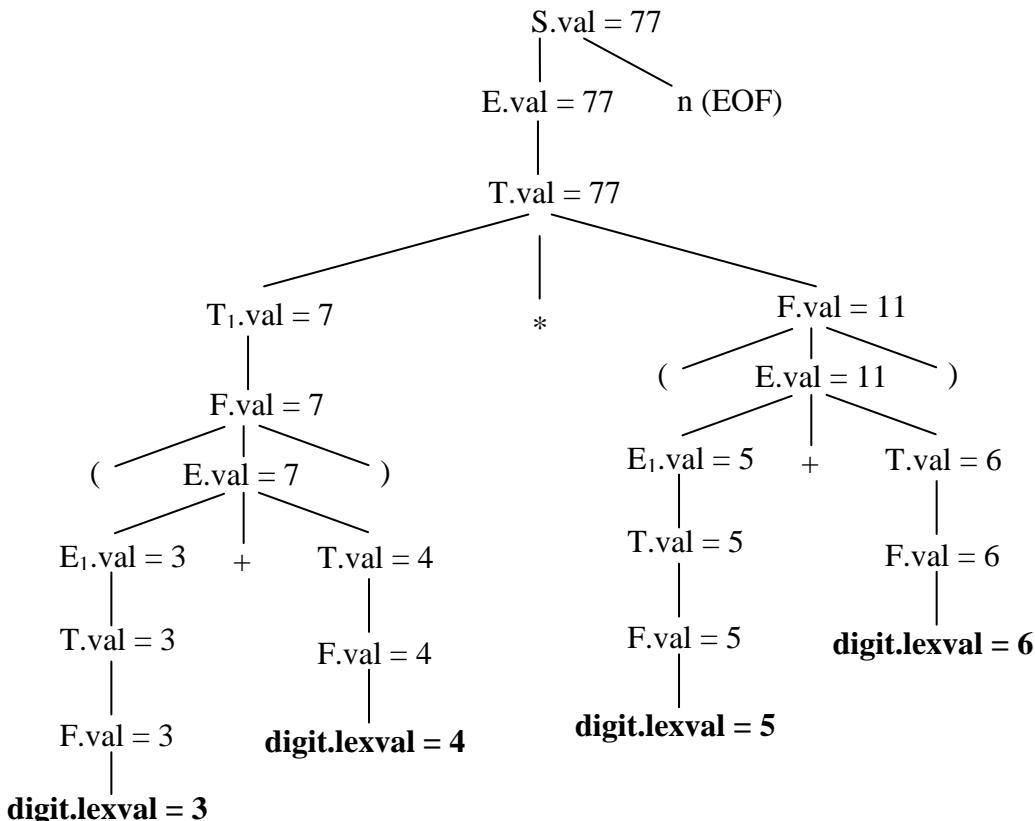
<u>Productions</u>	<u>Semantic rules</u>
$S \rightarrow En$	$S.val = E.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow E_1 - T$	$E.val = E_1.val - T.val$
$T \rightarrow T_1 / F$	$T.val = T_1.val + F.val$
$E \rightarrow T$	$E.val = T.val$
$F \rightarrow (E)$	$F.val = E.val$

## ■ Introduction to Compiler Design - 5.9

So, the final SDD for simple desk calculator can be written as shown below:

<u>Productions</u>	<u>Semantic Rules</u>
$S \rightarrow E_n$	$S.val = E.val$
$E \rightarrow E + T$	$E.val = E_1.val + T.val$
$E \rightarrow E - T$	$E.val = E_1.val - T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T * F$	$T.val = T_1.val * F.val$
$T \rightarrow T / F$	$T.val = T_1.val / F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$

The annotated parse tree for the expression  $(3+4)*(5+6)$  consisting of attribute values for each non-terminal is shown below:



## 5.10 □ Syntax Directed Translation

---

### 5.4 Evaluating an SDD at the Nodes of a Parse Tree

We can easily obtain an SDD using the following two steps:

**Step 1:** Construct the parse tree

**Step 2:** Use the rules to evaluate attributes of all the nodes of the parse tree.

**Step 3:** Obtain the attribute values for each non-terminal and write the semantic rules for each production. When complete annotated parse tree is ready, we will have the complete SDD

Now, the question is “How do we construct an annotated parse tree? In what order do we evaluate attributes?”

- ♦ If we want to evaluate an attribute of a node of a parse tree, it is necessary to evaluate all the attributes upon which its value depends.
- ♦ If all attributes are synthesized, then we must evaluate the attributes of all of its children before we can evaluate the attribute of the node itself.
- ♦ With synthesized attributes, we can evaluate attributes in any bottom up order.
- ♦ Whether synthesized or inherited attributes there is no single order in which the attributes have to be evaluated. There can be one or more orders in which the evaluation can be done.

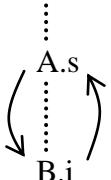
For example, to see how an annotated parse tree can be constructed, *refer example 5.1.*

#### 5.4.1 Circular dependency

Before proceeding further, let us see, “What is circular dependency when evaluating the attribute value of a node in an annotated parse tree?”

**Definition:** If the attribute value of a parent node depends on the attribute value of child node and vice-versa, then we say, there exists a **circular dependency** between the child node and parent node. In this situation, it is not possible to evaluate the attribute of either parent node or the child node since one value depends on another value.

For example, consider the non-terminal A with synthesized attribute A.s and non-terminal B with inherited attribute B.i with following productions and semantic rules:

<u>Production</u>	<u>Semantic rule</u>	<u>Partial annotated parse tree</u>
$A \rightarrow B$	$A.s = B.i$ $B.i = A.s + 6$	

**Note:** In the above semantic rule *s* is synthesized attribute and *i* is inherited attribute

## ■ Introduction to Compiler Design - 5.11

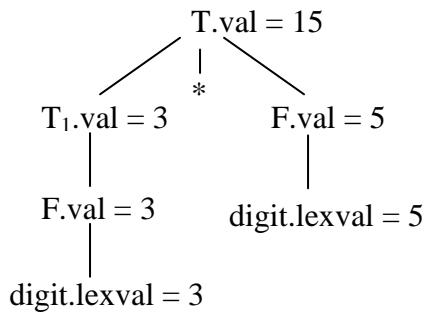
Note that the above two semantic rules are circular in nature (see above figure). Note that to compute A.s we require the value of B.i and to compute the value of B.i, we require the value of A.s. So, it is impossible to evaluate either the value of A.s or the value of B.i without evaluating other.

### 5.4.2 Why evaluate inherited attributes (Evaluate inherited attributes)

Now, let us see “What is the use of inherited attributes?” The reason for inherited attributes can be explained using an example. Consider the following grammar:

$$\begin{aligned} T &\rightarrow T * F \mid F \\ F &\rightarrow \text{digit} \end{aligned}$$

Using the above productions we can generate an un-parenthesized expression consisting of only \* operator such as: 3\*4 or 3\*4\*5 etc. The above grammar has left-recursion and it is suitable for *bottom up parser* such as LR parser. The annotated parse tree for evaluating the expression 3\*5 is shown below:



It is very easy to see how the values 3 and 5 are moved from bottom to top (propagated upwards) till we reach the root node to get the value 15 as shown below:

Semantic rules	Productions
$F.\text{val} = \text{digit}.lexval$	$F \rightarrow \text{digit}$
$T.\text{val} = F.\text{val}$	$T \rightarrow F$
$T.\text{val} = T_1.\text{val} * F.\text{val}$	$T \rightarrow T * F$

Thus, using bottom-up manner, the values 3 and 5 are moved upwards to get the result 15 using the semantic rules associated with each production. So, far there is no problem.

**Example 5.3:** Obtain SDD and annotated parse tree for the following grammar using top-down approach:

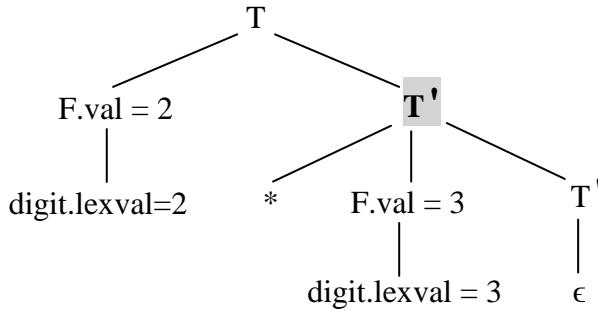
$$\begin{aligned} T &\rightarrow T * F \mid F \\ F &\rightarrow \text{digit} \end{aligned}$$

Let us see, “What happens if top-down parser is used to parser the input 3\*5 using the above grammar?” We have already seen earlier in previous chapters that when we use top-down parser, first we have to eliminate left-recursion and then do parsing. So, the given grammar is not suitable for top-down parser because of left-recursion. So, let us remove left-recursion. The grammar obtained after removing left-recursion is shown below:

## 5.12 □ Syntax Directed Translation

$$\begin{aligned} T &\rightarrow FT' \\ T' &\rightarrow *FT_1' \\ T' &\rightarrow \epsilon \\ F &\rightarrow \text{digit} \end{aligned}$$

To construct SDD for the above grammar, let us obtain the derivation tree for the expression  $2*3$  with some of the attributes as shown below:



Observe the following points from above partial annotated parse tree

- The values 2, 3 and 4 are moved upwards till we get node F. Since the attribute value of F is obtained from its child, the attribute of F denoted by *val* will be synthesized attribute and F.val can be obtained using semantic rule by considering the production  $F \rightarrow \text{digit}$  as shown below:

<u>Production</u>	<u>Semantic Rule</u>	<u>Type</u>
$F \rightarrow \text{digit}$	$F.\text{val} = \text{digit}.lexval$	synthesized

- Now, we need to multiply  $2*3$ . Observe that there is no node with 2, \* and 3 as the children. So, we can perform  $2*3$  using the production  $T' \rightarrow *FT_1'$  from the top as shown below:

$$\begin{array}{c} T' \rightarrow *FT_1' \\ \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \\ 2 \quad * \quad F \quad T_1' \\ \quad \quad \quad | \\ \quad \quad \quad 3 = 6 \end{array}$$

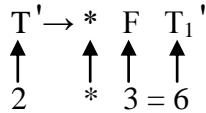
Now, the question is how to transfer 2 to  $T'$ , how to multiply  $2*3$  and how to store the result 6 in  $T_1'$ . These activities can be done as shown below:

- Observe from the partial annotated parse tree and above scenario that, the first operand 2 already present in F.val which is the left child of T must be transferred to right child  $T'$  using the production  $T \rightarrow FT'$  as shown below:

<u>Production</u>	<u>Semantic Rule</u>	<u>Type</u>
$T \rightarrow FT'$	$T'.\text{inh} = F.\text{val}$	Inherited

## ■ Introduction to Compiler Design - 5.13

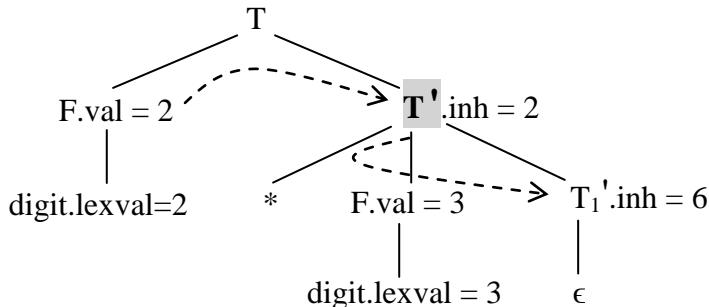
- 2) Now,  $2 * 3$  can be computed as shown below:



Observe from above figure that multiply  $2 * 3$  means we need to compute  $T' * F$  and store the result in  $T_1'$ . That is, take the inherited attribute value  $T'.inh$  and multiply with synthesized attribute value  $F.val$  and store the result in  $T_1'.inh$ . This can be done using as shown below:

<u>Production</u>	<u>Semantic Rule</u>	<u>Type</u>
$T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh * F.val$	Inherited

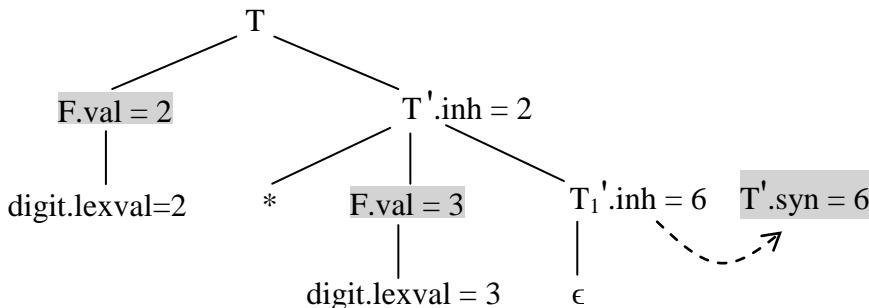
**Note:** Since  $T_1'.inh$  is computed using sibling attribute values, it is inherited attribute. Now, the partial annotated parse tree showing the attribute values computed so far is shown below:



- Observe from above figure that the node  $T_1'.inh$  which is same as  $T'.inh$  produce  $\epsilon$  and so, the synthesized attribute value  $T'.syn$  can be obtained using the production  $T' \rightarrow \epsilon$  and its semantic rule as shown below:

