System Programming and Compiler Design

Assembler

Why System Programing and Compilers?

- Gives insight into design of programming languages.
- It is for doing low level optimization and finding nasty bugs. Knowing how a compiler works will also make you a better programmer and increase your ability to learn new programming languages quickly.
- It touches upon many of the subareas -- computer architecture, general software issues, language and automata theory.
- The objective of the course is to find solutions to problems which are typically encountered when analyzing, translating, and executing programs on machines.

System vs Application Software

- Software in computers can be divided into broadly 2 categories:
- Application software: written by users for their computational requires eg database program, web browsers etc.
- System software: written by manufacturer or system administrator for proper and easy use of system and maintenance. System software closer to actual hardware than application program. It is designed to run a computer’s hardware and application programs

System Software

- Major system software:
- Operating System
- Compiler
- Assembler
- Linker
- Loader
- Text editor
- Debugger

Assembler

- Assembler is a tool to convert assembly language program into machine language.
- Assembly language is higher than machine language but lower than a high level language. It consists of instructions that are mnemonic codes for corresponding machine language instructions.
- Machine language is combination of bits (0s/1s) this is the only language computer can understand.

Assembler

- Machine language is combination of bits (0s/1s) this is the only language computer can understand.

```
0010000100100011
```

```
0010|0001|0010|0011

operation type  source register  other source  destination register
0010            0001             0010          0011
```

These values correspond to:

```
operation type 0010 = addition
source register 0001 = register 1
other source 0010 = register 2
destination register 0011 = register 3
```

So our original instruction from above could look like:

(meaning)	operation type	source register	other source	destination register
(machine code)	0010	0001	0010	0011
("English")	add	r1	r2	r3

Why we need assembly ?

Machine code has no single operation to perform multiple operations with multiple variables at once so it has to be converted to a sequence of machine code first y=(a+b)*(c+d)*(e+f).

If converted to machine directly by the compiler output would be list of bits, debugging tough, ask compiler to give kind of output you actually understand

- Assembly language provides you the privilege of controlling the total machine-cycles of your program, which other higher level language can't provide. The other use is size/speed optimization of code

Simple Manual Assembler

PROGRAM 3.1: A program to add 10 numbers.

```
 1              X dd 10, 20, 40. 5. 7. 9, 53, 8, 11, 13
 2              sum dd 0
 3              MVI I, X
 4              MVI B, 0
 5              MVI C, 0
 6        L1:   LOADI
 7              ADD C
 8              MOV C,
 9              INC B
10              CMP B,
11              JE L2
12              ADDI 4
13              JMP L1
14        L2:   STORE sum
15              STOP
```

Table 3.1: Machine opcode table for the simple processor.

Instruction	Size (in bytes)	Format	Meaning
MVI *A*, <*constant*>	5	<0, 4 byte constant>	$A \leftarrow$ *constant*
MVI *B*, <*constant*>	5	<1, 4 byte constant>	$B \leftarrow$ *constant*
MVI *C*, <*constant*>	5	<2, 4 byte constant>	$C \leftarrow$ *constant*
MVI *I*, <*constant*>	5	<3, 4 byte constant>	$I \leftarrow$ *constant*
LOAD <*constant*>	5	<4, 4 byte constant>	$A \leftarrow$ memory[<*constant*>]
STORE <*constant*>	5	<5, 4 byte constant>	memory[<*constant*>] $\leftarrow A$
LOADI	1	< 6 >	$A \leftarrow$ memory[I]
STORI	1	< 7 >	memory[I] $\leftarrow A$
ADD *B*	1	< 8 >	$A \leftarrow A + B$
ADD *C*	1	< 9 >	$A \leftarrow A + C$
MOV *A*, *B*	1	< 10 >	$A \leftarrow B$
MOV *A*, *C*	1	< 11 >	$A \leftarrow C$
MOV *B*, *C*	1	< 12 >	$B \leftarrow C$
MOV *B*, *A*	1	< 13 >	$B \leftarrow A$
MOV *C*, *A*	1	< 14 >	$C \leftarrow A$
MOV *C*, *B*	1	< 15 >	$C \leftarrow B$
INC *A*	1	< 16 >	$A \leftarrow A + 1$
INC *B*	1	< 17 >	$B \leftarrow B + 1$
INC *C*	1	< 18 >	$C \leftarrow C + 1$
CMP *A*, <*constant*>	5	<19, 4 byte constant>	compare *A* to *constant*
CMP *B*, <*constant*>	5	<20, 4 byte constant>	compare *B* to *constant*
CMP *C*, <*constant*>	5	<21, 4 byte constant>	compare *C* to *constant*
ADDI <*constant*>	5	<22, 4 byte constant>	$I \leftarrow I +$ *constant*
JE <*label*>	5	<23, 4 byte address>	jump on equal to <*label*>
JMP <*label*>	5	<24, 4 byte address>	jump to <*label*>
STOP	1	< 25 >	stop the processor

Simple Manual Assembler

Table 3.2: The symbol table of the program.

Name	Type	Offset
X	*variable*	0
sum	*variable*	40
L1	*label*	59
L2	*label*	83

```
 1 00000000 0A000000140000000      X dd 10, 20, 40, 5, 7, 9, 53, 8, 11, 13
            2800000005000000
            0700000009000000
            3500000008000000
            0B000000D000000
 2 00000028 00000000               sum dd 0
 3 0000002C 03000000               MVI I, X
 4 00000031 01000000               MVI B, 0
 5 00000036 02000000               MVI C, 0
 6 0000003B 06               L1:   LOADI
 7 0000003C 09                     ADD C
 8 0000003D 0E                     MOV C, A
 9 0000003E 11                     INC B
10 0000003F 140A000000             CMP B, 10
11 00000044 1753000000             JE L2
12 00000049 1604000000             ADDI 4
13 0000004E 183B000000             JMP L1
```

Assembler Design Process

Program in assembly language is composed of following components:

- **Machine Instruction**: Tell the processor the exact operation to be performed eg MOV, ADD, SUB
- **Variable declaration**: Declaration of storage space e.g. db, dw, etc.
- **Assembler Directives**: Direct assembler to produce code in structured manner so that different sections of program can be handled efficiently e.g. section, END
- Comments:

MOV AX,39 ;move 39 to AX

Major Data Structure Used

- Machine Opcode Table (MOT)
- Pseudo Opcode Table (POT)
- Symbol Table (SYMTAB)

Major Data Structure Used

- Machine Opcode Table (MOT): static, contents do not change during assembler's lifetime. Holds opcode used by processor for different instruction mnemonics.

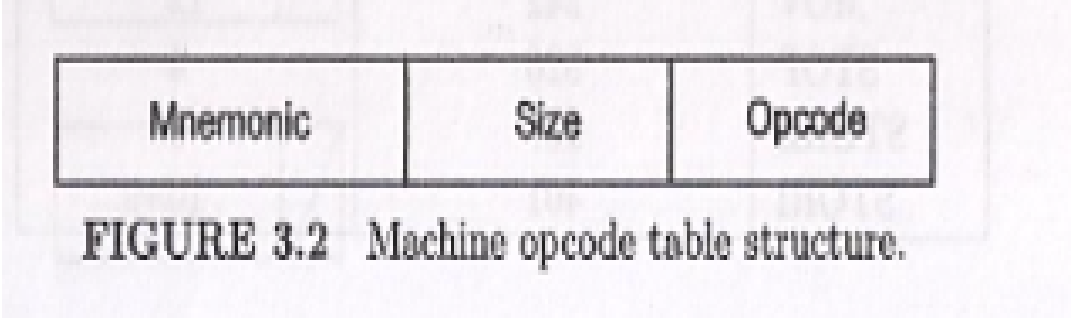


FIGURE 3.2 Machine opcode table structure.

Mnemonic: holds instruction mnemonics

Size: holds size of instruction

Opcode: machine code corresponding to mnemonic

Organization of MOT

- Organization of table determined by insertion, deletion, search, update etc.
- Being static MOT does not need insertion and deletion, only searching is there.
- For O(1) constant time search, hash table.
- Good collision resolution strategies needed

Organization of MOT

h(s)=(Sum of ASCII values of characters of s) mod 23.
Collision for ADD and CMP, for mnemonics LOAD and MOV, MVI and STORE.

Table 3.4: Hash table index calculation.

Mnemonics	Sum of ASCII values	hash index ($h(s)$)
ADD	201	17
ADDI	274	21
CMP	224	17
INC	218	11
JE	143	5
JMP	231	1
LOAD	288	12
LOADI	361	16
MVI	236	6
MOV	242	12
STOP	326	4
STORE	397	6
STORI	401	10

Organization of MOT

- Binary Search Tree
- -Linked List indexed by first alphabet of mnemonic.

