

SPCC

The diagram on the whiteboard illustrates the two-pass assembly process:

```
graph LR; SrcPrg((Src Prg)) -- "S/P" --> Pass1((Pass 1)); Pass1 -- "Data Structure" --> DataStruct; DataStruct -- "Intermediate code" --> Pass2((Pass 2)); Pass2 --> MCCode((M/C Code));
```

Below the video player, the title is "Design Of 2-PASS Assembler Explained in Hindi || System Programming And Operating System".

Design Of 2-PASS Assembler Explained in Hindi || System Programming And Operating System

46,007 views • Oct 25, 2018

1.5K DISLIKE SHARE THANKS CLIP SAVE ...

▶ Design Of 2-PASS Assembler Explained in Hindi || System Programming And Operating System

The whiteboard contains the following information:

Data Structures In Pass-1

- I Assembly Prg.
- II M/C opcode Table (MOT)
- III Symbol table:
Symbol | addr
---|---
- IV Literal table:
Literal | addr
---|---
= '5' | = '5'
- V Pool Table:-
→ → → LTORQ

Mnemonic	Class	m/code	length
STOP	IS	00	1
ADD	IS	01	1
SUB	IS	02	1
MULTI	IS	03	1
MOVER	IS	04	1
MOVEM	IS	05	1
COMB	IS	06	1
BC	IS	07	1
DIV	IS	08	1
READ	IS	09	1
PRINT	IS	10	1
END	AD	02	-
START	AD	01	-
ORIGIN	AD	03	-
EQU	AD	04	-
LTORG	AD	05	-
DS	DL	01	-
DC	DL	02	1
A - C	R9	01 - 03	-

Below the video player, the title is "Data Structures In Pass 1 || 2-PASS Assembler || Explained with Examples in Hindi".

Data Structures In Pass 1 || 2-PASS Assembler || Explained with Examples in Hindi

54,960 views • Oct 25, 2018

1.5K DISLIKE SHARE THANKS CLIP SAVE ...

▶ Data Structures In Pass 1 || 2-PASS Assembler || Explained with Examples in Hindi

Pass 1

Src Code	LC	IC
START 200	200	(AD,01) (CC,200)
MOVER AREQ, = '5'	200	(IS,04) (RG,01) (L,0)
MOVEM AREQ, X	201	(IS,05) (RG,01) (S,0)
L1 MOVE R BREQ, = '2'	202	(S,1) (IS,04) (RG,02) (L,1)
ORIGIN L1 + 3	203	(AD,03) (C,205)
LTORG	205	(AD,05) (DL,02) (C,5)
	206	(AD,05) (DL,02) (C,2)
X DS 1	207	(S,0) (DL,01) (C,1)
END	208	(AD,02)

ST

S	A
0 X	207
1 L1	202

LT

L	A
0 = '5'	205
1 = '2'	206

Pool

O

DS DC

6:49 / 7:41

PASS-1 Of 2-PASS Assembler Explained with Solved Example in Hindi || Part-2 || SPOS

43,802 views • Oct 25, 2018

1.6K

DISLIKE

SHARE

THANKS

CLIP

SAVE

...

Assembly code

	LC	IC	M/C code
START 100	100	(AD,01) (C,100)	(09) (C,1) (109)
READ N	101	(IS,09) (S,0)	(04) (02) (104)
MOVER B, = '1'	102	(IS,04) (RG,02) (L,0)	(05) (02) (110)
MOVEM B, TERM	103	(IS,05) (RG,02) (S,1)	(03) (02) (110)
A MUL B, TERM	104	(S,2) (IS,03) (RG,02) (S,1)	(00) (00) (001)
LTORG	105	(AD,05) (DL,02) (C,1)	(03) (107)
MOVER C, = '2'	106	(IS,04) (RG,03) (L,1)	(04) (02) (108)
MOVEM B, = '5'	107	(IS,05) (RG,02) (L,2)	(05) (02) (102)
LTORG	108	(AD,05) (DL,02) (C,2)	(00) (00) (002)
N DS 1	109	(AD,05) (DL,02) (CC,5)	(00) (00) (005)
TERM DS 1	110	(S,0) (DL,01) (C,1)	—