<u>Production</u>	<u>Semantic Rule</u>	<u>Type</u>
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$	Synthesized

The partial annotated parse tree is shown below:

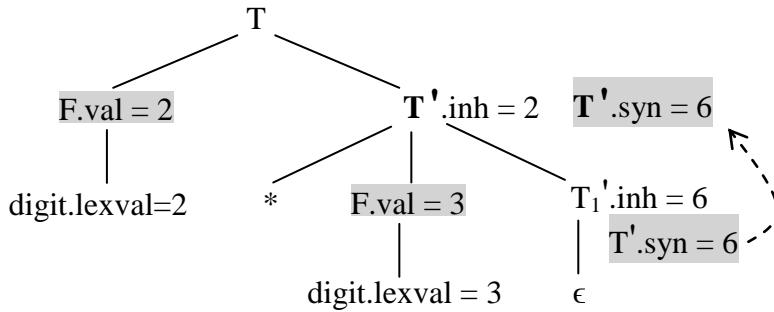


## 5.14 □ Syntax Directed Translation

- The synthesized value  $T'.syn = 6$  which we call as  $T_1'.syn$  is transferred to its parent  $T'$  with attribute value  $T'.syn = 6$  using the production  $T' \rightarrow * F T_1'$  as shown below:

<u>Production</u>	<u>Semantic Rule</u>	<u>Type</u>
$T' \rightarrow * F T_1'$	$T'.syn = T_1'.sys$	Synthesized

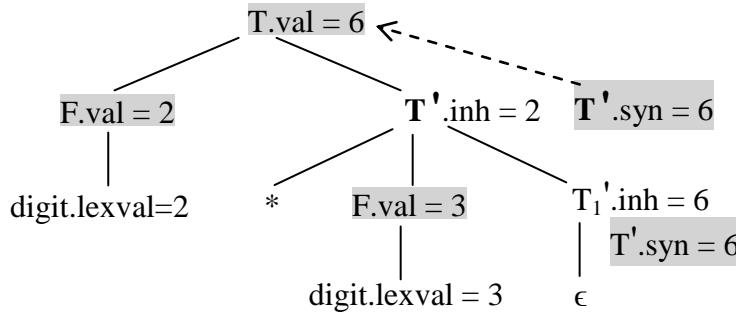
The partial annotated parse tree is shown below:



- Finally  $T'.syn = 6$  is transferred to its parent  $T$  using the production  $T \rightarrow F T'$  as shown below:

<u>Production</u>	<u>Semantic Rule</u>	<u>Type</u>
$T \rightarrow F T'$	$T.val = T'.syn$	Synthesized

The annotated parse tree that shows the value of  $T.val$  is shown below:



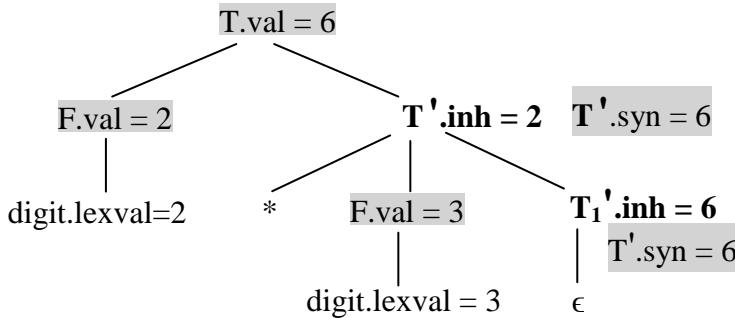
So, the final SDD for the given grammar can be written as shown below:

<u>Productions</u>	<u>Semantic Rules</u>	<u>Type</u>
$T \rightarrow F T'$	$T'.inh = F.val$	Inherited
	$T.val = T'.syn$	Synthesized

## ■ Introduction to Compiler Design - 5.15

$T' \rightarrow * F T_1'$	$T_1'.inh = T'.inh * F.val$	Inherited
	$T'.syn = T_1'.sys$	Synthesized
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$	Synthesized
$F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$	Synthesized

The final annotated parse tree is shown below:



**Example 5.4:** Obtain SDD for the following grammar using top-down approach:

$$\begin{aligned} S &\rightarrow En \\ E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{digit} \end{aligned}$$

and obtain annotated parse tree for the expression  $(3 + 4) * (5 + 6)n$

**Solution:** The given grammar has left recursion and hence it is not suitable for top-down parser. To make it suitable for top-down parsing, we have to eliminate left recursion. After eliminating left recursion (see example 2.13 for details), the following grammar is obtained:

$$\begin{aligned} S &\rightarrow En \\ E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid \epsilon \\ T &\rightarrow F T' \\ T' &\rightarrow * F T_1' \mid \epsilon \\ F &\rightarrow (E) \mid \text{digit} \end{aligned}$$

**Note:** The variables S, E, T and F are present both in given grammar and grammar obtained after left recursion. So, only for the variables S, E, T and F we use the attribute name  $v$  (stands for val) and for all other variables we use  $s$  for synthesized attribute and  $i$  for inherited attribute

Consider the following productions:

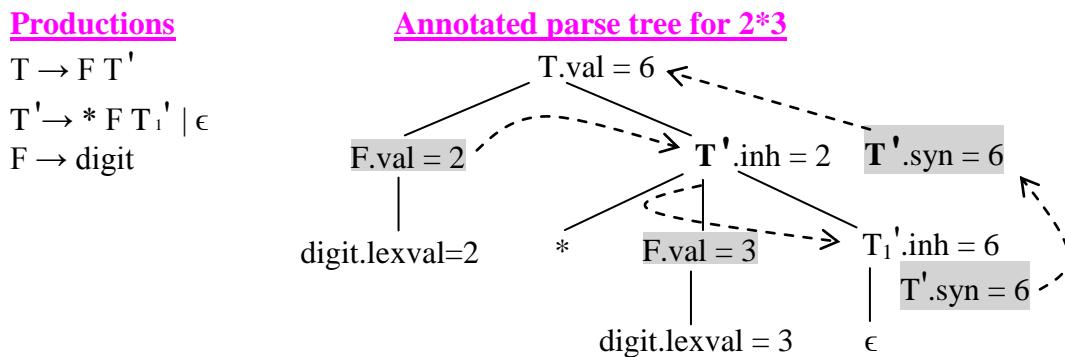
$$\begin{aligned} S &\rightarrow En \\ F &\rightarrow (E) \mid \text{digit} \end{aligned}$$

## 5.16 □ Syntax Directed Translation

They do not have left recursion and they are retained in the grammar which is obtained after eliminating left recursion. So, we can compute the attribute value of LHS (head) from the attribute values of RHS (i.e., children) for the above productions and hence they have synthesized attributes. The productions, semantic rules and type of the attribute are shown below:

<u>Production</u>	<u>Semantic Rule</u>	<u>Type</u>
$S \rightarrow E\ n$	$S.v = E.v$	Synthesized
$F \rightarrow ( E )$	$F.v = E.v$	Synthesized
$F \rightarrow d$	$F.v = d.lexval$	Synthesized

Consider the following productions and write the annotated parse tree for the expression  $2*3$  with flow of information (See example 5.3 for detailed explanation) as shown below:



By following the dotted arrow lines, we can write the various semantic rules for the corresponding productions as shown below:

	<u>Semantic rule</u>	<u>Production</u>
$F.val = 2$ is copied to $T'.inh$	$T'.inh = F.val$	$T \rightarrow F\ T'$
$T'.inh * F.val$ is copied to $T_1'.inh$	$T_1'.inh = T'.inh * F.val$	$T' \rightarrow * F\ T_1'$
$T_1'.inh$ is copied to $T'.syn$	$T'.syn = T_1'.inh$	$T' \rightarrow \epsilon$
$T'.syn$ is moved to its parent $T$	$T'.syn = T_1'.syn$	$T' \rightarrow * F\ T_1'$
$T'.syn$ is moved to its parent $T$	$T.val = T'.syn$	$T \rightarrow F\ T'$

The above productions and their respective semantic rules along with the type of attribute are shown below:

<u>Production</u>	<u>Semantic rules</u>
$T \rightarrow F\ T'$	$T'.inh = F.val$
	$T.val = T'.syn$

## ■ Introduction to Compiler Design - 5.17

---

$T' \rightarrow * F T_1'$	$T_1'.inh = T'.inh * F.val$
	$T'.syn = T_1'.syn$
$T' \rightarrow \epsilon$	$T'.syn = T_1'.inh$

Exactly similar to the above we can write the semantic rules for the given productions as shown below:

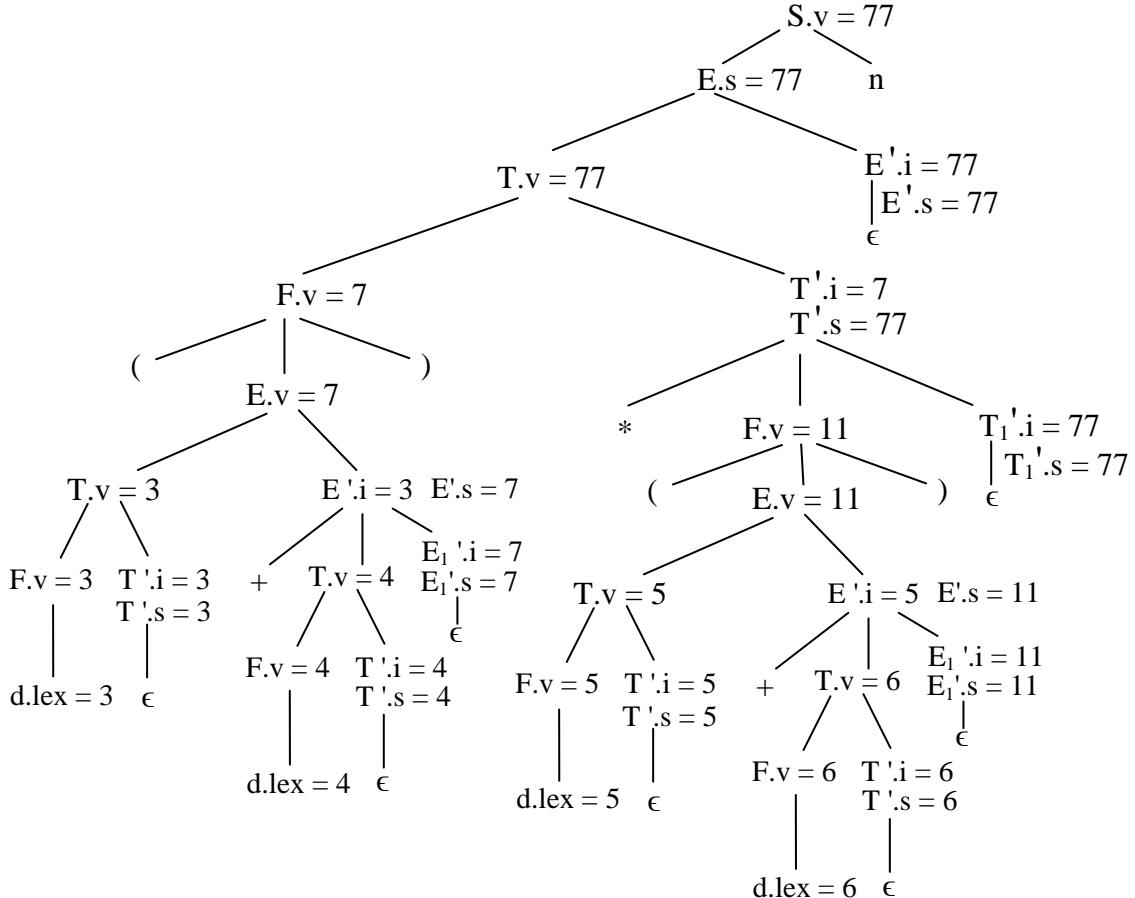
<u>Production</u>	<u>Semantic rules</u>
$E \rightarrow T E'$	$E'.inh = T.val$ $E.val = E'.syn$
$E' \rightarrow + T E_1'$	$E_1'.inh = E'.inh + T.val$ $E'.syn = E_1'.syn$
$E' \rightarrow \epsilon$	$E'.syn = E_1'.inh$

Combining all productions and semantic rules we can write the **final SDD** as shown below:

<u>Production</u>	<u>Semantic Rule</u>	<u>Type</u>
$S \rightarrow E n$	$S.v = E.v$	Synthesized
$E \rightarrow T E'$	$E'.inh = T.val$ $E.val = E'.syn$	Inherited Synthesized
$E' \rightarrow + T E_1'$	$E_1'.inh = E'.inh + T.val$ $E'.syn = E_1'.syn$	Inherited Synthesized
$E' \rightarrow \epsilon$	$E'.syn = E_1'.inh$	Synthesized
$T \rightarrow F T'$	$T'.inh = F.val$ $F.val = T'.syn$	Inherited Synthesized
$T' \rightarrow * F T_1'$	$T_1'.inh = T'.inh * F.val$ $T'.syn = T_1'.syn$	Inherited Synthesized
$T' \rightarrow \epsilon$	$T'.syn = T_1'.inh$	Synthesized
$F \rightarrow ( E )$	$F.val = E.val$	Synthesized
$F \rightarrow d$	$F.val = d.lexval$	Synthesized

## 5.18 □ Syntax Directed Translation

The annotated parse tree that shows the values of each attributed value while evaluating the following expression  $(3 + 4) * (5 + 6)$  is shown below:



## 5.5 Evaluation order for SDD's

The evaluation order to find attribute values in a parse tree using semantic rules can be easily obtained with the help of dependency graph. While annotated parse tree shows the values of attributes, a dependency graph helps us to determine how those values can be computed.