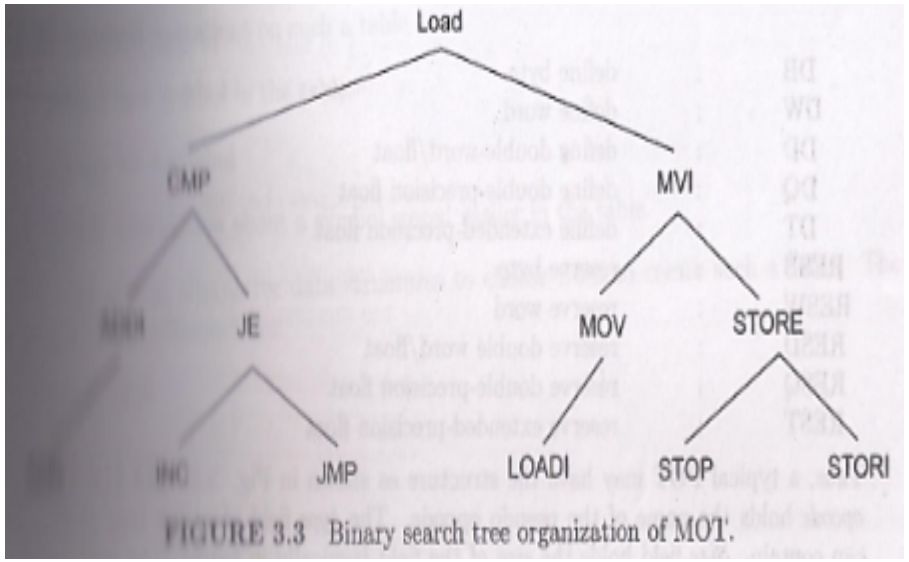


FIGURE 3.3 Binary search tree organization of MOT.

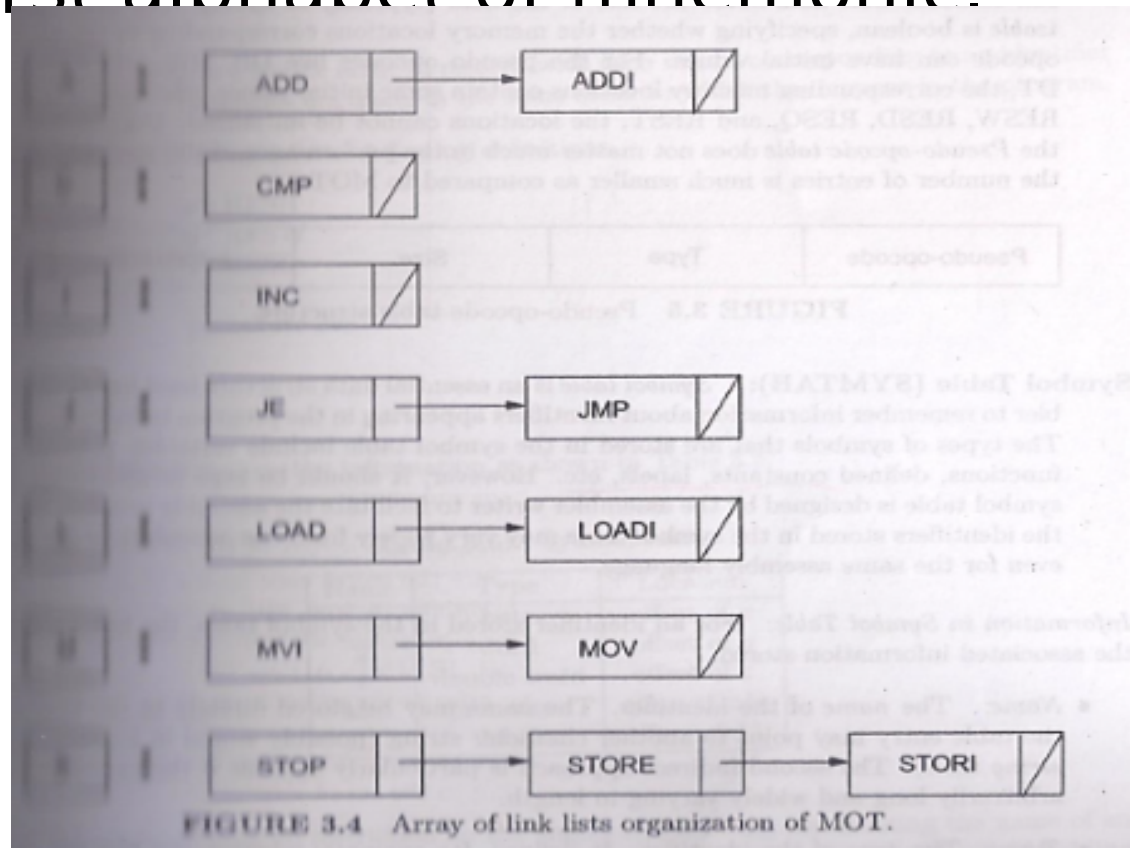


FIGURE 3.4 Array of link lists organization of MOT.

Pseudo Opcode Table (POT)

- It contains the pseudo opcodes supported by assembler. They are Used to reserve memory space and initialize it.

DB	:	define byte
DW	:	define word
DD	:	define double-word/float
DQ	:	define double-precision float
DT	:	define extended-precision float
RESB	:	reserve byte
RESW	:	reserve word
RESD	:	reserve double word/float
RESQ	:	reserve double-precision float
REST	:	reserve extended-precision float

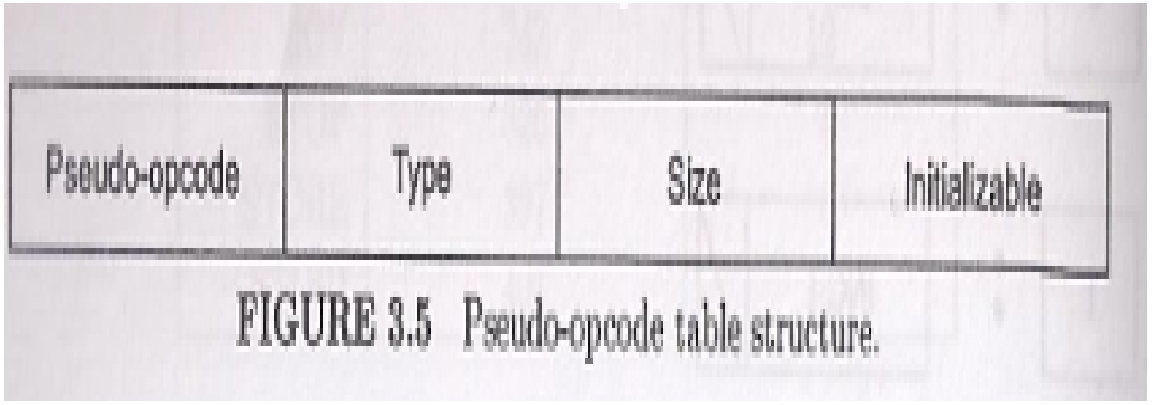


FIGURE 3.5 Pseudo-opcode table structure.

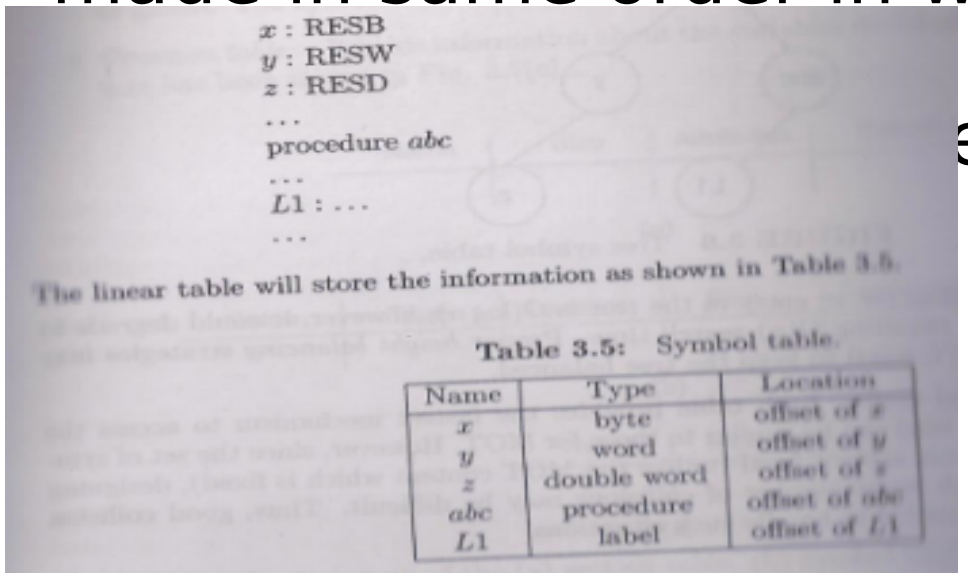
Psuedo-opcode: holds name of pseudo opcode.
type: type of data it can contain.
Size: holds size of field.
Initializable: memory can be initialized or not

Symbol Table (SYMTAB)

- Stores variables, procedures, functions, defined constants, labels etc.
- For identifier stored in table, following are associated information:
- Name, Type, Location.
- Fundamental operations on such a table are:
- Enter a new symbol in table
- Lookup for symbol
- Modify information about symbol

Symbol Table (SYMTAB)

- Data structure to create table are:
- Linear Table, Ordered list, Tree, Hash Table
- Linear Table: Simple array of records with each record corresponding to an identifier of program. Entries are made in same order in which they appear in the and modify operations eing number of



x : RESB
y : RESW
z : RESD
...
procedure *abc*
...
$L1$: ...
...