ST

S	A
0 N	109
1 TERM	110
2 A	103

LT

L	A
0 = '1'	104
1 = '2'	107
2 = '5'	108

POOL

O
1
3

10:41 / 11:26

PASS-2 Of 2-PASS Assembler Explained with Solved Example in Hindi || SPOS

45,242 views • Oct 25, 2018

1K

DISLIKE

SHARE

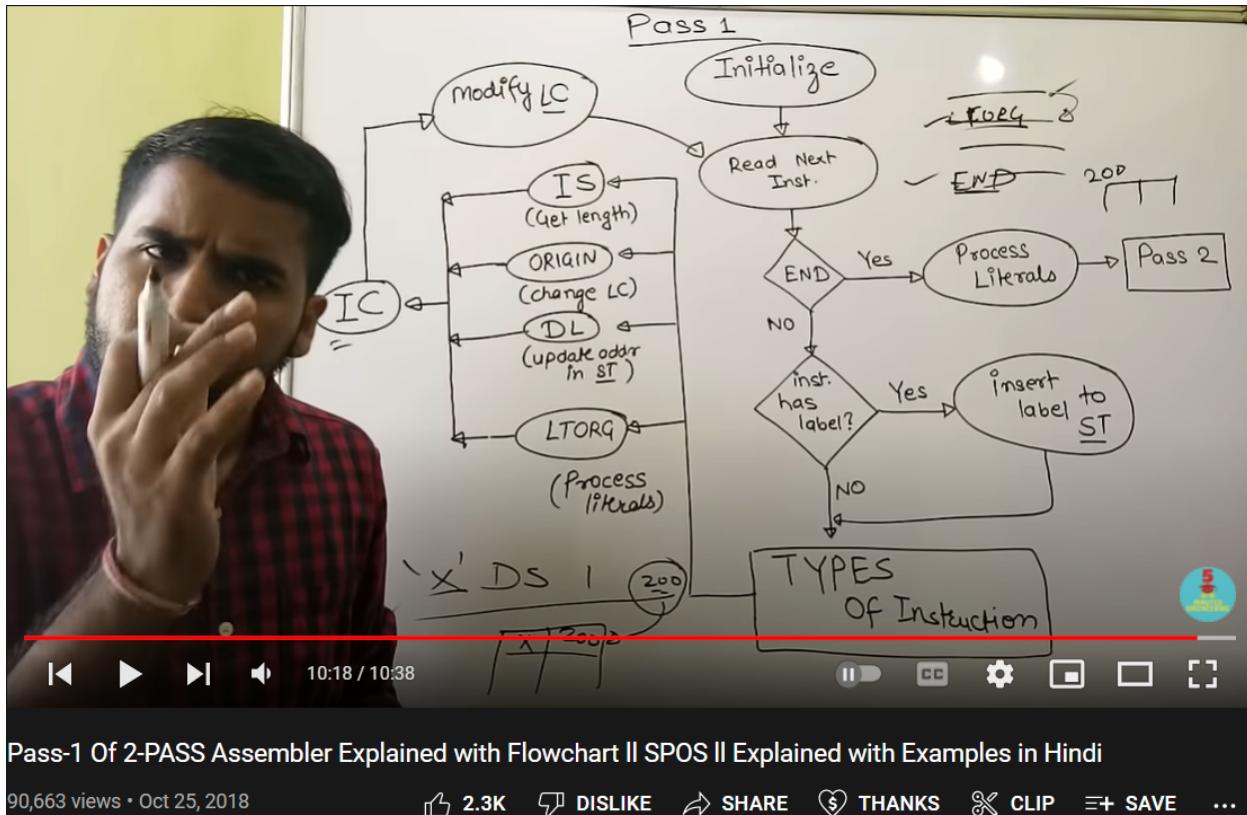
THANKS

CLIP

SAVE

...

PASS-2 Of 2-PASS Assembler Explained with Solved Example in Hindi || SPOS



Pass-1 Of 2-PASS Assembler Explained with Flowchart || SPOS || Explained with Examples in Hindi

90,663 views • Oct 25, 2018

2.3K DISLIKE SHARE THANKS CLIP SAVE ...

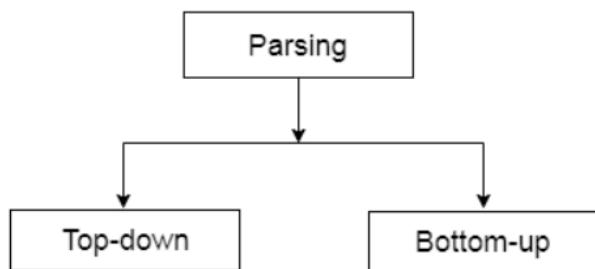
[Pass-1 Of 2-PASS Assembler Explained with Flowchart || SPOS || Explained with Examples in Hindi](#)

Phaser:

Parser is a compiler that is used to break the data into smaller elements coming from lexical analysis phase.

A parser takes input in the form of sequence of tokens and produces output in the form of parse tree.

Parsing is of two types: top down parsing and bottom up parsing.



Top down paring

- The top down parsing is known as recursive parsing or predictive parsing.
- Bottom up parsing is used to construct a parse tree for an input string.
- In the top down parsing, the parsing starts from the start symbol and transform it into the input symbol.

Bottom up parsing

- Bottom up parsing is also known as shift-reduce parsing.
- Bottom up parsing is used to construct a parse tree for an input string.
- In the bottom up parsing, the parsing starts with the input symbol and construct the parse tree up to the start symbol by tracing out the rightmost derivations of string in reverse.

LR Parser



LR parsing is one type of bottom up parsing. It is used to parse the large class of grammars.

In the LR parsing, "L" stands for left-to-right scanning of the input.

"R" stands for constructing a right most derivation in reverse.