### 5.5.1 Dependency graphs

Now, let us see “What is a dependency graph?”

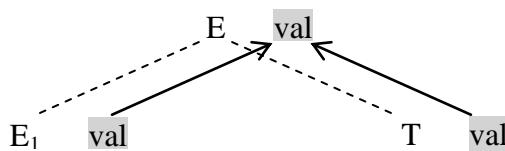
**Definition:** A graph that shows the flow of information which helps in computation of various attribute values in a particular parse tree is called *dependency graph*. An edge from one attribute instance to another attribute instance indicates that the attribute value of the first is needed to compute the attribute value of the second.

## ■ Introduction to Compiler Design - 5.19

For example, consider the following production and rule:

<b>Production</b>	<b>Semantic rule</b>
$E \rightarrow E_1 + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$

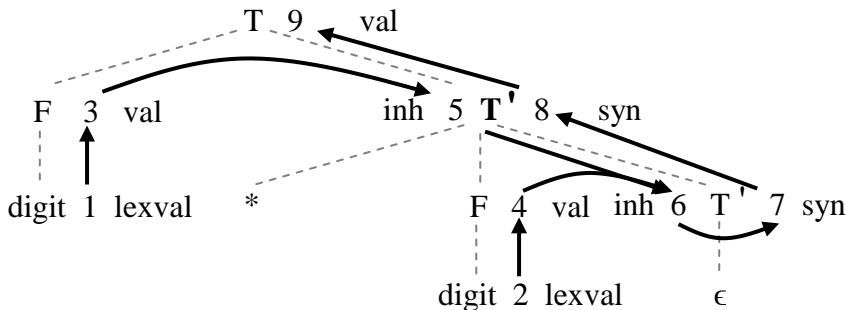
This production has left recursion and hence, it is suitable for bottom-up parser. In bottom up parser, the attribute value of LHS (head) depends on the attribute value of RHS (body of the production or children in the parse tree). So, the attribute value of  $E$  is obtained from its children  $E_1$  and  $T$ . The portion of a dependency graph for this production can be written as shown below:



In the above figure, the dotted lines along with nodes connected to them represent the parse tree. The shaded nodes represented as **val** with solid arrows originating from one node and ends in another node is the dependency graph.

**Example 5.5:** Obtain the dependency graph for the annotated parse tree obtained in example 5.3

The dependency graph for the annotated parse tree obtained in example 5.3 can be written as shown below:



Observe the following points from the above dependency graph

- ♦ Nodes identified by numbers 1 and 2 represent the attribute *lexval* which is associated with two leaves labeled **digit**
- ♦ Nodes 3 and 4 represent the attribute *val* associated with two nodes labeled **F**.
- ♦ The edge from node 1 to node 3 and from node 2 to node 4 indicates that in the semantic rule, the attribute value  $F.\text{val}$  is obtained using attribute value  $\text{digit}.lexval$
- ♦ Nodes 5 and 6 represent the inherited attribute  $T'.inh$  which is associated with each of the non-terminal  $T'$

## 5.20 □ Syntax Directed Translation

---

- ♦ The edge from 3 to 5 indicate that  $T'.inh$  is obtained from its sibling  $F.val$  and hence  $T'$  has an inherited attribute name *inh*
- ♦ The edge from 5 to 6 and another edge from 4 to 6 indicate that the two attribute values  $T'.inh$  and  $F.val$  are multiplied to get the attribute value at node 6
- ♦ The edge from 6 to 7 indicate that there is an  $\epsilon$ -production and the attribute value is obtained from itself and hence its attribute value  $T'.syn$  is obtained from  $T'.inh$
- ♦ The node 7 is obtained from itself, 8 is obtained from node 7 and 9 is obtained from node 8 and all are synthesized attributes.
- ♦ Finally,  $T.val$  at node 9 is obtained from its child at node 8.

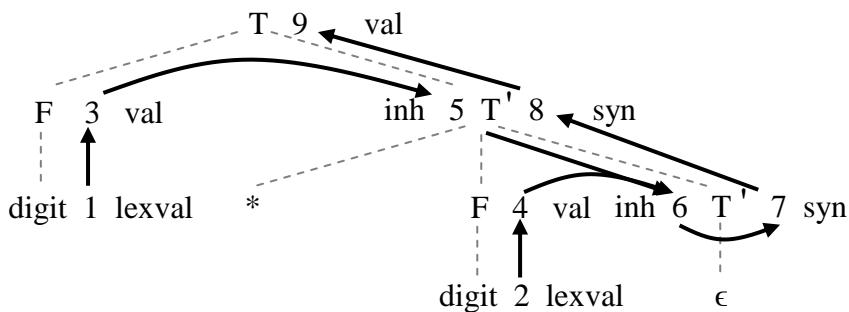
### 5.5.2 Ordering the evaluation of attributes

Now, let us see “What is topological sort of the graph?”

**Definition:** Topological sort of a directed graph is a sequence of nodes which gives the order in which the various attribute values can be computed in a parse tree. Using the dependency graph, we can write the order in which we can evaluate various attribute values in the parse tree. This ordering is nothing but the topological sort of the graph. There may be one or more orders to evaluate attribute values. If the dependency graph has an edge from A to B, then the attribute corresponding to A must be evaluated before evaluating attribute value at node B.

**Example 5.6:** Give the topological sort of the following dependency graph

---



**Solution:** The various topological sorts that can be obtained using the dependency graph are shown below:

**Topological sort 1:** 1, 2, 3, 4, 5, 6, 7, 8, 9

**Topological sort 2:** 1, 3, 2, 4, 5, 6, 7, 8, 9

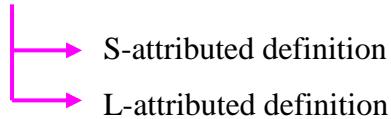
**Topological sort 3:** 1, 3, 5, 2, 4, 6, 7, 8, 9 and so on

**Note:** If there is a cycle in the dependency graph, topological sort does not exist.

### 5.5.3 S-attributed definitions

In this section, let us see Given an SDD, it is very difficult to tell whether there exist a cycle in the dependency graph corresponding a parse tree. But, in practice, translations can be implemented using classes of SDD's that will guarantee an evaluation order, since they do not permit dependency graphs with cycles.

Now, let us see “**What are different classes of SDD’s that guarantee evaluation order?**” The two classes of SDD’s that guarantee an evaluation order are:



Now, let us see “**What is S-attributed definition?**”

**Definition:** An SDD that contains only synthesized attributes is called **S-attributed**. In an S-attributed SDD, each semantic rule computes the attribute value of a non-terminal that occurs on LHS of the production that represent head of the production with the help of the attribute values of non-terminals on the right hand side of the production that represent body of the production.

For example, the SDD shown below is an S-attributed definition.

<u>Productions</u>	<u>Semantic Rules</u>
$S \rightarrow E_n$	$S.val = E.val$
$E \rightarrow E + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$

Note that all the attributes in above SDD such as  $S.val$ ,  $E.val$ ,  $T.val$  and  $F.val$  are synthesized attributes and hence the SDD is an S-attributed. Observe the following points:

- ♦ When an SDD is S-attributed, we can evaluate its attributes in any bottom up order of the nodes of the parse tree

## 5.22 □ Syntax Directed Translation

---

- ♦ S-attributed definitions can be implemented during bottom-up parsing, since a bottom-up parse corresponds to a postorder traversal.
- ♦ The postorder traversal corresponds exactly to the order in which LR parser reduces a production body (RHS of the production) to its head (LHS of the production)
- ♦ The attributes can be evaluated very easily by performing the postorder traversal of the parse tree at a node N as shown below:

```
postorder (N)
{
    for (each child C of N from left)
        postorder(C)
    end for

    evaluate the attributes associated with node N
}
```

Now, let us see “[What is an attribute grammar?](#)”

**Definition:** An SDD without any side effects is called *attribute grammar*. The semantic rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants. Attribute grammars have the following properties:

- ♦ They do not have any side effects
- ♦ They allow any evaluation order consistent with dependency graph.

For example, the SDD obtained from example 5.1 is an attribute grammar. For simplicity, the SDD’s that we have seen so far have semantic rules without side effects. But, in practice, it is convenient to allow SDD’s to have limited side effects, such as printing the result computed by a desk calculator or interacting with symbol table and so on.

### 5.5.4 L-attributed definitions

L-attributed definition is a second class of SDD. The idea behind the class is that “Between the attributes associated with a production body, dependency graph edges can go only from left to right, but not from right to left. Hence L-attributed. Now, formally, let us see “[What is an L-attributed definition?](#)”

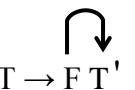
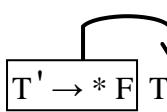
**Definition:** An L-attributed definition is one of the following:

- 1) Synthesized or
- 2) Inherited, but with following rules. Consider the production  $A \rightarrow X_1X_2\ldots\ldots X_n$  and let  $X_i.a$  is an inherited attribute. In this situation one of the following rules are applicable:

## Introduction to Compiler Design - 5.23

- a) Inherited attributes must be associated with A which is LHS of the production (called head of the production)
  - or
- b) All the symbols  $X_1, X_2, X_3, \dots, X_{i-1}$  appearing to the left of  $X_i$  have either synthesized or inherited attributes
  - or
- c) Inherited or synthesized attributes associated with this occurrence of  $X_i$  itself but without forming any cycles

For example, the SDD shown in example 5.3 is L-attributed. To see why, let us consider the following productions and semantic rules:

<u>Productions</u>	<u>Semantic rules</u>
$T \rightarrow F T'$ 	$T'.inh = F.val$
$T' \rightarrow * F T_1'$ 	$T_1'.inh = T'.inh * F.val$

Observe the following points from above two semantic rules:

- ♦ In the first rule observe that attribute value of F denoted by  $F.val$  is copied into attribute value of  $T'$  denoted by  $T'.inh$  shown using an edge going from left to right. Note that F appears to the left of  $T'$  in the production body as required
- ♦ In the second rule, the inherited attribute value of  $T'$  on LHS (head) of the production and synthesized attribute value of F present in body of the production are multiplied and the result is stored in attribute value of  $T_1'$  denoted by  $T_1'.inh$ . Note that both  $T'$  and F appears to the left of  $T_1'$  ad required by the rule.
- ♦ In each of these cases, the rules use the information “from above or from the left” as required by the class. The remaining attributes are synthesized. Hence, the SDD is L-attributed.

### 5.5.5 Semantic rules with controlled side effects

Now, let us see “What is a side effect in a SDD?”

**Definition:** The main job of the semantic rule is to compute the attribute value of each non-terminal in the corresponding parse tree. Any other activity performed other than computing the attribute value is treated as side effect in a SDD.

## 5.24 □ Syntax Directed Translation

---

For example, attribute grammars have no side effects and allow any evaluation order consistent with dependency graph. But, translation schemes impose left-to-right evaluation and allow semantic actions to contain any program fragment. In practice, translation involves side effects:

- ◆ printing the result computed by a desk calculator
- ◆ Interacting with symbol table. That is, a code generator might enter the type of an identifier into a symbol table etc.

Now, let us see “How to control the side effects in SDD?” The side effects in SDD can be controlled in one of the following ways:

- ◆ Permitting side effects when attribute evaluation based on any topological sort of the dependency graph produces a correct translation
- ◆ Impose constraints in the evaluation order so that the same translation is produced for any allowable order

For example, consider the SDD given in example 5.1. This SDD do not have any side effects. Now, let us consider the first semantic rule and corresponding production shown below:

<u>Production</u>	<u>Semantic rule</u>
$S \rightarrow E_n$	$S.\text{val} = E.\text{val}$

Let us modify the semantic rule and introduce a side effect by printing the value of  $E.\text{val}$  as shown below:

<u>Production</u>	<u>Semantic rule</u>
$S \rightarrow E_n$	$\text{print}(E.\text{val})$

Now, the semantic rule “ $\text{print}(E.\text{val})$ ” is treated as dummy synthesized attribute associated with LHS (head) of the production. The modified SDD produces the same translation under any topological sort, since the print statement is executed at the end only after computing the value of  $E.\text{val}$ .