The linear table will store the information as shown in Table 3.5.

Table 3.5: Symbol table.

Name	Type	Location
x	byte	offset of x
y	word	offset of y
z	double word	offset of z
abc	procedure	offset of *abc*
$L1$	label	offset of $L1$

- Ordered List: Variation of linear table in which a list organization is used. List sorted in some fashion and binary search can be used to access table O(log n) time. Insertion may be costly as need to main sorted order.
- Tree:

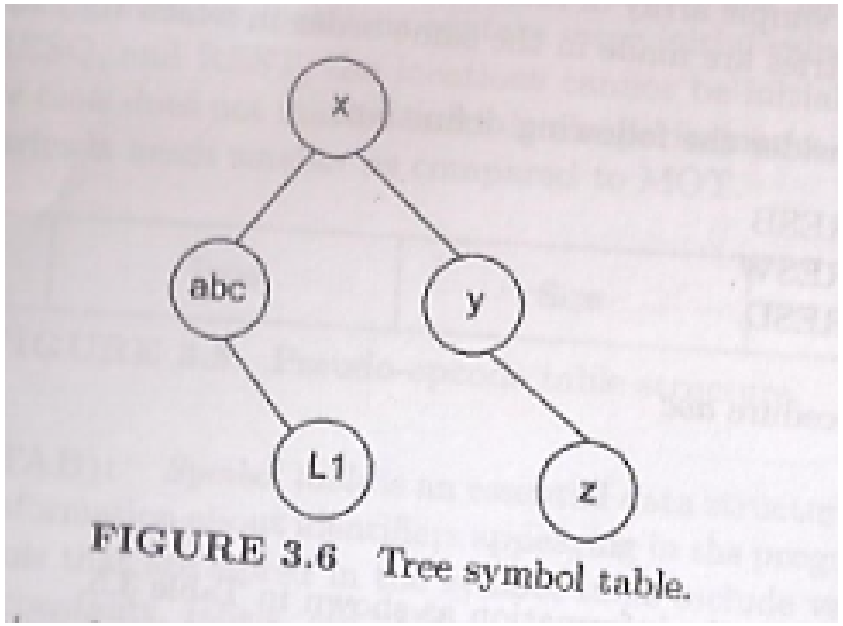


FIGURE 3.6 Tree symbol table.

- Hash: Since dynamically we add or delete entries, designing good hash function may be difficult.

Two Pass Assembler

- In pass 1, different data structures (tables) are filled up with information pertaining to symbols and sections whereas in second pass actual code is generated.
- Both passes uses a variable, location counter (lc) to compute current offset from beginning of section

Two Pass Assembler

- In first pass, it fills tables :

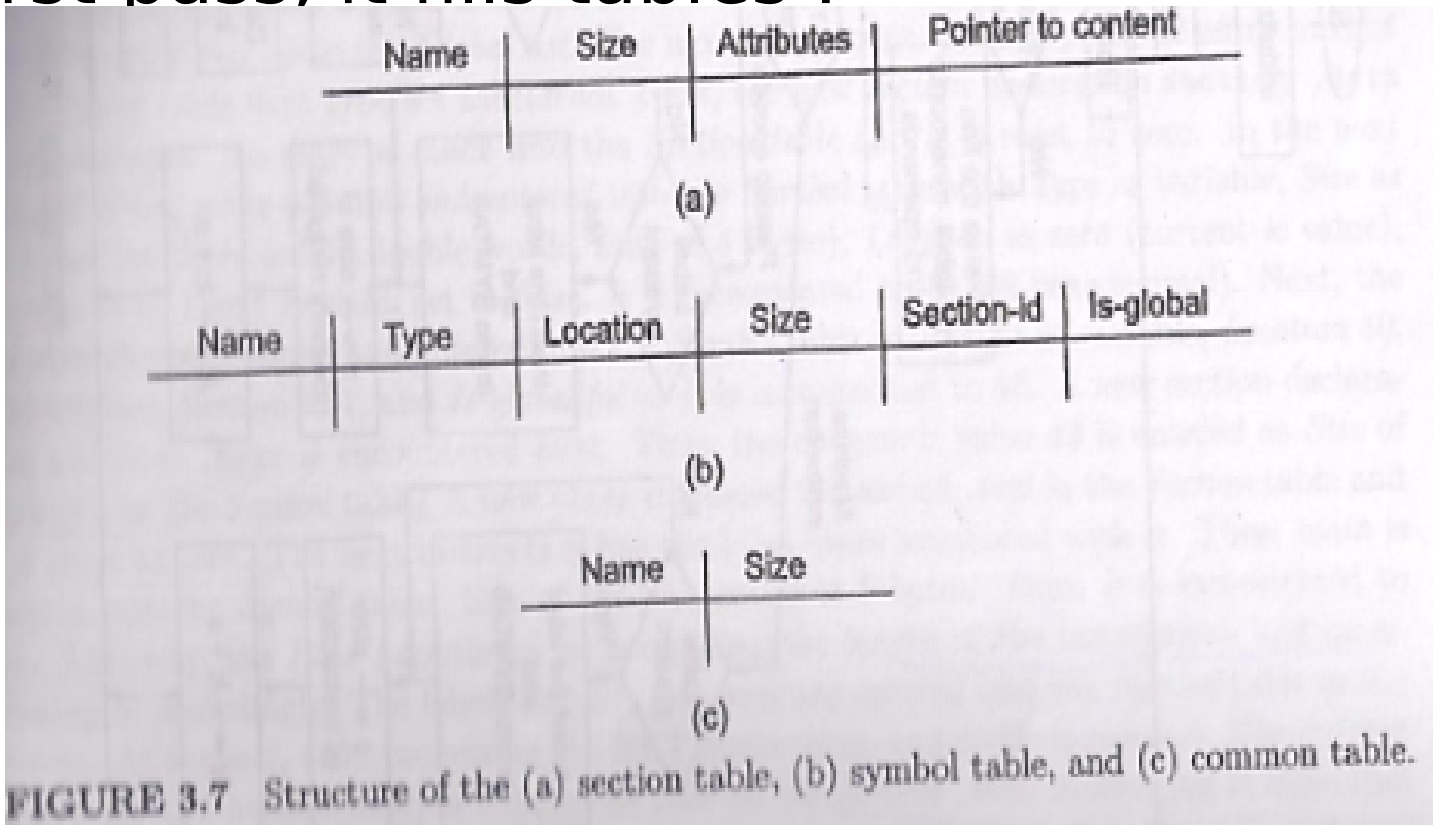


FIGURE 3.7 Structure of the (a) section table, (b) symbol table, and (c) common table.