"K" is the number of input symbols of the look ahead used to make number of parsing decision.

LR parsing is divided into four parts: LR (0) parsing, SLR(1) parsing, CLR parsing and LALR parsing.

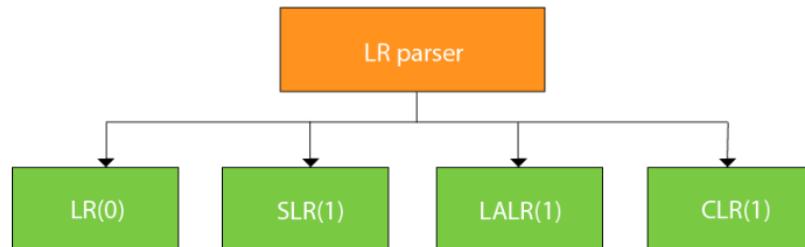


Fig: Types of LR parser

LR algorithm:

The LR algorithm requires stack, input, output and parsing table. In all type of LR parsing, input, output and stack are same but parsing table is different.

Construction of LL(1) Parsing Table

Difficulty Level : Medium • Last Updated : 03 Jun, 2021

Prerequisite - [Classification of top-down parsers, FIRST Set, FOLLOW Set](#)

A top-down parser builds the parse tree from the top down, starting with the start non-terminal. There are two types of Top-Down Parsers:

1. Top-Down Parser with Backtracking
2. Top-Down Parsers without Backtracking

LL(1) Parsing:

Here the 1st L represents that the scanning of the Input will be done from Left to Right manner and the second L shows that in this parsing technique we are going to use Left most Derivation Tree. And finally, the 1 represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

LL(1) parsing is a top-down parsing method in the syntax analysis phase of compiler design. Required components for LL(1) parsing are input string, a stack, parsing table for given grammar, and parser. Here, we discuss a parser that determines that given string can be generated from a given grammar(or parsing table) or not.

The image shows a video frame with handwritten notes on a lined notebook page. The notes are organized into sections:

- Top section:** Includes "end → MEND", "MACRO", "(DEC)", "EX, RY", "MOVER BRC, EX", "SUR BRG, EX", "MOUIM CREG, EX", and "MIND".
- START section:** Includes "MOSEM ARGS, B", "DECRA, B", "MULT DEC, B", "INCR B, A", "DEC B, A", "A DS T", and "D BS 10".
- MNT section:** Includes "I. INCR", "5. DEC", "4. MEND", and "7. MOUIM CREC, #2".
- MDT section:** Includes "2. MOUIM ARGS, #1", "3. MOUIM ARGS, #1", and "6. SUB IIG, #1".
- MA section:** Includes "#1 A", "#2 B", and "#3 C".
- Bottom section:** Includes "7. MOUIM CREC, #2", "A FOR DEC", "X #2", and "Y #3".

Below the notes is a video player interface with controls for play/pause, volume, and progress (3:35 / 8:24). The video has a caption "Macro-Processor.MNT,MDT,ALA,IC,EXPANDED." and a view count of 1,312 views from Feb 27, 2020.

Specify the problem

- In Pass-I the macro definitions are searched and stored in the macro definition table and the entry is made in macro name table
- In Pass-II the macro calls are identified and the arguments are placed in the appropriate place and the macro calls are replaced by macro definitions.

Specification of databases:-

Pass 1:-

- The input macro source program.
- The output macro source program to be used by Pass2.
- Macro-Definition Table (MDT), to store the body of macro def^{ns}.
- Macro-Definition Table Counter (MDTC), to mark next available entry MDT.
- Macro- Name Table (MNT), used to store names of macros.
- Macro Name Table counter (MNTC), used to indicate the next available entry in MNT.
- Argument List Array (ALA), used to substitute index markers for dummy arguments before storing a macro-def^{ns}.

Specification of databases:-

- **Pass 2:-**

- The copy of the input from Pass1.
- The output expanded source to be given to assembler.
- MDT, created by Pass1.
- MNT, created by Pass1.
- Macro-Definition Table Pointer (MDTP), used to indicate the next line of text to be used during macro-expansion.
- Argument List Array (ALA), used to substitute macro-call arguments for the index markers in the stored macro-def^{ns}

MACROS	PROCEDURE
¹ The corresponding machine code is written every time a macro is called in a program.	1 The Corresponding m/c code is written only once in memory
² Program takes up more memory space.	2 Program takes up comparatively less memory space.
³ No transfer of program counter.	3 Transferring of program counter is required.
⁴ No overhead of using stack for transferring control.	4 Overhead of using stack for transferring control.
⁵ Execution is fast	5 Execution is comparatively slow.
⁶ Assembly time is more.	6 Assembly time is comparatively less.
⁷ More advantageous to the programs when repeated group of instruction is too short.	7 More advantageous to the programs when repeated group of instructions is quite large.

AD = assembler directive

IS = imperative statements

DL= DS & DC = declaration statements & declaration counters

RG = registers

CC= comparison condition