The complete SDD of desk calculator with side effects is shown below:

---

**Example 5.7:** Write the grammar and SDD for a simple desk calculator with side effect

---

**Solution:** The SDD for simple desk calculator along with side effect of printing the value of an attribute after evaluation can be written as shown below:

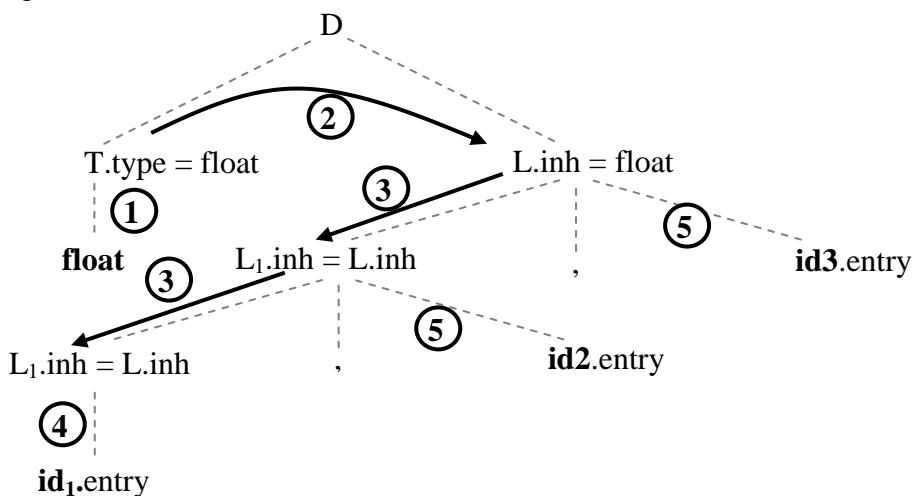
<u>Productions</u>	<u>Semantic Rules</u>
$S \rightarrow E_n$	$\text{print}(E.\text{val})$
$E \rightarrow E + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$
$E \rightarrow E - T$	$E.\text{val} = E_1.\text{val} - T.\text{val}$
$E \rightarrow T$	$E.\text{val} = T.\text{val}$
$T \rightarrow T * F$	$T.\text{val} = T_1.\text{val} * F.\text{val}$
$T \rightarrow T / F$	$T.\text{val} = T_1.\text{val} / F.\text{val}$
$T \rightarrow F$	$T.\text{val} = F.\text{val}$
$F \rightarrow (E)$	$F.\text{val} = E.\text{val}$
$F \rightarrow \text{digit}$	$F.\text{val} = \text{digit}.\text{lexval}$

**Example 5.8:** Write the SDD for a simple type declaration and write the annotated parse tree and the dependency graph for the declaration “**float a, b, c**”

**Solution:** The grammar for a simple type declaration can be written as shown below:

$$\begin{aligned} D &\rightarrow T L \\ T &\rightarrow \text{int} \mid \text{float} \\ L &\rightarrow L_1 , \text{id} \mid \text{id} \end{aligned}$$

Consider the parse tree for the declaration: “**float a, b, c**” where  $a$ ,  $b$  and  $c$  are identifiers represented by  $id1$ ,  $id2$  and  $id3$  respectively along with partial annotated parse tree showing the direction of evaluations:



## 5.26 □ Syntax Directed Translation

---

- ① The declaration D consists of a basic type T followed by a list L of identifiers. T can be either **int** or **float**. Thus, the tokens corresponding **int** and **float** such as *integer* and *float* obtained from the lexical analyzer are copied into attribute value of T. The production and its semantic rule can be written as shown below:

<u>Productions</u>	<u>Semantic rules</u>
T → int	T.type = <b>integer</b>
T → float	T.type = <b>float</b>

- ② Observe that the attribute value T.type available in the left subtree should be transferred to right subtree to L. Since attribute value is transferred from left sibling to right sibling its attribute must be inherited attribute and it is denoted by L.inh and can be obtained using the production D → T L as shown below:

<u>Productions</u>	<u>Semantic rules</u>
D → T L	L.inh = T.type

- ③ The type L.inh must be transferred to identifier **id** and hence it has to be copied into L<sub>1</sub>.inh which is the left most child in RHS of the production “L → L<sub>1</sub>, id”. This can be done as shown below:

<u>Productions</u>	<u>Semantic rules</u>
L → L <sub>1</sub> , id	L <sub>1</sub> .inh = L.inh

- ④ The attribute value of L.inh in turn must be entered as the type for identifier **id** using the production “L → id”. This can be done as shown below:

<u>Productions</u>	<u>Semantic rule with side effect</u>
L → id	Addtype(id.entry, L.inh)

where Addtype() is a function which accepts two parameters:

- id.entry is a lexical value that points to a symbol table
- L.inh is the type being assigned to every identifier in the list
- The function installs the type L.inh as the type of the corresponding identifier

- ⑤ The attribute value of L.inh in turn must be entered as the type for identifier **id** which is the right most child in RHS of the production “L → L<sub>1</sub>, id”. This can be done as shown below:

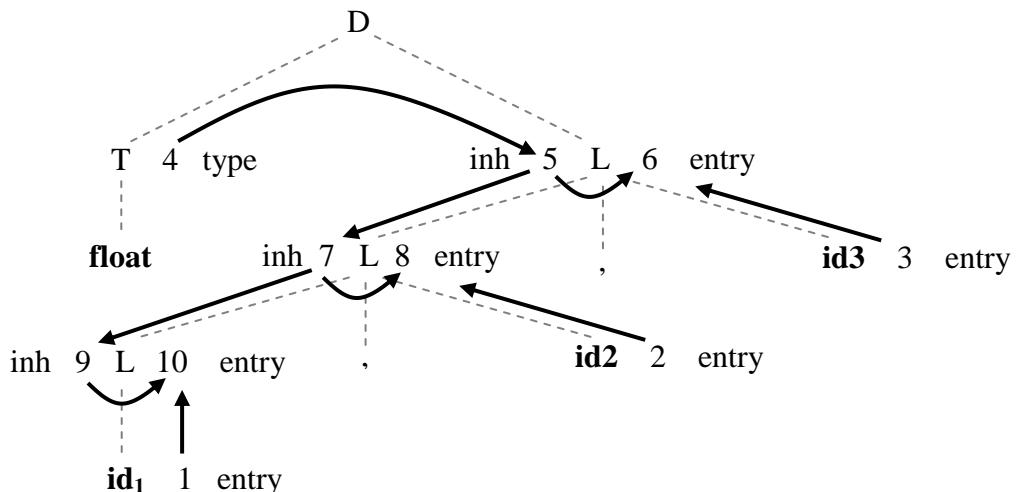
<u>Productions</u>	<u>Semantic rule with side effect</u>
L → L <sub>1</sub> , id	Addtype(id.entry, L.inh)

## ■ Introduction to Compiler Design - 5.27

Finally, the SDD for the grammar can be written by looking into dependency graph as shown below:

<u>Productions</u>	<u>Semantic Rules</u>
$D \rightarrow T L$	$L.inh = T.type$
$T \rightarrow \mathbf{int}$	$T.type = \text{integer}$
$T \rightarrow \mathbf{float}$	$T.type = \text{float}$
$L \rightarrow L_1, \mathbf{id}$	$L_1.inh = L.inh$ $\text{Addtype}(L.inh, \mathbf{id}.entry)$
$L \rightarrow \mathbf{id}$	$\text{Addtype}(L.inh, \mathbf{id}.entry)$

**Note:** Thus, we have an L-attributed definition with side effect of adding the type into symbol table for the corresponding identifier. The dependency graph for type declaration statement “**float a, b, c**” is shown below:



Observe the following points from above diagram:

- ♦ Numbers 1 through 10 represent the nodes of the dependency graph.
- ♦ Nodes 1, 2 and 3 represent attribute *entry* associated with each of the leaves labeled **id**.
- ♦ Node 4 represent the attribute *T.type* and is actually where attribute evaluation begins
- ♦ Nodes 5, 7 and 9 represent the attribute *L.inh* associated with each occurrence of the non-terminal *L*.
- ♦ Nodes 6, 8 and 10 represent the dummy attributes that represent the application of the function *addType()* to a type and one of these *entry* values

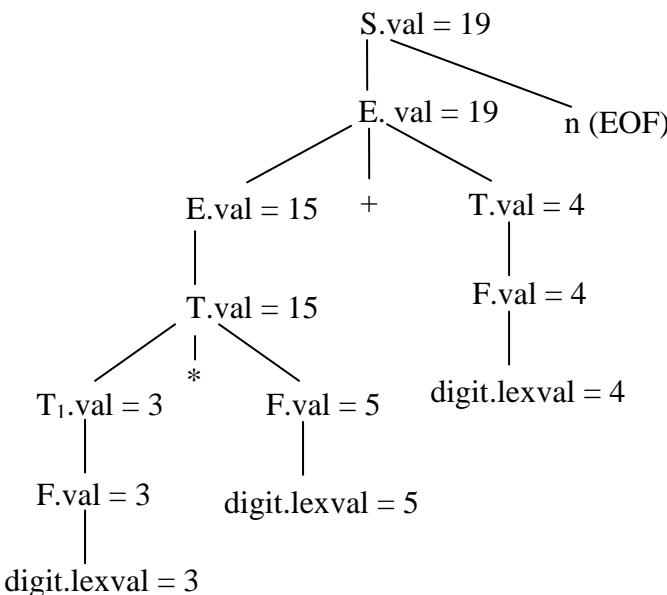
## 5.28 □ Syntax Directed Translation

**Example 5.9:** Write the SDD for a simple desk calculator. Write the annotated parse tree for the expression  $3*5+4n$

**Solution:** The SDD for a simple desk calculator is shown below: (See example 5.2 for detailed explanation)

<u>Productions</u>	<u>Semantic Rules</u>
$S \rightarrow E_n$	$S.val = E.val$
$E \rightarrow E + T$	$E.val = E_1.val + T.val$
$E \rightarrow E - T$	$E.val = E_1.val - T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T * F$	$T.val = T_1.val * F.val$
$T \rightarrow T / F$	$T.val = T_1.val / F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$

The annotated parse tree for the expression  $3*5 + 4n$  can be written as shown below:



## 5.6 Syntax directed translation

Now, let us see “What is syntax directed translation?”

**Definition:** The **Syntax Directed Translation** (in short SDT) is a context free grammar with embedded semantic actions. The semantic actions are nothing but the sequence of steps or program fragments that will be carried out when that production is used in the derivation. The SDTs are used:

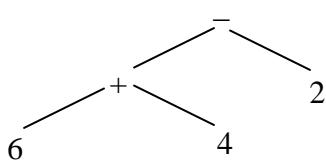
- ◆ To build syntax trees for programming constructs.
- ◆ To translate infix expressions into postfix notation
- ◆ To evaluate expressions

### 5.6.1 Construction of syntax trees

The main application of SDT in this section is the construction of syntax trees. Since some of the compilers use syntax trees as an intermediate representation, an SDD accepts the input string and produce a syntax tree. The syntax tree is also called *abstract syntax tree*. The parse tree is also called *concrete syntax tree*.

Before proceeding further, let us see “What is a syntax tree? What is the difference between syntax tree and parse tree?”

**Definition:** A *syntax tree* also called *abstract syntax tree* is a compressed form of parse tree which is used to represent language constructs. In a syntax tree for an expression, each interior node represents an operator and the children of the node represent the operands of the operator. In general, any programming construct can be handled by making up an operator for the construct and treat semantically meaningful components of that construct as operands. **For example**, the syntax tree for the expression “ $6 + 4 - 2$ ” is shown below:



In the syntax tree observe the following points:

- ◆ The operator – represent the root node.
- ◆ The sub-trees of the root represent the sub-expressions “ $6+4$ ” and  $2$
- ◆ Similarly, for the expression,  $+$  is the root and  $6$  and  $4$  are the operands representing the children

It is easy to construct syntax trees with the help of SDD’s. The syntax trees resemble the parse trees to some extent but with following changes:

**Note 1:** In a syntax tree, all the operators and keywords appear as interior nodes.

**Note 2:** In a parse tree all operators and keywords appear as leaf nodes.

## 5.30 □ Syntax Directed Translation

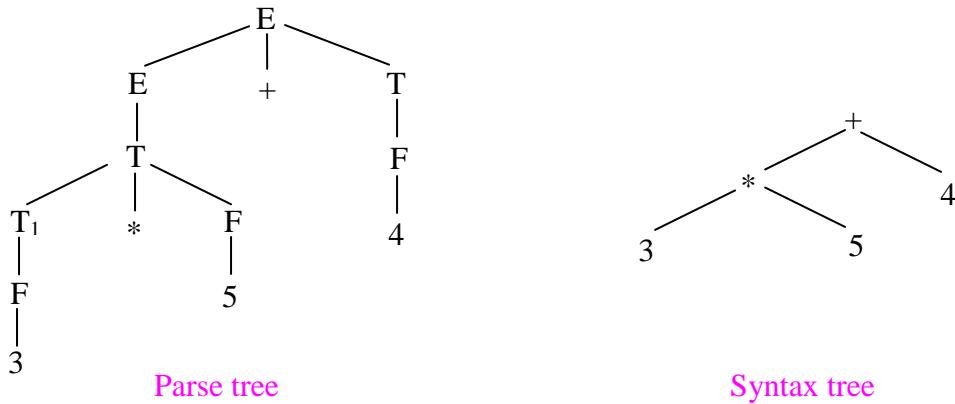
---

**Example 5.10:** For the following grammar show the parse tree and syntax tree for the expression  $3 * 5 + 4$ :

$$\begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T / F \mid F \\ F &\rightarrow (E) \mid \text{digit} \mid \text{id} \end{aligned}$$


---

**Solution:** The annotated parse tree for the expression  $3 * 5 + 4$  is shown in example 5.9. Now, the parse tree and syntax tree for the expression  $3 * 5 + 4$  are shown below:

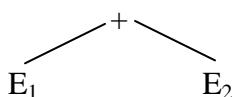


Now, let us see “How to construct syntax trees?” The syntax tree for expressions can be constructed using two SDD’s namely:

- ♦ S-attributed definition which is used for bottom-up parsing
- ♦ L-attributed definition which is used for top-down parsing

### 5.6.1.1 Syntax tree for S-attributed definition

The main application of SDT in this section is the construction of syntax trees. Now, let us see “How to construct semantic rules that help us to create syntax trees for the expressions?” Each node in a syntax tree represents a programming construct and the children of the node represent meaningful components of that construct. A syntax-tree node representing an expression  $E_1 + E_2$  has label  $+$  and two children representing the sub-expressions  $E_1$  and  $E_2$  as shown below:



The nodes of a syntax tree can be implemented by creating objects where each object containing two or more fields. The two functions that are useful in constructing the syntax trees are shown below:

## Introduction to Compiler Design - 5.31

- ♦ **Leaf(*op, val*) :** This function is called only for the terminals and it is used to create only leaf nodes containing two fields namely:
  - *op* field holds the label for the node
  - *val* field holds the lexical value obtained from the lexical analyzer
- ♦ **Node(*op, c<sub>1</sub>, c<sub>2</sub>, c<sub>3</sub>,...,c<sub>n</sub>*) :** This function is called to create only interior nodes with various fields namely:
  - *op* field holds the label for the node
  - *c<sub>1</sub>, c<sub>2</sub>, c<sub>3</sub>, ...c<sub>n</sub>* refer (or pointers) to children for the node labeled *op*.