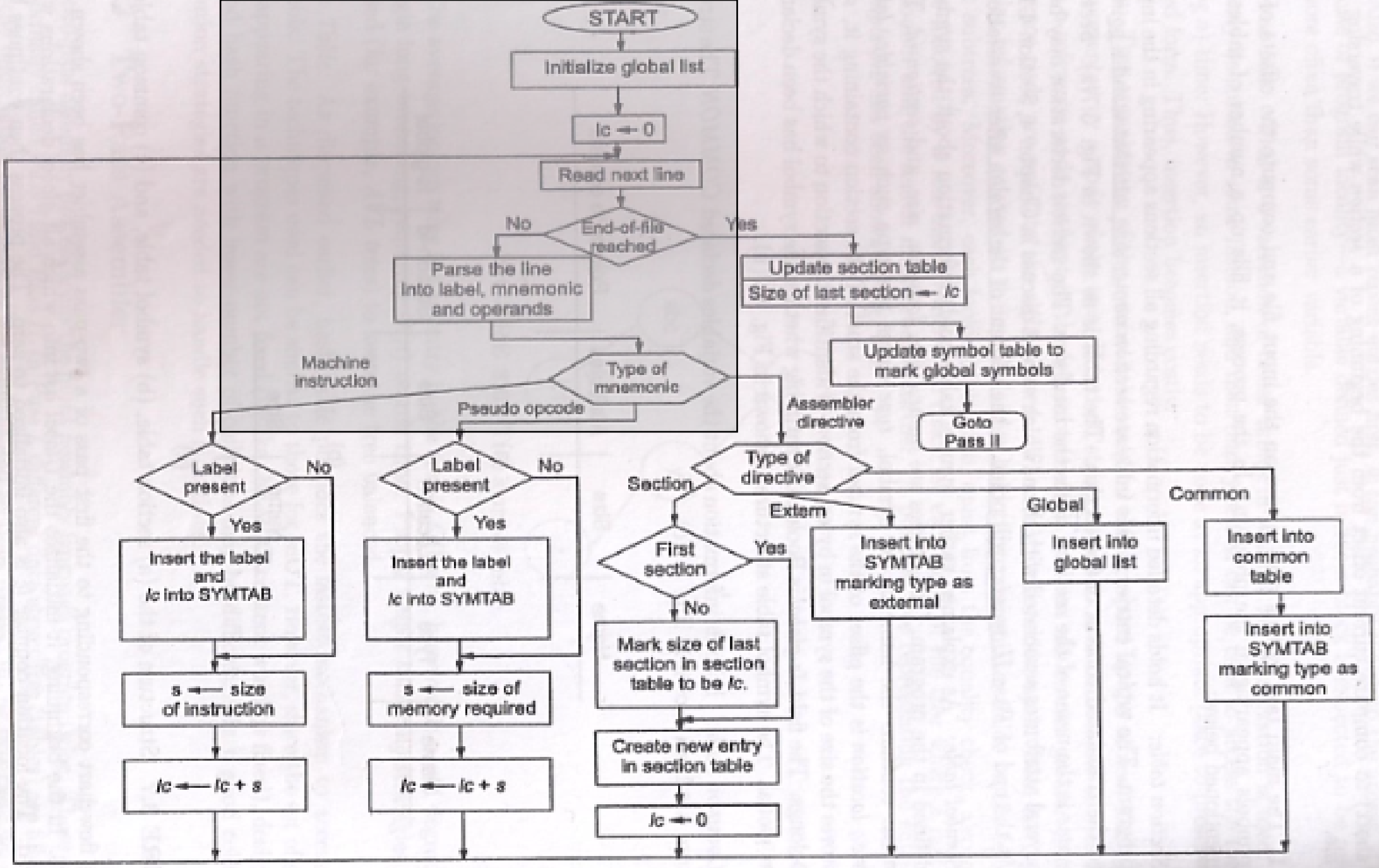


FIGURE 3.8 Pass I of the two-pass assembler.

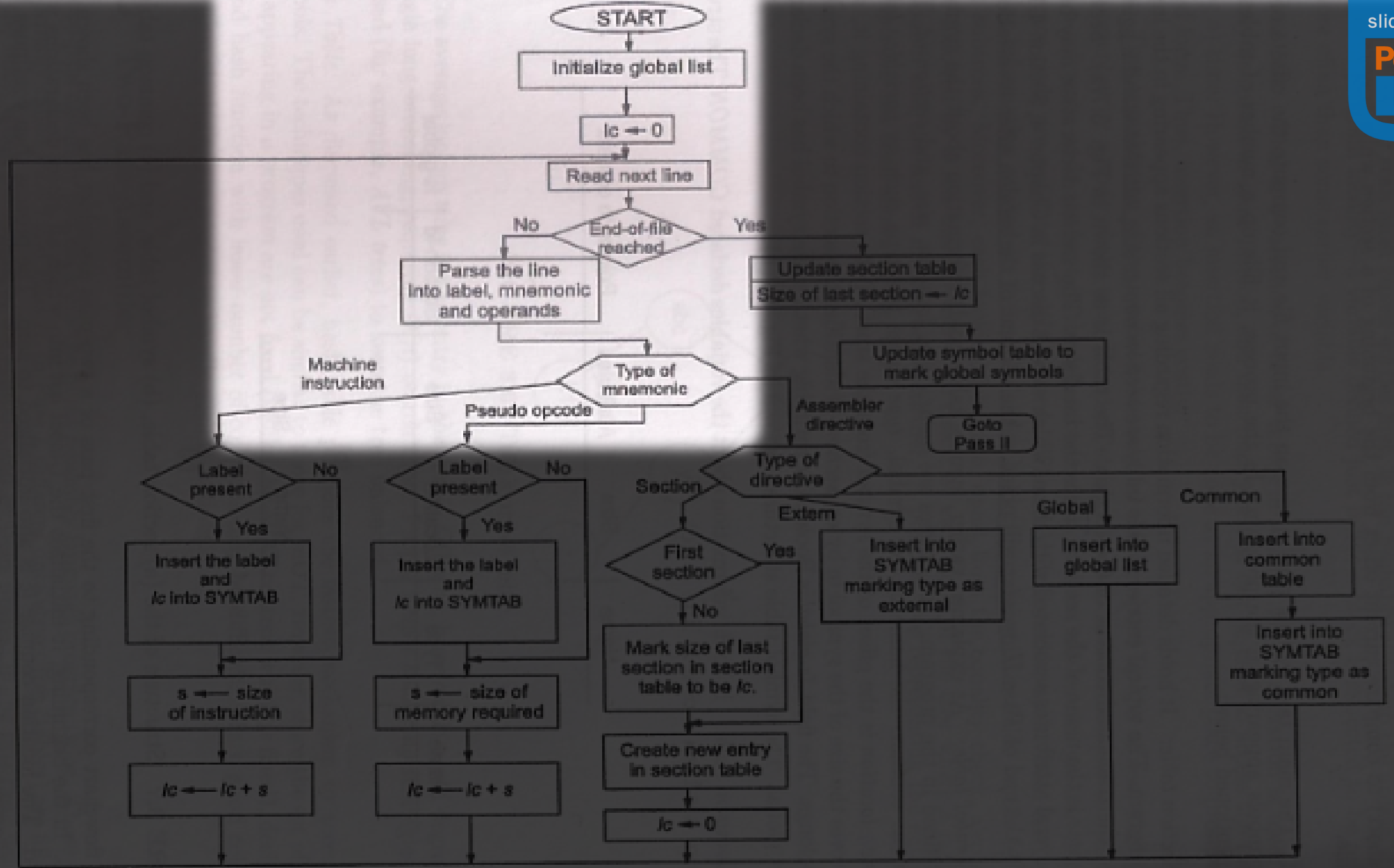


FIGURE 3.8 Pass I of the two-pass assembler.

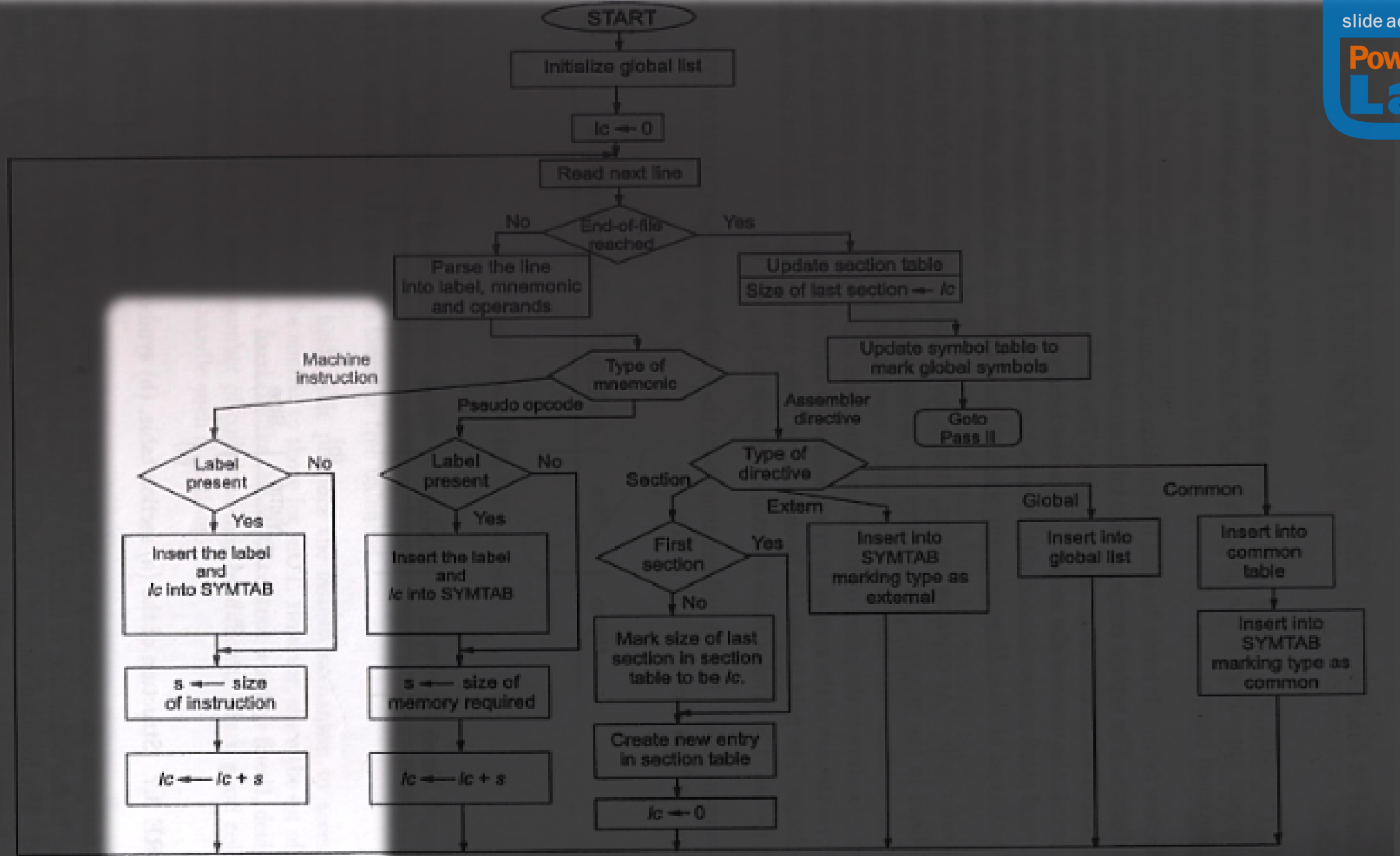


FIGURE 3.8 Pass I of the two-pass assembler.

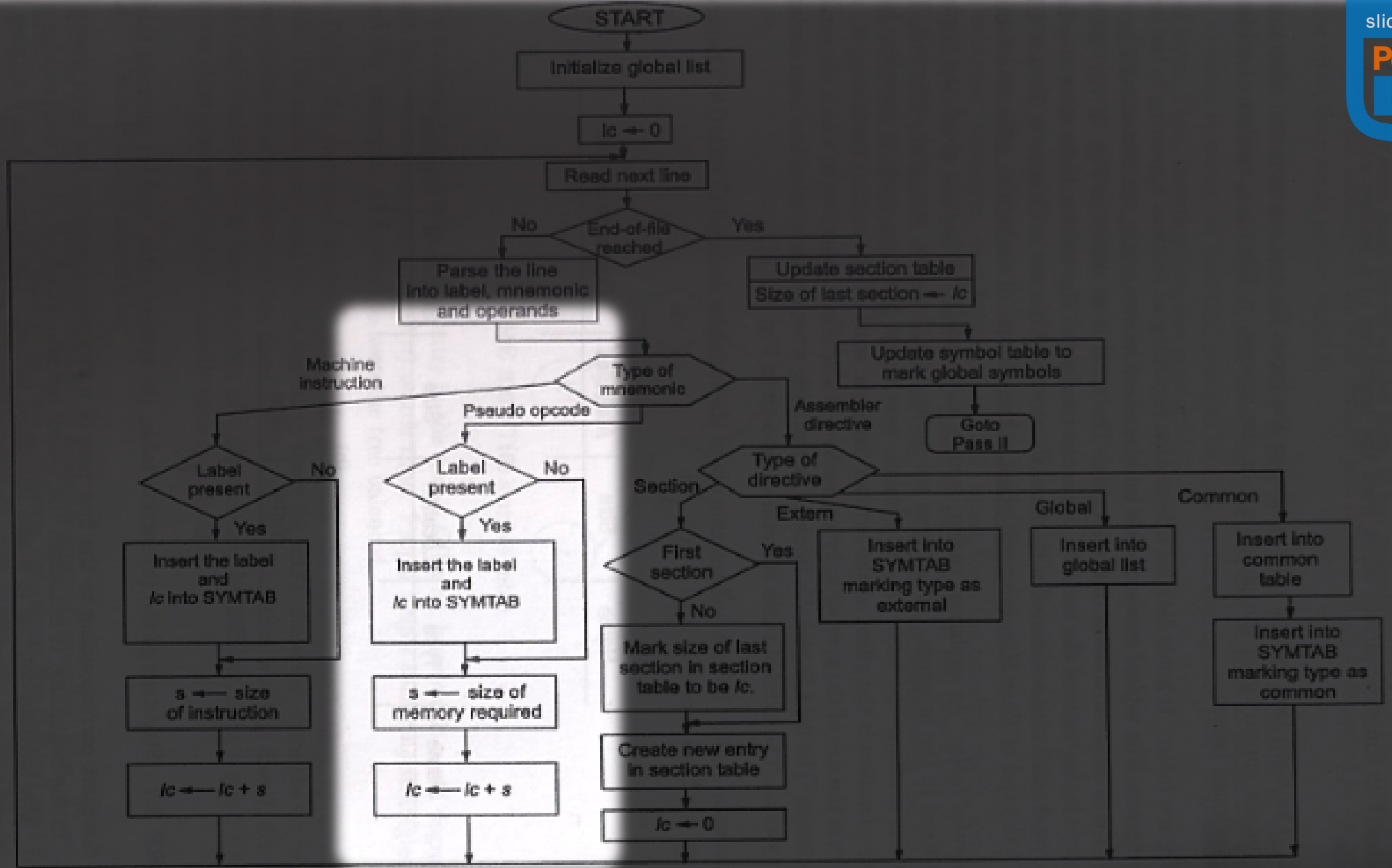


FIGURE 3.8 Pass I of the two-pass assembler.

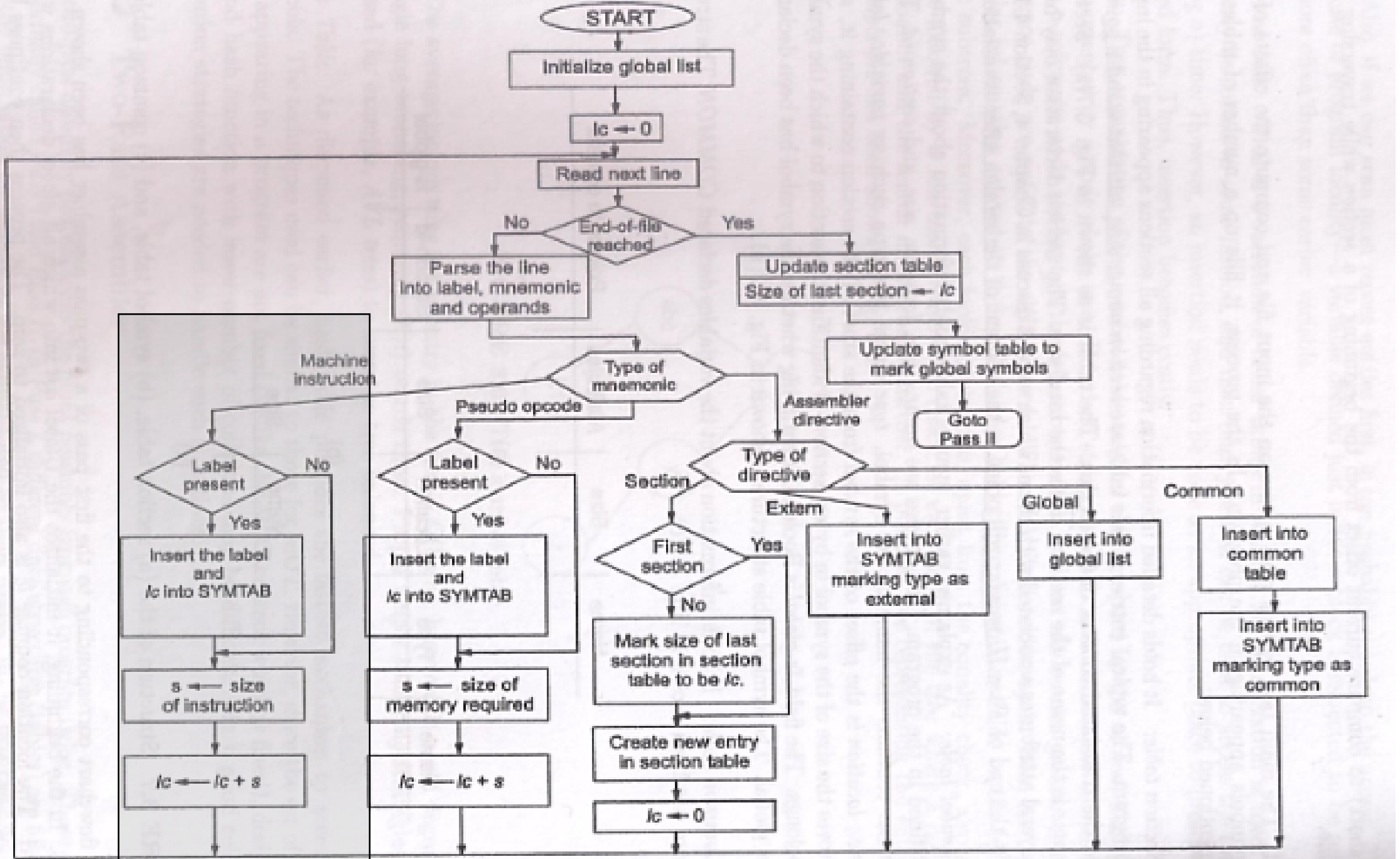


FIGURE 3.8 Pass I of the two-pass assembler.