---

**Example 5.11:** Obtain the semantic rules to construct a syntax tree for simple arithmetic expressions using bottom up approach

---

**Solution:** The grammar to generate an arithmetic expression is shown below:

$$\begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T / F \mid F \\ F &\rightarrow (E) \mid \text{digit} \mid \text{id} \end{aligned}$$

It is easy to construct syntax trees with the help of SDD's (For detail of SDD of arithmetic expression see example 5.2) as shown below:

- ♦ For the production of the form  $E \rightarrow E_1 + E_2$ , we have to use a rule that creates a node with '+' for *op* fields two children  $E_1.\text{node}$  and  $E_2.\text{node}$  that represent sub-expressions. This can be done by creating a node with **new** operator using function **Node()** as shown below:

$$\begin{array}{ccc} E & \rightarrow & \underbrace{E_1 + E_2}_{\downarrow} \\ & & \text{E.node} = \text{new Node}('+' , E_1.\text{node}, E_2.\text{node}) \end{array}$$

On similar lines, we can write the semantic rules for other productions consisting of the operators +, -, \* and / as shown below:

<u>Production</u>	<u>Semantic rules</u>
$E \rightarrow E_1 + T$	$E.\text{node} = \text{new Node}('+' , E_1.\text{node}, E_2.\text{node})$
$E \rightarrow E_1 - T$	$E.\text{node} = \text{new Node}('-' , E_1.\text{node}, T.\text{node})$
$T \rightarrow T_1 * F$	$T.\text{node} = \text{new Node}('*' , T_1.\text{node}, F.\text{node})$
$T \rightarrow T_1 / F$	$T.\text{node} = \text{new Node}('/' , T_1.\text{node}, F.\text{node})$

## 5.32 □ Syntax Directed Translation

---

- ♦ For the production of the form  $E \rightarrow T$  and  $E \rightarrow ( T )$ , no node is created, since  $E.\text{node}$  is the same as that of  $T.\text{node}$ . So, semantic rules for the productions of the above form can be written as shown below:

<u>Production</u>	<u>Semantic rules</u>
$E \rightarrow T$	$E.\text{node} = T.\text{node}$
$T \rightarrow F$	$T.\text{node} = F.\text{node}$
$F \rightarrow ( E )$	$F.\text{node} = E.\text{node}$

- ♦ For the production of the form  $A \rightarrow a$  where  $a$  is a terminal, use the function  $\text{Leaf}()$  with  $a$  and  $a.\text{entry}$  as the parameter if  $a$  is an identifier or  $a.\text{val}$  as the parameter if  $a$  is a number of digit. So, semantic rules for the productions of the above form can be written as shown below:

<u>Production</u>	<u>Semantic rules</u>
$F \rightarrow \text{digit}$	$F.\text{node} = \text{new Leaf}(\text{digit}, \text{digit.val})$
$F \rightarrow \text{id}$	$F.\text{node} = \text{new Leaf}(\text{id}, \text{id.entry})$

So, the final set of SDDs used to construct syntax trees for simple expressions can be written as shown below:

<u>Production</u>	<u>Semantic rules</u>
$E \rightarrow E_1 + T$	$E.\text{node} = \text{new Node}( '+ ', E_1.\text{node}, T.\text{node} )$
$E \rightarrow E_1 - T$	$E.\text{node} = \text{new Node}( ' - ', E_1.\text{node}, T.\text{node} )$
$E \rightarrow T$	$E.\text{node} = T.\text{node}$
$T \rightarrow T_1 * F$	$T.\text{node} = \text{new Node}( ' * ', T_1.\text{node}, F.\text{node} )$
$T \rightarrow T_1 / F$	$T.\text{node} = \text{new Node}( ' / ', T_1.\text{node}, F.\text{node} )$
$T \rightarrow F$	$T.\text{node} = F.\text{node}$
$F \rightarrow ( E )$	$F.\text{node} = E.\text{node}$
$F \rightarrow \text{digit}$	$F.\text{node} = \text{new Leaf}(\text{digit}, \text{digit.val})$
$F \rightarrow \text{id}$	$F.\text{node} = \text{new Leaf}(\text{id}, \text{id.entry})$

**Note:** The above semantic rules show a left-recursive grammar that is S-attributed (so all attributes are synthesized). Now, let us see how to create a syntax tree using above semantic rules.

---

**Example 5.12:** Create a syntax tree for the expression “ $a - 4 + c$ ”

---

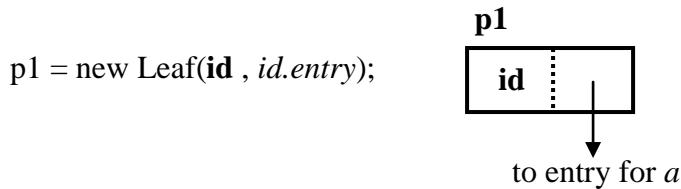
## ■ Introduction to Compiler Design - 5.33

**Solution:** Consider for the arithmetic expression “**a - 4 + c**”. Here, we make use of the following functions to create the nodes of syntax trees for expressions with binary operators.

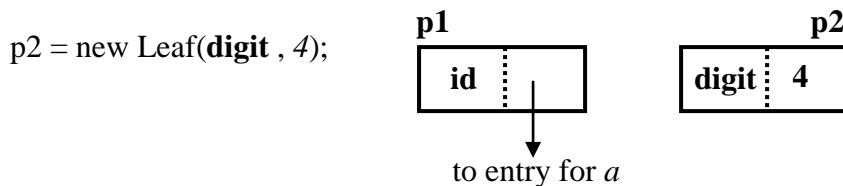
- ♦ **Node(*op* , *left* , *right*)** creates an operator node with label *op* and two fields containing pointers to *left* and *right* sub-tree
- ♦ **Leaf(**id** , *entry*)** creates a identifier node with label **id** and a field containing *entry*, which pointer to the symbol-table entry for the identifier.
- ♦ **Leaf(**digit** , *val*)** creates a number node with label **digit** and a field containing *val* which represent the value of the number.

The following sequence of function calls creates the syntax tree for the arithmetic expression **a - 4 + c** as shown below:

**Step 1:** Create a leaf node for identifier **a** using the function Leaf() as shown below:

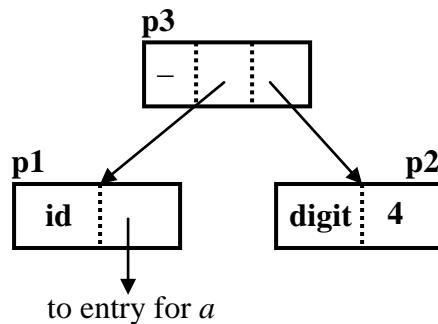


**Step 2:** Create a leaf node for digit 4 using the function Leaf() as shown below:



**Step 3:** Create an interior node by passing ‘-‘ as the first argument and pointers to the first two leaves p1 and p2 with the help of function Node() as shown below:

p3 = new Node(‘-‘ , p1, p2);

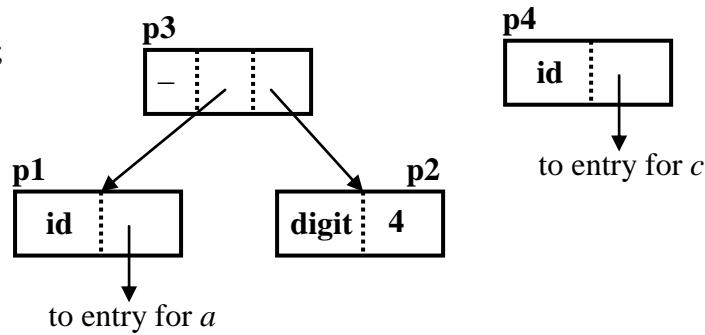


### 5.34 □ Syntax Directed Translation

---

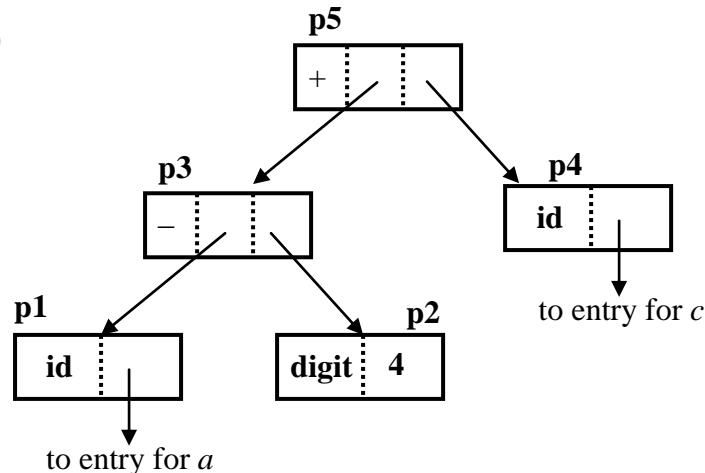
**Step 4:** Create a leaf node for identifier  $c$  using the function Leaf() as shown below:

`p4 = new Leaf(id, id.entry);`



**Step 5:** Create an interior node by passing '+' as the first argument and pointers to the left sub-tree identified by  $p3$  and pointer to the right sub-tree identified by  $p4$  with the help of function Node() as shown below:

`p5 = new Node('+', p3, p4)`



Thus, to get the above syntax tree for the expression “ $a - 4 + c$ ”, the various steps that are used are shown below:

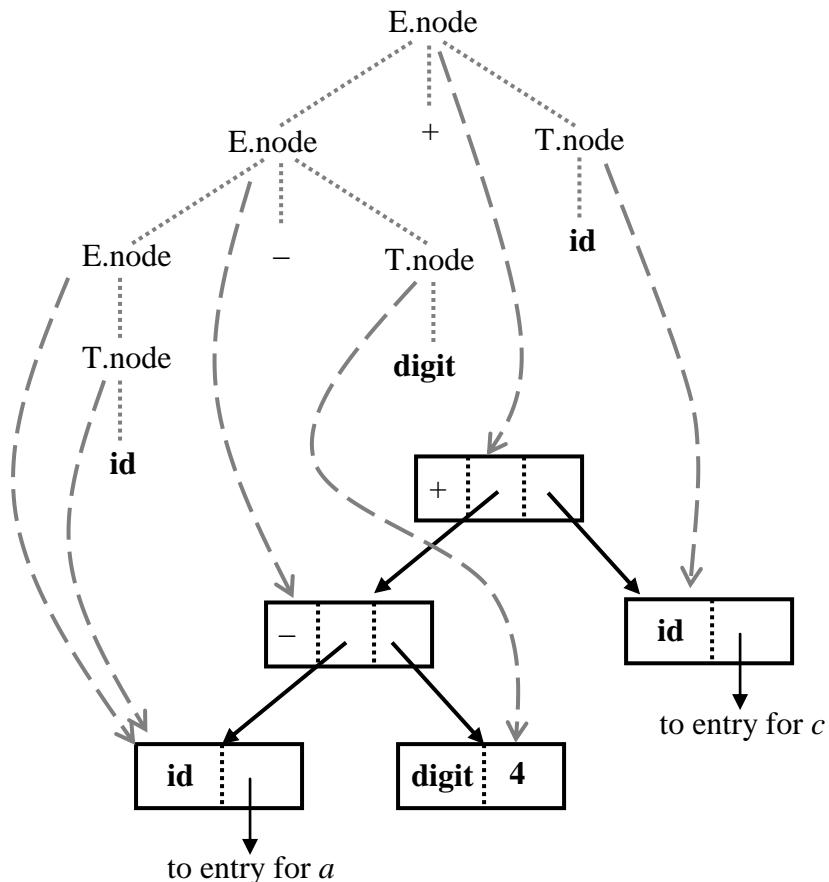
```

p1 = new Leaf(id , id.entry);
p2 = new Leaf(digit , 4);
p3 = new Node(‘-‘ , p1, p2);
p4 = new Leaf(id, id.entry);
p5 = new Node(‘+’ , p3, p4)
  
```

## ■ Introduction to Compiler Design - 5.35

**Note:** If the above rules are evaluated during a postorder traversal of the parse tree or with reductions during bottom-up parse, then above sequence of steps ends with p5 pointing to the root of the constructed parse tree.