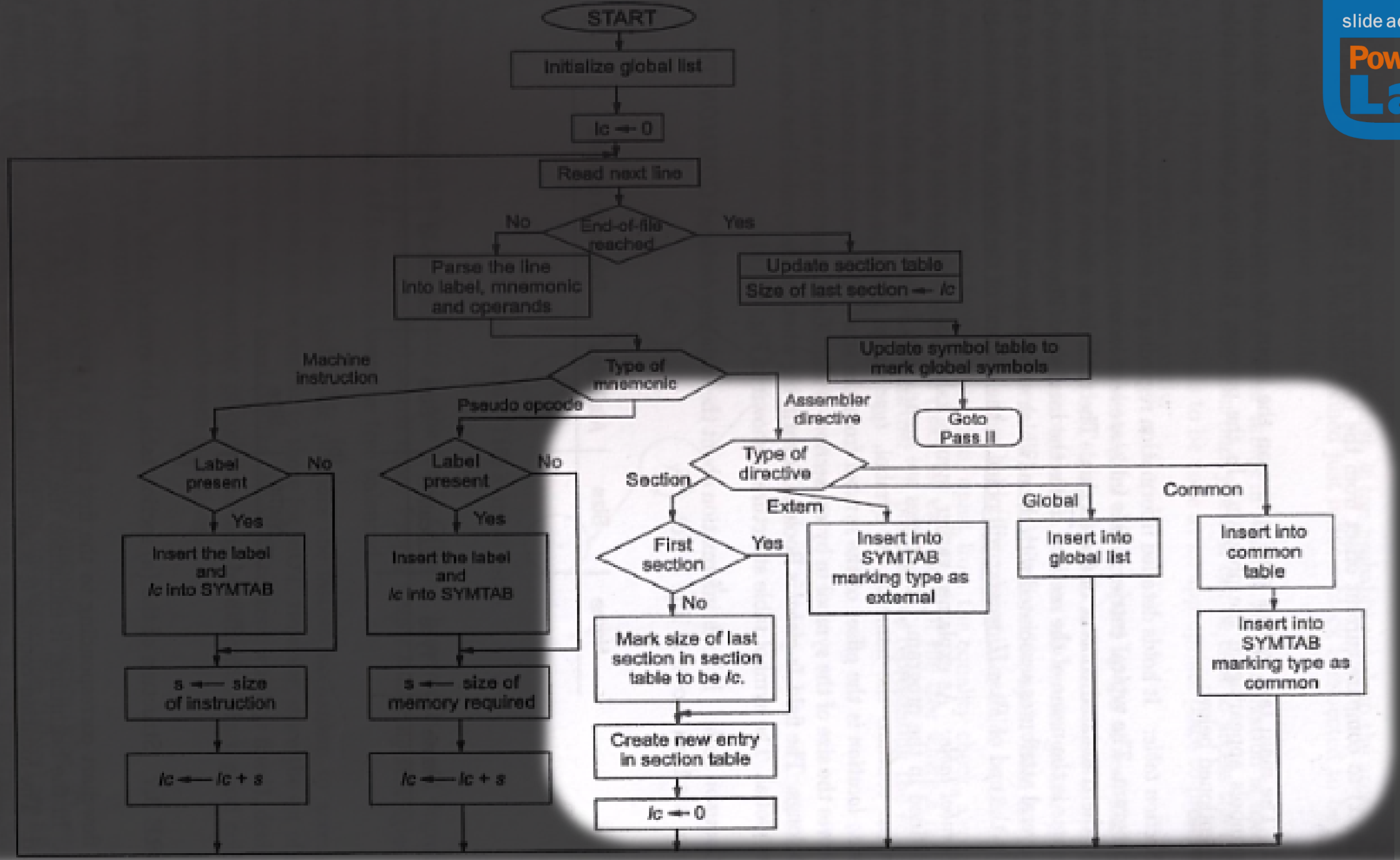


FIGURE 3.8 Pass I of the two-pass assembler.

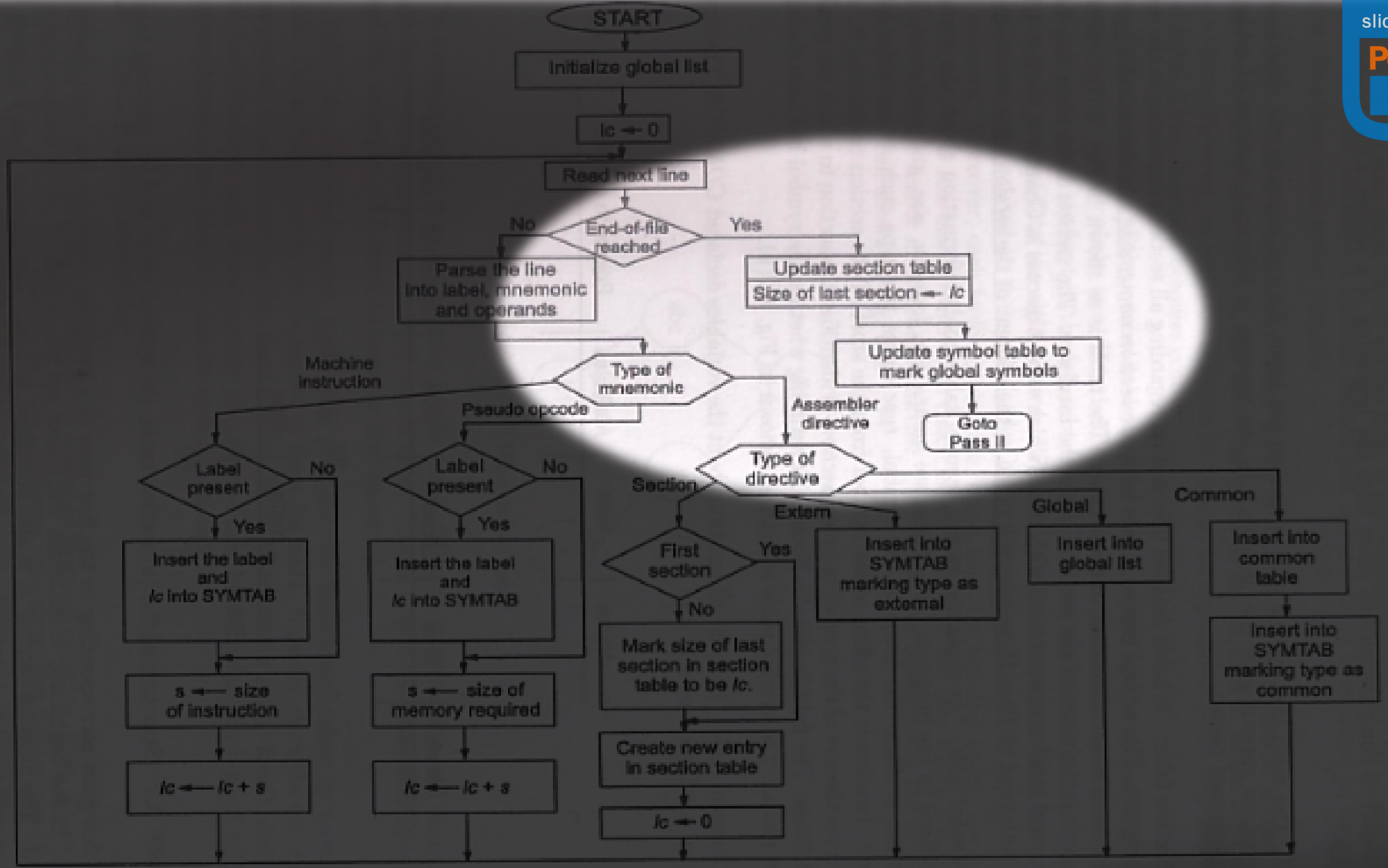


FIGURE 3.8 Pass I of the two-pass assembler.

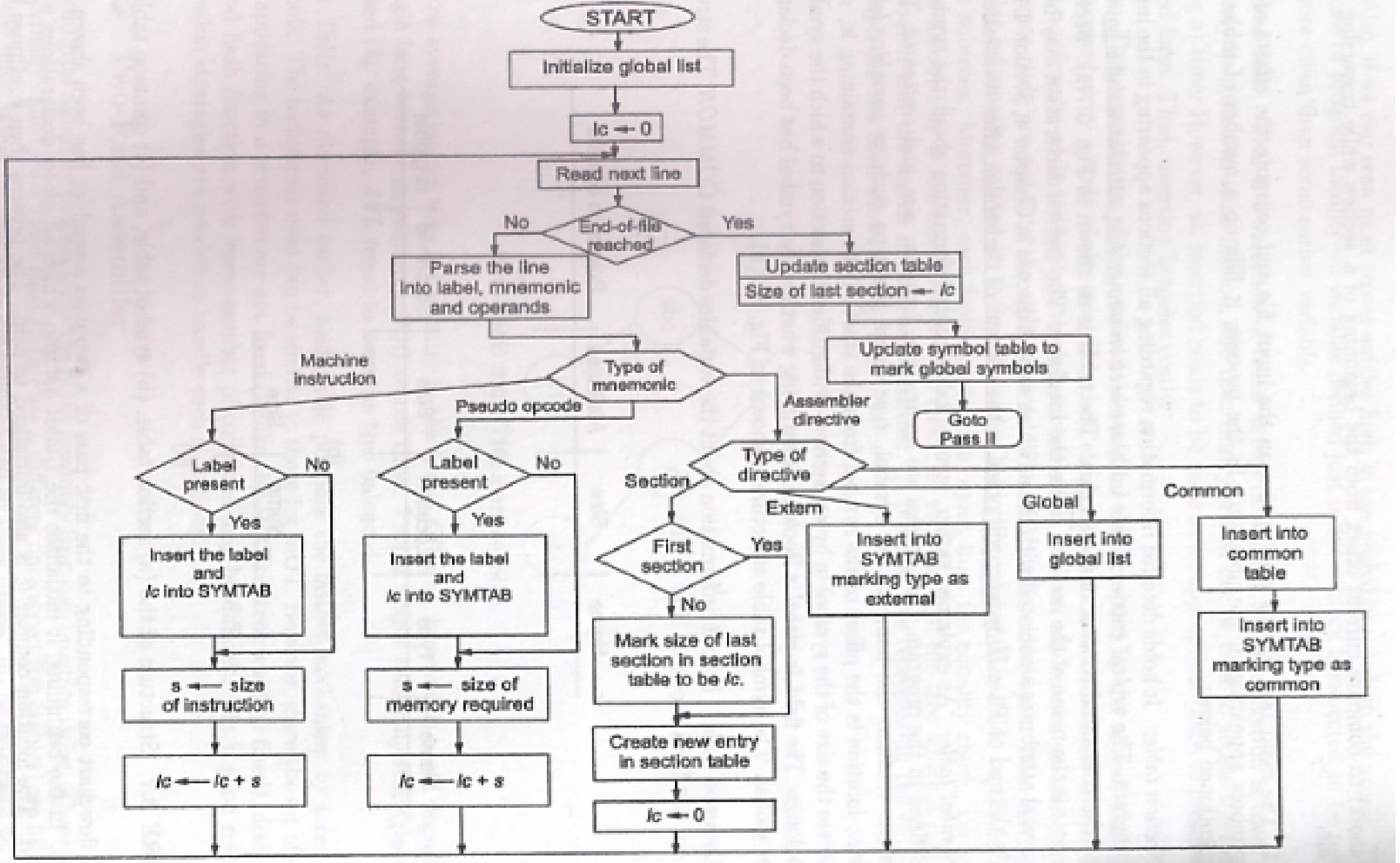


FIGURE 3.8 Pass I of the two-pass assembler.

PROGRAM 2.2: List file of the program to find the maximum.

```
                                        global main
                                        extern printf

                                        section .data
                                        my_array:
00000000 0A0000001400000001E-               dd 10, 20, 30, 100, 200, 56, 45,
00000009 00000064000000C800-                67, 89, 77
00000012 00003800000002D0000-
0000001B 00430000005900000-
00000024 4D000000

                                        format:
00000028 2564400A00                         db '%d', 10, 0
                                        section .text
                                        main:
00000000 B900000000                         MOV ECX, 0
00000005 A1[00000000]                       MOV EAX,  [my_array]
                                        L2:
0000000A 41                                 INC ECX
0000000B 81F90A000000                       CMP ECX, 10
00000011 7412                               JZ over

00000013 3B048D[00000000]                   CMP EAX, [my_array + ECX*4]
0000001A 7D07                               JGE L1
0000001C 8B048D[00000000]                   MOV EAX, [my_array + ECX*4]
                                        L1:
00000023 EBE5                               JMP L2

                                        over:
00000025 50                                 PUSH EAX
00000026 68[28000000]                       PUSH dword format
0000002B E8(00000000)                       CALL printf
00000030 81C408000000                       ADD ESP, 8


00000036 C3                                 RET

```

Name	Size	Attributes	Pointer to content
.data	43		
.text	55		

(a)

Name	Type	Location	Size	Section-id	Is-global
printf	external				
my-array	variable	0	40	1	false
format	variable	40	3	1	false
main	label	0		2	true
L2	label	10		2	false
L1	label	35		2	false
over	label	37		2	false

(b)

FIGURE 3.9 (a) Section table, and (b) Symbol table for the example.

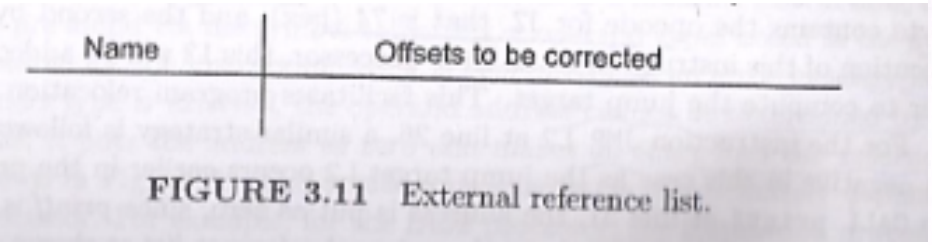


FIGURE 3.11 External reference list.

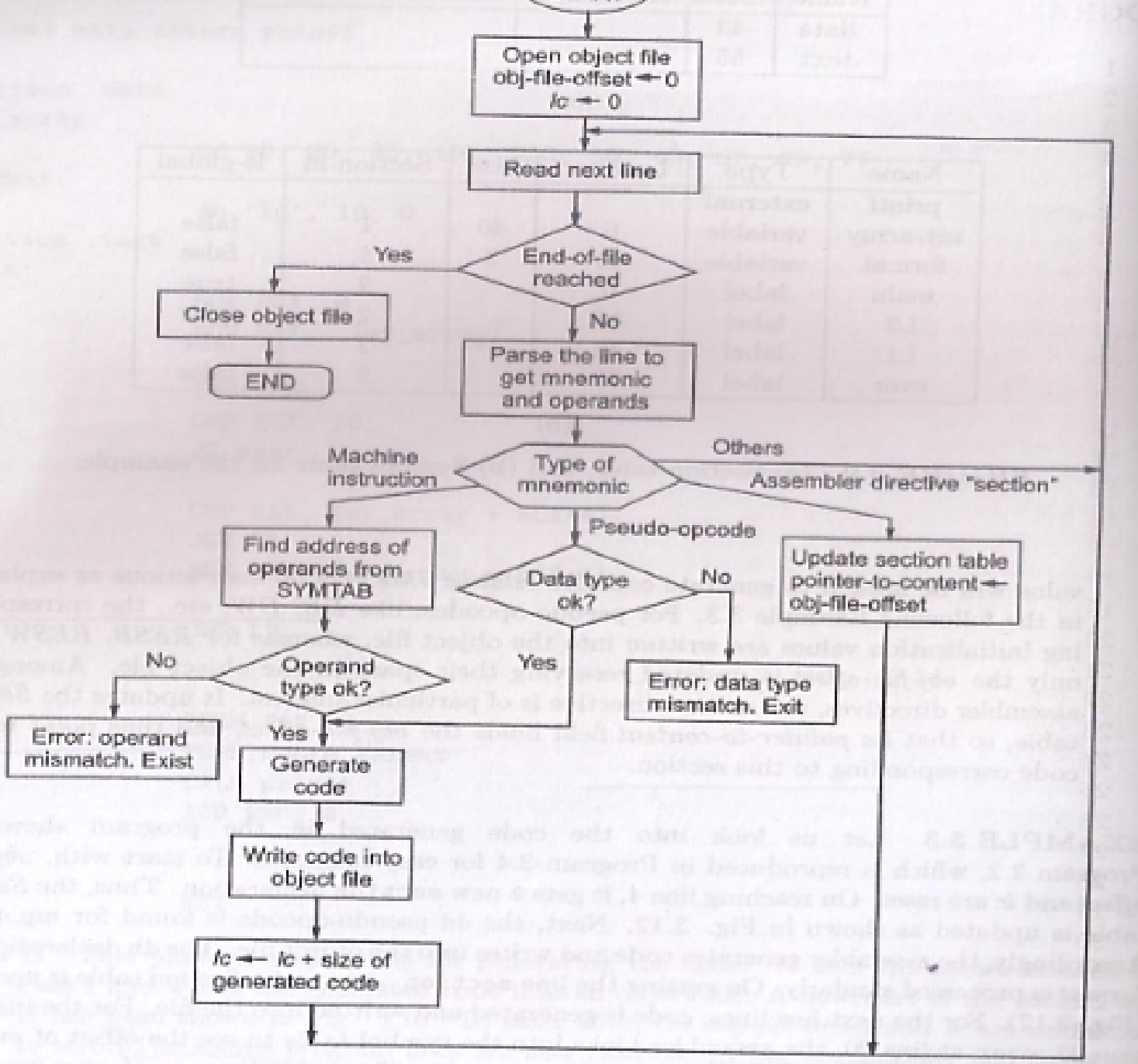


FIGURE 3.10 Pass II of a two-pass assembler.