The parse tree, annotated parse tree depicting the construction of a syntax tree for the arithmetic expression “a - 4 + c” is shown below:



Observe the following points:

- ♦ The nodes of the syntax tree are shown as records with *op* as the first field
- ♦ Syntax tree edges are shown using solid lines
- ♦ The parse tree which need not actually be constructed is shown with dotted edges.
- ♦ The third type of line, shown dashed represents the values of E.node and T.node where each line points to the appropriate syntax-tree node
- ♦ At the bottom we see leaves for *a*, 4 and *c* constructed with the help of function Leaf()
- ♦ The interior nodes for the operators are created with the help of function Node()

## 5.36 □ Syntax Directed Translation

---

### 5.6.1.2 Syntax tree for L-attributed definition

The method of constructing syntax tree for L-attributed definition remains same as the method of constructing syntax tree for S-attributed definition. The functions Leaf() and Node() are used with same number and type of parameters.

---

**Example 5.13:** Obtain the semantic rules to construct a syntax tree for simple arithmetic expressions using top-down approach with operators + and –

---

**Solution:** The grammar to generate an arithmetic expression with two operators + and – is shown below:

$$\begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow (E) \mid \text{digit} \mid \text{id} \end{aligned}$$

The above grammar is not suitable for top-down parser since it has left recursion. After eliminating left recursion, we get the following grammar:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid -TE' \mid \epsilon \\ T &\rightarrow (E) \mid \text{digit} \mid \text{id} \end{aligned}$$

To construct syntax tree, first obtain the SDD for the above grammar. The SDD for the above grammar is shown below (For details refer example 5.4)

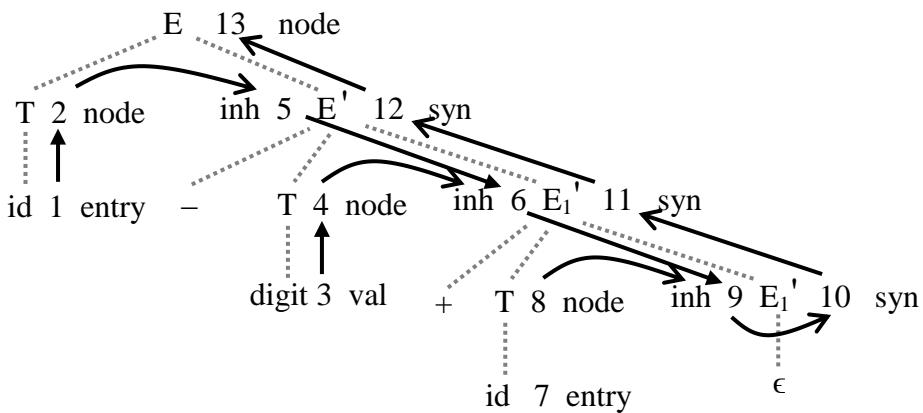
<u>Production</u>	<u>Semantic Rule</u>
$E \rightarrow TE'$	$E'.inh = T.val$ $E.val = E'.syn$
$E' \rightarrow +TE_1'$	$E_1'.inh = E'.inh + T.val$ $E'.syn = E_1'.syn$
$E' \rightarrow -TE_1'$	$E_1'.inh = E'.inh - T.val$ $E'.syn = E_1'.syn$
$E' \rightarrow \epsilon$	$E'.syn = E_1'.inh$
$T \rightarrow (E)$	$T.val = E.val$
$T \rightarrow \text{digit}$	$T.val = \text{digit.lexval}$
$T \rightarrow \text{id}$	$T.val = \text{id.entry}$

## ■ Introduction to Compiler Design - 5.37

The functions Leaf() and Node() functions are used to create the syntax tree. Now, constructing syntax trees during top-down parsing can be obtained using the following semantic rules:

<u>Production</u>	<u>Semantic Rule</u>
$E \rightarrow T E'$	$E'.inh = T.node$ $E.node = E'.syn$
$E' \rightarrow + T E_1'$	$E_1'.inh = \text{new Node}( '+', E'.inh, T.node)$ $E'.syn = E_1'.syn$
$E' \rightarrow - T E_1'$	$E_1'.inh = \text{new Node}( ' - ', E'.inh, T.node)$ $E'.syn = E_1'.syn$
$E' \rightarrow \epsilon$	$E'.syn = E_1'.inh$
$T \rightarrow ( E )$	$T.node = E.node$
$T \rightarrow \text{digit}$	$T.node = \text{new Leaf}(\text{digit}, \text{digit.lexval})$
$T \rightarrow \text{id}$	$T.node = \text{new Leaf}(\text{id}, \text{id.entry})$

The annotated parse tree along with dependency graph for the expression “a – 4 + c” can be written as shown below:




---

**Example 5.14:** Obtain the semantic rules to construct a syntax tree for simple arithmetic expressions with operators -, +, \*, / using top-down approach

---

## 5.38 □ Syntax Directed Translation

---

**Solution:** The grammar to generate an arithmetic expression is shown below:

$$\begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T / F \mid F \\ F &\rightarrow (E) \mid \text{digit} \mid \text{id} \end{aligned}$$

The above grammar is not suitable for top-down parser since it has left recursion. After eliminating left recursion, we get the following grammar:

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid - T E' \mid \epsilon \\ T &\rightarrow F T' \\ T' &\rightarrow * F T' \mid / F T' \mid \epsilon \\ F &\rightarrow (E) \mid \text{digit} \mid \text{id} \end{aligned}$$

To construct syntax tree, first obtain the SDD for the above grammar. The SDD for the above grammar is shown below (For details refer example 5.4)

<u>Production</u>	<u>Semantic Rule</u>
$E \rightarrow T E'$	$E'.\text{inh} = T.\text{val}$ $E.\text{val} = E'.\text{syn}$
$E' \rightarrow + T E'$	$E_1'.\text{inh} = E'.\text{inh} + T.\text{val}$ $E'.\text{syn} = E_1'.\text{syn}$
$E' \rightarrow - T E'$	$E_1'.\text{inh} = E'.\text{inh} - T.\text{val}$ $E'.\text{syn} = E_1'.\text{syn}$
$E' \rightarrow \epsilon$	$E'.\text{syn} = E_1'.\text{inh}$
$T \rightarrow F T'$	$T'.\text{inh} = F.\text{val}$ $T.\text{val} = T'.\text{syn}$
$T' \rightarrow * F T'$	$T_1'.\text{inh} = T'.\text{inh} * F.\text{val}$ $T'.\text{syn} = T_1'.\text{syn}$
$T' \rightarrow / F T'$	$T_1'.\text{inh} = T'.\text{inh} / F.\text{val}$ $T'.\text{syn} = T_1'.\text{syn}$

## ■ Introduction to Compiler Design - 5.39

---

$T' \rightarrow \epsilon$	$T'.syn = T_1'.inh$
$F \rightarrow ( E )$	$F.val = E.val$
$F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$
$F \rightarrow \text{id}$	$F.val = \text{id.entry}$

The same functions Leaf() and Node() functions are used to create the syntax tree. Now, constructing syntax trees during top-down parsing can be obtained using the following semantic rules:

<u>Production</u>	<u>Semantic Rule</u>
$E \rightarrow T E'$	$E'.inh = T.\text{node}$ $E.\text{node} = E'.syn$
$E' \rightarrow + T E_1'$	$E_1'.inh = \text{new Node}( '+ ', E'.inh, T.\text{node})$ $E'.syn = E_1'.syn$
$E' \rightarrow - T E_1'$	$E_1'.inh = \text{new Node}( '- ', E'.inh, T.\text{node})$ $E'.syn = E_1'.syn$
$E' \rightarrow \epsilon$	$E'.syn = E_1'.inh$
$T \rightarrow F T'$	$T'.inh = F.\text{node}$ $T.\text{node} = T'.syn$
$T' \rightarrow * F T_1'$	$T_1'.inh = \text{new Node}( '*', T'.inh, F.\text{node})$ $T'.syn = T_1'.syn$
$T' \rightarrow / F T_1'$	$T_1'.inh = \text{new Node}( '/', T'.inh, F.\text{node})$ $T'.syn = T_1'.syn$
$T' \rightarrow \epsilon$	$T'.syn = T_1'.inh$
$F \rightarrow ( E )$	$F.\text{node} = E.\text{node}$
$F \rightarrow \text{digit}$	$F.\text{node} = \text{new Leaf}(\text{digit}, \text{digit.lexval})$
$F \rightarrow \text{id}$	$F.\text{node} = \text{new Leaf}(\text{id}, \text{id.entry})$

## 5.40 □ Syntax Directed Translation

### 5.6.2 The structure of a type

Now, let us see “What is the use of inherited attributes?” During top-down parsing, the grammar should not have left recursion. If the grammar has left recursion, we have to eliminate left recursion. The resulting grammar needs inherited attributes. Inherited attributes are useful when the structure of the parse tree differs from the syntax tree for the specified input. The attributes can then be used to carry information from one part of the parse tree to another part of the parse tree. But sometimes, even though the grammar does not have left recursion, the language itself demands inherited attributes. This can be explained by considering array type as shown below:

---

**Example 5.15:** “Give the syntax directed translation of type **int [2][3]** and also given the semantic rules for the respective productions”

---

**Solution:** The grammar for multi-dimensional array can be written as shown below:

$$\begin{aligned} T &\rightarrow BC \\ B &\rightarrow \mathbf{int} \\ B &\rightarrow \mathbf{float} \\ C &\rightarrow [\text{num}] C_1 \\ C &\rightarrow \epsilon \end{aligned}$$

Observe the following points from above grammar:

- ♦ The non-terminal T generates either a basic type such as **int** or **float**. This is achieved using the following derivation:

$$T \Rightarrow BC \Rightarrow \mathbf{int} C \Rightarrow \mathbf{int} \quad \text{[Finally replacing } C \text{ with } \epsilon \text{]}$$

or

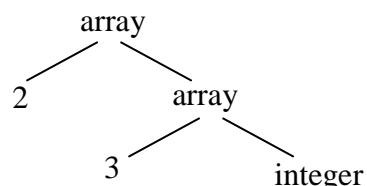
$$T \Rightarrow BC \Rightarrow \mathbf{float} C \Rightarrow \mathbf{float} \quad \text{[Finally replacing } C \text{ with } \epsilon \text{]}$$

- ♦ The non-terminal T also generates array of components consisting of a sequence of integers where each integer is surrounded by brackets as shown below:

$$T \Rightarrow BC \Rightarrow \mathbf{int} C \Rightarrow \mathbf{int} [\text{num}] C \Rightarrow \mathbf{int} [\text{num}] [\text{num}] C \Rightarrow \mathbf{int} [\text{num}] [\text{num}] \quad \Rightarrow \mathbf{int} [\text{num}] [\text{num}]$$

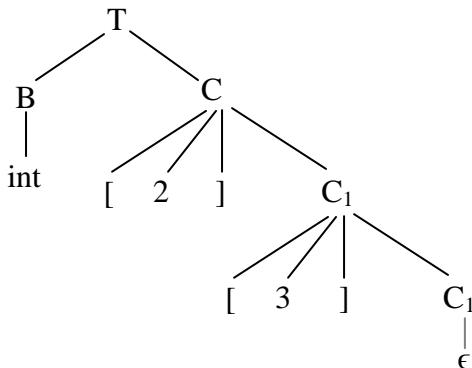
Finally replacing C with  $\epsilon$

In C, **int [2][3]** is interpreted as “Array of 2 arrays of 3 integers” which can be denoted by `array(2, array(3, integer))` can be written in the form of a tree as shown in the figure:



## ■ Introduction to Compiler Design - 5.41

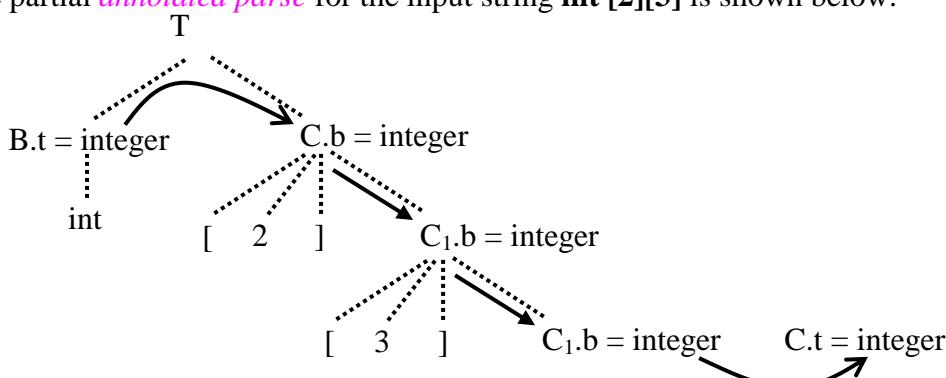
The parse tree to get the string **int [2][3]** is shown below:



It is very easy for us to write SDD, if we have annotated parse tree along with directions that gives the flow. So, let us use the following attribute names for the non-terminals:

- ♦ For the synthesized attribute, we use attribute name as  $t$
- ♦ For the inherited attribute, we use attribute name as  $b$

The partial *annotated parse* for the input string **int [2][3]** is shown below:



Observe the following points from above partial annotated parse tree:

- ♦ The type **int** is moved to parent B and attribute  $t$  is synthesized. The production and equivalent semantic rule is shown below:

Production

$$\begin{aligned} B &\rightarrow \text{int} \\ B &\rightarrow \text{float} \end{aligned}$$

Semantic rule

$$\begin{aligned} B.t &= \text{integer} \\ B.t &= \text{float} \end{aligned}$$

- ♦ The type *integer* has to be transferred from  $B.t$  on the left to  $C$  on the right using the production  $T \rightarrow B C$  as shown using the arrow mark and hence it is inherited attribute denoted by  $b$ . The production and equivalent semantic rule is shown below:

Production

$$T \rightarrow B C$$

Semantic rule

$$C.b = B.t$$

- ♦ Now, the attribute value of  $C.b$  moves down and copied into  $C_1$  using the production  $C \rightarrow [\text{num}] C_1$ . It must be inherited. The equivalent semantic rule is shown below:

Production

$$C \rightarrow [\text{num}] C_1$$

Semantic rule

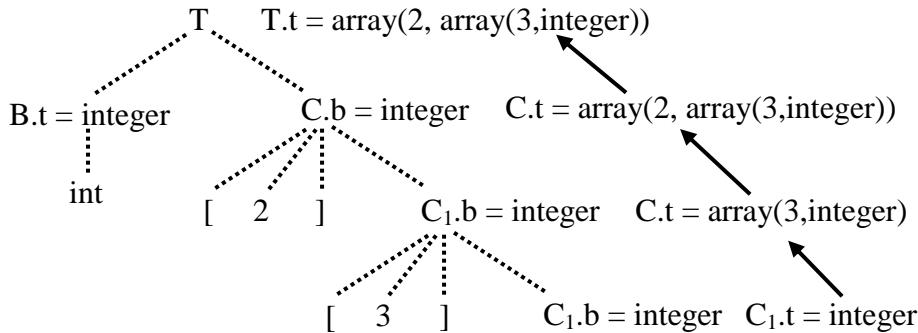
$$C_1.b = C.b$$

## 5.42 □ Syntax Directed Translation

- ♦ Because of the production  $C \rightarrow \epsilon$ , the right most  $C$  in the above tree, takes its synthesized value  $C.t$  from itself using inherited attribute value  $C.b$ . The semantic rule is shown below:

<u>Production</u>	<u>Semantic rule</u>
$C \rightarrow \epsilon$	$C.t = C.b$

Now, the synthesized attribute value *integer* has to be moved from right most non-terminal to the root node as shown below in partial annotated parse tree:



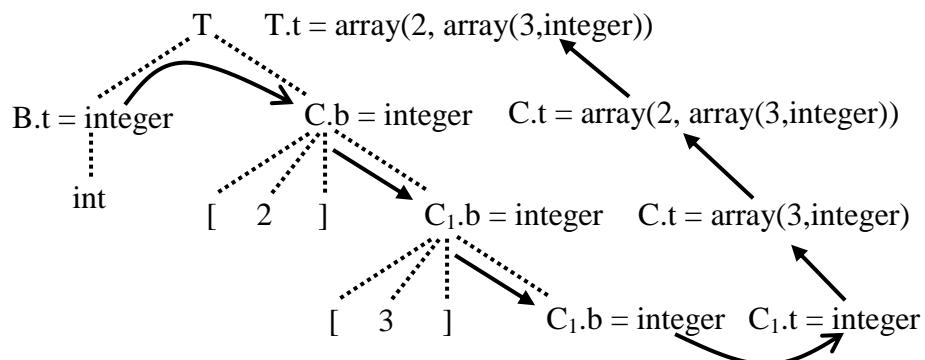
- ♦ The attribute value  $C_1.t$  is moved to  $C.t$  upwards using the production  $C \rightarrow [num] C_1$ . The semantic rule can be written as shown below:

<u>Production</u>	<u>Semantic rule</u>
$C \rightarrow [num] C_1$	$C.t = array(num.val, C_1.t)$

- ♦ Finally, attribute value of  $C.t$  is moved to  $T.t$  using the production  $T \rightarrow B\ C$  and equivalent semantic rule is shown below:

<u>Production</u>	<u>Semantic rule</u>
$T \rightarrow B\ C$	$T.t = C.t$

So, the final annotated parse tree is shown below:



## ■ Introduction to Compiler Design - 5.43

The final semantic rules are shown below:

<u>Production</u>	<u>Semantic rule</u>	<u>Type</u>
T → B C	C.b = B.t	Inherited
	T.t = C.t	Synthesized
B → int	B.t = integer	Synthesized
B → float	B.t = float	Synthesized
C → [num] C <sub>1</sub>	C <sub>1</sub> .b = C.b	Inherited
	C.t = array(num.val, C <sub>1</sub> .t)	Synthesized
C → ε	C.t = C.b	Synthesized

## 5.7 Syntax directed translation schemes

Now, let us see “What is syntax directed translation scheme?”

**Definition:** A syntax-directed translation scheme is a context free grammar with program fragments embedded within production bodies. The program fragments are called semantic actions and can appear at any position within the production body. By convention, semantic actions are enclosed within braces. If braces are used as grammar symbols in the production body then we quote them.

Note the following points:

- ♦ Any SDT can be implemented by first building a parse tree and then performing the actions in a pre-order manner.
- ♦ But, typically, SDT's are implemented during parsing, without building a parse tree
- ♦ The SDT's are used to implement two important classes of SDD's namely:
  - 1) Underlying grammar is LR-parsable and the SDD is S-attributed
  - 2) Underlying grammar is LL-parsable and the SDD is L-attributed
- ♦ The semantic rules in an SDD can be converted into an SDT with actions that are executed at the right time. During parsing, an action in a production body is executed as soon as all the grammar symbols to the left of the action have been matched.

### 5.7.1 Postfix translation schemes

In this section, let us see “What is Postfix Syntax-directed-translation or Postfix SDT?”

**Definition:** In an SDD implementation, we parse the grammar bottom-up and the SDD is S-attributed. An SDT is constructed such that the actions to be executed are placed at the

## 5.44 □ Syntax Directed Translation

---

end of the production and are executed only when the right hand side of the production is reduced to left hand side of the production i.e., reduction of the body to the head of the production. The SDT's with all actions at the right end of the production bodies are called *postfix SDT's* or *postfix syntax-directed-translations*.

---

**Example 5.16:** Obtain Postfix SDT implementation of the desk calculator to evaluate the given expression

---

SDT can be easily obtained by looking at the SDD shown in example 5.7. Observe that all the semantic rules in SDD enclosed within braces results in SDT. The postfix SDT implementation of the desk calculator is shown below:

<u>Productions</u>	<u>Actions</u>
$S \rightarrow E \ n$	{ print(E.val) }
$E \rightarrow E_1 + T$	{ E.val = E_1.val + T.val }
$E \rightarrow T$	{ E.val = T.val }
$T \rightarrow T_1 * F$	{ T.val = T_1.val * F.val }
$T \rightarrow F$	{ T.val = F.val }
$F \rightarrow ( E )$	{ F.val = E.val }
$F \rightarrow \text{digit}$	{ F.val = digit.lexval }

In above SDT observe that actions enclosed within braces are present at the end of each production.

**Note:** In a typical SDT, actions can be placed either at the beginning of the production or at the end of the production or somewhere in between. Now, instead of evaluating the arithmetic expression, let us convert the infix expression to postfix expression

---

**Example 5.17:** Write the SDD and annotate parse tree for converting an infix to postfix expression

---

The grammar to generate un-parenthesized arithmetic expression expressed using infix notation consisting of the operators + and \* is shown below:

$$\begin{aligned} E &\rightarrow E_1 + T \\ E &\rightarrow T \\ T &\rightarrow T_1 * F \\ T &\rightarrow F \\ F &\rightarrow 0 \\ F &\rightarrow 1 \\ &\dots \\ &\dots \\ F &\rightarrow 9 \end{aligned}$$

## ■ Introduction to Compiler Design - 5.45

**Note:** Infix expression:  $a + b$

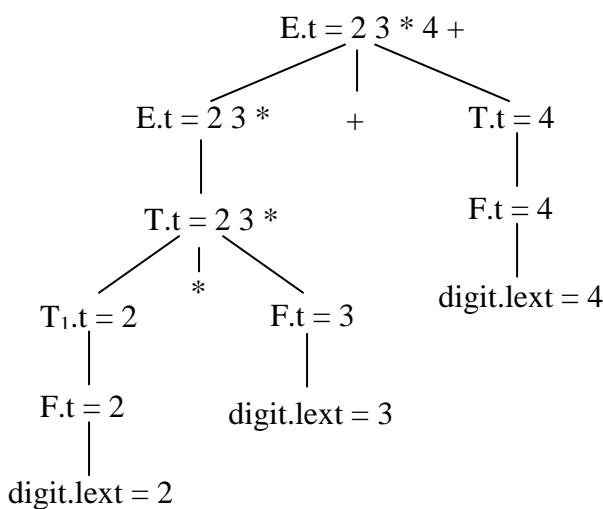
Postfix expression:  $a\ b\ +$

Instead of writing the postfix expression as:  $ab\ +$  we write as:  $a\ ||\ b\ ||\ '+'$  where the symbol  $\parallel$  is treated as concatenation operator.

So, SDD for infix to postfix translation is written as shown below:

<u>Production</u>	<u>Semantic rules (Postfix notation)</u>
$E \rightarrow E_1 + T$	$E.t = E_1.t \parallel T.t \parallel '+'$
$E \rightarrow T$	$E.t = T.t$
$T \rightarrow T_1 * F$	$T.t = T_1.t \parallel F.t \parallel '*'$
$T \rightarrow F$	$T.t = F.t$
$F \rightarrow 0$	$F.t = '0'$
$F \rightarrow 1$	$F.t = '1'$
.....	
.....	
$F \rightarrow 9$	$F.t = '9'$

**Note:** The semantic rule for the production  $E \rightarrow E_1 + T$  is  $E.t = E_1.t + T.t$ . But, in the postfix notation, it can be written as:  $E.t = E_1.t \parallel T.t \parallel '+'$  where the symbol  $\parallel$  is used as concatenation operator. The annotated parse tree for converting the expression  $2*3+4$  can be written by writing the parse tree for the given expression as shown below:



## 5.46 □ Syntax Directed Translation

**Example 5.18:** Write the SDT for converting an infix to postfix expression. Show the actions for translating the expression  $2 * 3 + 4$  into its equivalent postfix expression

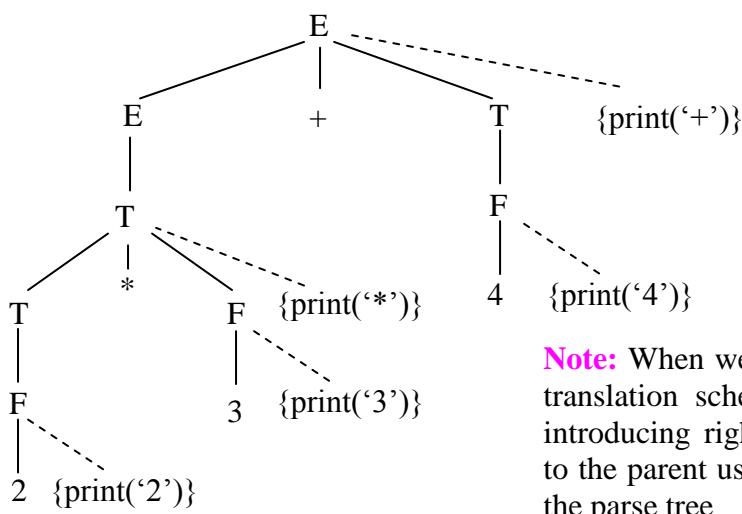
The grammar to generate un-parenthesized arithmetic expression expressed using infix notation consisting of the operators + and \* is shown below:

$$\begin{aligned} E &\rightarrow E_1 + T \\ E &\rightarrow T \\ T &\rightarrow T_1 * F \\ T &\rightarrow F \\ F &\rightarrow 0 \\ F &\rightarrow 1 \\ &\dots \\ &\dots \\ F &\rightarrow 9 \end{aligned}$$

To convert an infix expression to postfix expression, write the parse tree to get the expression  $2 * 3 + 4$  and print the operator or operand using the following rules:

- ♦ If only child is present (representing an operand), introduce one right sibling printing the operand
- ♦ If any one of the children contains an operator, introduce right most sibling to print the operator
- ♦ The rightmost sibling introduced is shown in dotted arrow for convenience

The parse tree to get the expression “ $2 * 3 + 4$ ” is shown below:



**Note:** When we draw the parse tree for the translation scheme, indicate an action by introducing rightmost child and connect it to the parent using dashed line as shown in the parse tree

Now, traversing in postorder and executing only the actions we get the postfix expression:

2 3 \* 4 +

## Introduction to Compiler Design - 5.47

Now, the SDT for converting an infix expression to postfix expression can be easily written using the above parse tree and is shown below:

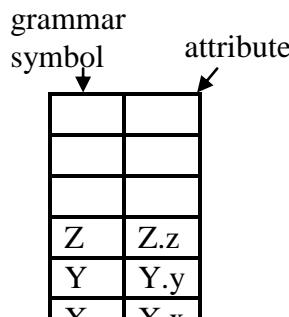
### Actions present at the end of productions (Postfix)

$E \rightarrow E_1 + T$	{ print('+' ) }
$E \rightarrow T$	
$T \rightarrow T_1 * F$	{ print('*') }
$T \rightarrow F$	
$F \rightarrow 0$	{ print('0') }
$F \rightarrow 1$	{ print('1') }
.....	
.....	
$F \rightarrow 9$	{ print('9') }

### **5.7.2 Parser-Stack Implementation of Postfix SDT's**

Now, let us “Explain parser-stack implementation of Postfix SDT’s?” Postfix SDT’s can be implemented during LR parsing by executing the actions whenever reduction occurs. This can be explained as shown below:

- ♦ The grammar symbols to be reduced to LHS of the production are present on top of the stack. For example, for the production of the form:  $A \rightarrow X Y Z$  and assume stack contains X, Y and Z and they are the symbols to be reduced
- ♦ Apart from placing grammar symbols on the top of the stack, place the attribute values of grammar symbols also on the top of the stack
- ♦ A grammar symbol may have more than one attribute and hence all the attributes associated with a grammar symbol should be passed on to the stack
- ♦ So, the stack should be implemented as an array of records where each record on the stack has the grammar symbol and its associated attributes. For the sake of convenience only one attribute of grammar symbol is shown in figure.
- ♦ If the attributes are synthesized and actions are present at the end of the production, then we can compute the attribute value of LHS of the production using the attribute values of RHS of the production. For example,



Stack

## 5.48 □ Syntax Directed Translation

If we reduce by a production such as  $A \rightarrow X Y Z$ , then we have all the attributes of X, Y and Z on the top of the stack. Remove those grammar symbols from the stack and push the grammar symbol A on to the stack along with its attribute.

**Example 5.18:** Write the actions of desk calculator SDT so that they manipulate the parser explicitly

**Solution:** The SDT for desk calculator can be written as shown below: (See example 5.16 for details)

$S \rightarrow E n$	{ print(E.val) }
$E \rightarrow E_1 + T$	{ E.val = E <sub>1</sub> .val + T.val }
$E \rightarrow T$	{ E.val = T.val }
$T \rightarrow T_1 * F$	{ T.val = T <sub>1</sub> .val * F.val }
$T \rightarrow F$	{ T.val = F.val }
$F \rightarrow ( E )$	{ F.val = E.val }
$F \rightarrow \text{digit}$	{ F.val = digit.lexval }

The desk calculator can be implemented using the stack as shown below:

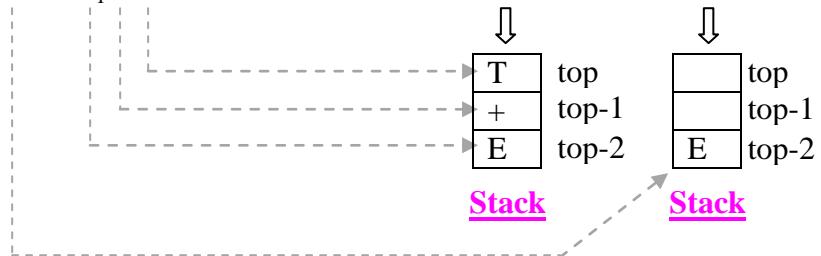
- ♦ Consider the production  $S \rightarrow E n$  and its stack contents



Using the above SDD, the attribute value  $E.val$  which is in stack at position  $\text{top-1}$  has to be printed. This can be done using the statement:

```
Print(stack[top-1]);
Top = top - 1
```

- ♦ Consider the production  $E \rightarrow E_1 + T$  and its stack contents before and after reduction



The equivalent statements are:

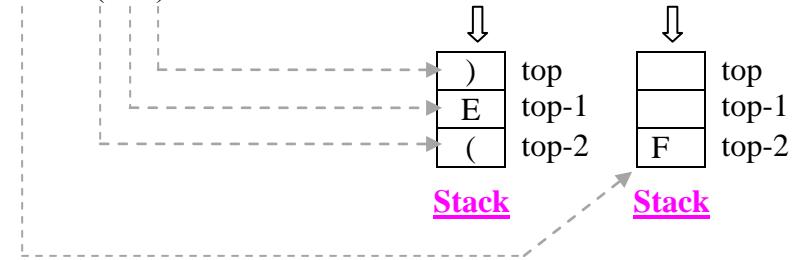
```
stack[top-2].val = stack[top].val + stack[top-2].val
top = top - 2
```

## ■ Introduction to Compiler Design - 5.49

Similarly, for the production  $T \rightarrow T_1 * F$ , we can use the following statements:

```
stack[top-2].val = stack[top].val * stack[top-2].val
top = top -2
```

- ♦ But, for the productions  $E \rightarrow T$  and  $T \rightarrow F$ , no action is necessary because the length of the stack will not change.
- ♦ Consider the production  $F \rightarrow ( E )$  and its stack contents before and after reduction



The equivalent statements are:

```
stack[top-2].val = stack[top-1].val
top = top -2
```

The final actions for the respective productions are shown below:

<u>Productions</u>	<u>Actions</u>
$S \rightarrow E n$	{ print(stack[top-1]); top = top -1 }
$E \rightarrow E_1 + T$	{ stack[top-2].val = stack[top].val + stack[top-2].val top = top -2 }
$E \rightarrow T$	
$T \rightarrow T_1 * F$	{ stack[top-2].val = stack[top].val * stack[top-2].val top = top -2 }
$T \rightarrow F$	
$F \rightarrow ( E )$	{ stack[top-2].val = stack[top-1].val top = top -2 }
$F \rightarrow \text{digit}$	

## 5.50 □ Syntax Directed Translation

---

### 5.7.3 SDT's with actions inside productions

Now, let us see “When action part specified in the production is executed?” An action can be placed at any position within the body of the function. The action is performed immediately after all symbols to its left are processed. Thus, if we have the production:

$$B \rightarrow X \{ a \} Y$$

then action  $a$  is performed after we have recognized  $X$  (if  $X$  is a terminal) or all terminals derived from  $X$  (if  $X$  is a non-terminal)

---

**Example 5.18:** Write the SDT for converting an infix to prefix expression. Show the actions for translating the expression  $2*3+4$  into its equivalent prefix expression

---

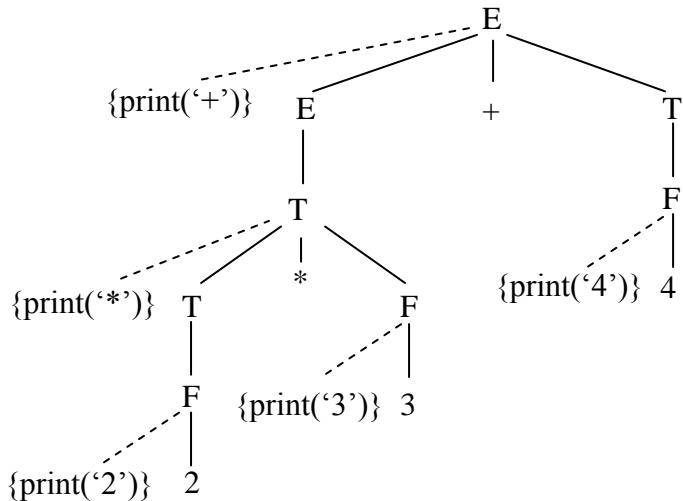
The grammar to generate un-parenthesized arithmetic expression expressed using infix notation consisting of the operators + and \* is shown below:

$$\begin{aligned} E &\rightarrow E_1 + T \\ E &\rightarrow T \\ T &\rightarrow T_1 * F \\ T &\rightarrow F \\ F &\rightarrow 0 \\ F &\rightarrow 1 \\ &\dots\dots \\ &\dots\dots \\ F &\rightarrow 9 \end{aligned}$$

To convert an infix expression to prefix expression, write the parse tree to get the expression  $2*3+4$  and print the operator or operand using the following rules:

- ♦ If only child is present (representing an operand), introduce one left sibling printing the operand
- ♦ If any one of the children contains an operator, introduce leftmost sibling to print the operator
- ♦ The leftmost sibling introduced is shown in dotted arrow for convenience

**Note:** When we draw the parse tree for the translation scheme, indicate an action by introducing leftmost child and connect it to the parent using dashed line as shown in the parse tree. The parse tree to get the expression “ $2 * 3 + 4$ ” is shown below:



Now, traversing in preorder and executing only the actions we get the prefix expression:

+ \* 2 3 4

Now, the SDT for converting an infix expression to postfix expression can be easily written using the above parse tree and is shown below:

#### Actions present in beginning of productions (Prefix)

$E \rightarrow \{ \text{print}( '+') \} E_1 + T$

$E \rightarrow T$

$T \rightarrow \{ \text{print}( '*' ) \} T_1 * F$

$T \rightarrow F$

$F \rightarrow 0 \{ \text{print}( '0') \}$

$F \rightarrow 1 \{ \text{print}( '1') \}$

.....

.....

$F \rightarrow 9 \{ \text{print}( '9') \}$

## 5.52 □ Syntax Directed Translation

---

### Exercises:

- 1) What is semantic analysis? What is syntax directed definition (SDD)?
- 2) What is an attribute? Explain with example. What are the different types or classifications of attributes?
- 3) What is a semantic rule? Explain with example
- 4) What is synthesized attribute? Explain with example
- 5) What is inherited attribute? Explain with example
- 6) What is annotated parse tree? Explain with example
- 7) Write the SDD for the following grammar:

$S \rightarrow En$  where  $n$  represent end of file marker  
 $E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow (E) \mid \text{digit}$

- 8) Write the grammar and syntax directed definition for a simple desk calculator and show annotated parse tree for the expression  $(3+4)*(5+6)$
- 9) What is circular dependency when evaluating the attribute value of a node in an annotated parse tree
- 10) What is the use of inherited attributes
- 11) Obtain SDD and annotated parse tree for the following grammar using top-down approach:

$T \rightarrow T * F \mid F$   
 $F \rightarrow \text{digit}$

- 12) Obtain SDD for the following grammar using top-down approach:

$S \rightarrow En$   
 $E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow ( E ) \mid \text{digit}$

- 13) and obtain annotated parse tree for the expression  $(3 + 4) * (5 + 6)n$
- 14) What is a dependency graph

## Introduction to Compiler Design - 5.53

---

- 15) Obtain the dependency graph for the annotated parse tree obtained in example 5.3
- 16) What is topological sort of the graph
- 17) Give the topological sort of the following dependency graph
- 18) What are different classes of SDD's that guarantee evaluation order
- 19) What is S-attributed definition
- 20) What is an attribute grammar
- 21) What is an L-attributed definition
- 22) What is a side effect in a SDD
- 23) How to control the side effects in SDD
- 24) Write the grammar and SDD for a simple desk calculator with side effect
- 25) Write the SDD for a simple type declaration and write the annotated parse tree and the dependency graph for the declaration “**float a, b, c**”
- 26) Write the SDD for a simple desk calculator. Write the annotated parse tree for the expression  $3 * 5 + 4$
- 27) What is syntax directed translation
- 28) What is a syntax tree? What is the difference between syntax tree and parse tree?”
- 29) For the following grammar show the parse tree and syntax tree for the expression  
 $3 * 5 + 4:$   
 $E \rightarrow E + T \mid E - T \mid T$   
 $T \rightarrow T * F \mid T / F \mid F$   
 $F \rightarrow (E) \mid \text{digit} \mid \text{id}$
- 30) How to construct semantic rules that help us to create syntax trees for the expressions?
- 31) Obtain the semantic rules to construct a syntax tree for simple arithmetic expressions using bottom up approach
- 32) Create a syntax tree for the expression “ $a - 4 + c$ ”
- 33) Obtain the semantic rules to construct a syntax tree for simple arithmetic expressions using top-down approach with operators + and -

## 5.54 □ Syntax Directed Translation

---

- 34) Obtain the semantic rules to construct a syntax tree for simple arithmetic expressions with operators -, +, \* and / using top-down approach
- 35) Give the syntax directed translation of type **int [2][3]** and also given the semantic rules for the respective productions
- 36) What is syntax directed translation scheme
- 37) What is Postfix Syntax-directed-translation or Postfix SDT
- 38) Obtain Postfix SDT implementation of the desk calculator to evaluate the given expression
- 39) Write the SDD and annotate parse tree for converting an infix to postfix expression
- 40) Write the SDT for converting an infix to postfix expression. Show the actions for translating the expression  $2*3+4$  into its equivalent postfix expression
- 41) Explain parser-stack implementation of Postfix SDT's
- 42) Write the actions of desk calculator SDT so that they manipulate the parser explicitly
- 43) When action part specified in the production is executed
- 44) Write the SDT for converting an infix to prefix expression. Show the actions for translating the expression  $2*3+4$  into its equivalent prefix expression