PROGRAM 3.4: Code generated for the program to find the maximum.

```
1
2
3
4
5
6  00000000 0A0000001400000001E-
7  00000009 0000006400000C800-
8  00000012 0000380000002D0000-
9  0000001B 0043000000590000000-
10 00000024 4D000000
11
12 00000028 25640A00
13
14
15 00000000 B900000000
16 00000005 A1[00000000]
17
18 0000000A 41
19 0000000B 81F90A000000
20 00000011 7412
21
22 00000013 3B048D[00000000]
23 0000001A 7D07
24 0000001C 8B048D[00000000]
25
26 00000023 EBE5
27
28
29 00000025 50
30 00000026 68[28000000]
31 0000002B E8(00000000)
32 00000030 81C408000000
33
34
35 00000036 C3
```

```
global main
extern printf

section .data
my_array:
    dd 10, 20, 30, 100, 200, 56, 45,
       67, 89, 77


format:
    db '%d', 10, 0
section .text
main:
    MOV ECX, 0
    MOV EAX,  [my_array]
L2:
    INC ECX
    CMP ECX, 10
    JZ over

    CMP EAX, [my_array + ECX*4]
    JGE L1
    MOV EAX, [my_array + ECX*4]
L1:
    JMP L2

over:
    PUSH EAX
    PUSH dword format
    CALL printf
    ADD ESP, 8


    RET
```

Name	Size	Attributes	Pointer to content
.data	43		0
.text	55		44

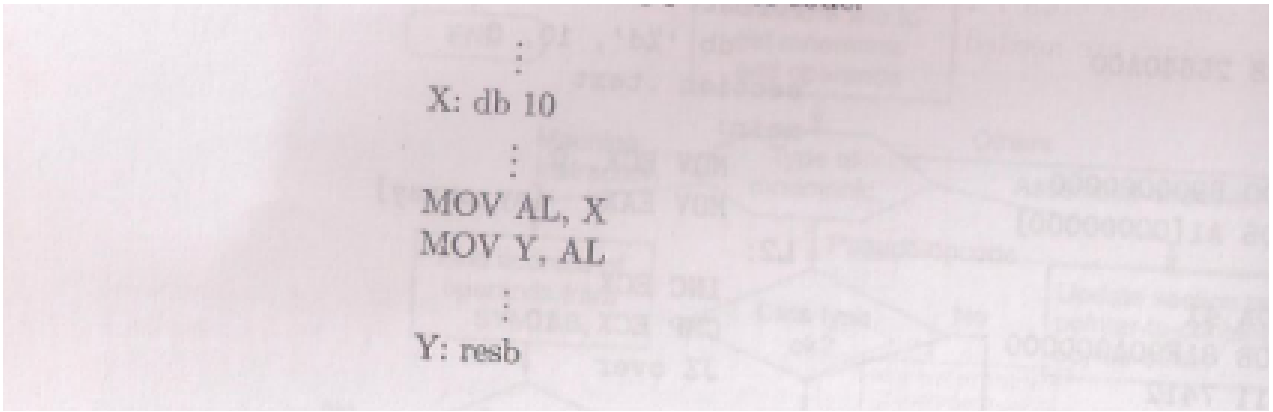
FIGURE 3.12 Updated *Section* table.

Name	Offsets to be corrected
printf	43

FIGURE 3.11 External reference list.

Single Pass Assembler

- Scans the input file only once.



- Problem of forward referencing.
- Once address of operands available afterwards, process of backpatching.

Single Pass Assembler

- Extra processing:
- On reaching EOF, check if forward referenced symbol left ‘undefined’. If yes issue error.
- If operand already present in table with ‘undefined’ type, label seared in the forward reference list and current location is added to list of locations requiring corrections.
- Whenever a new symbol defined in form of label to some instruction process of backpatching.

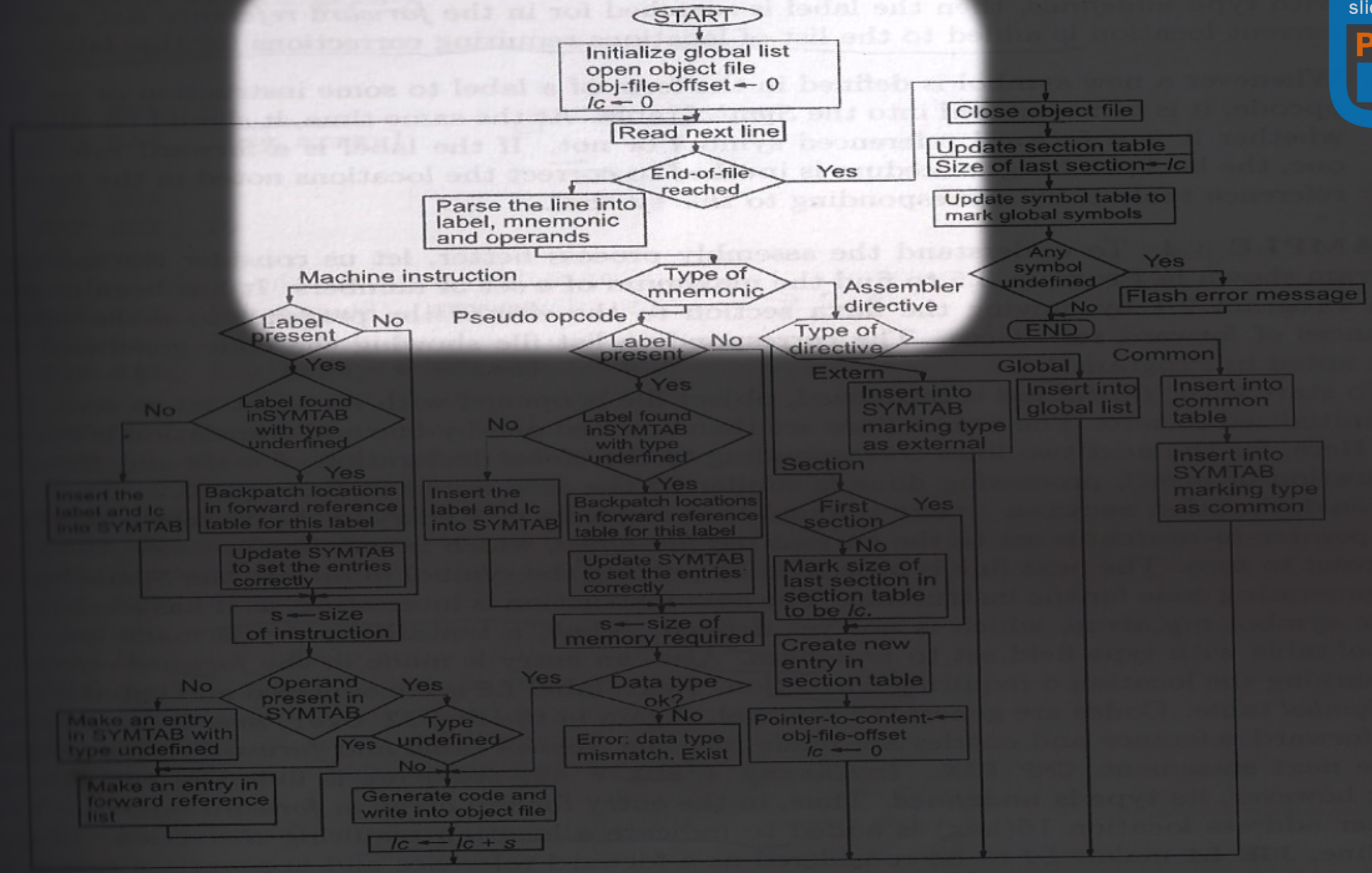


FIGURE 3.15 Single-pass assembler.

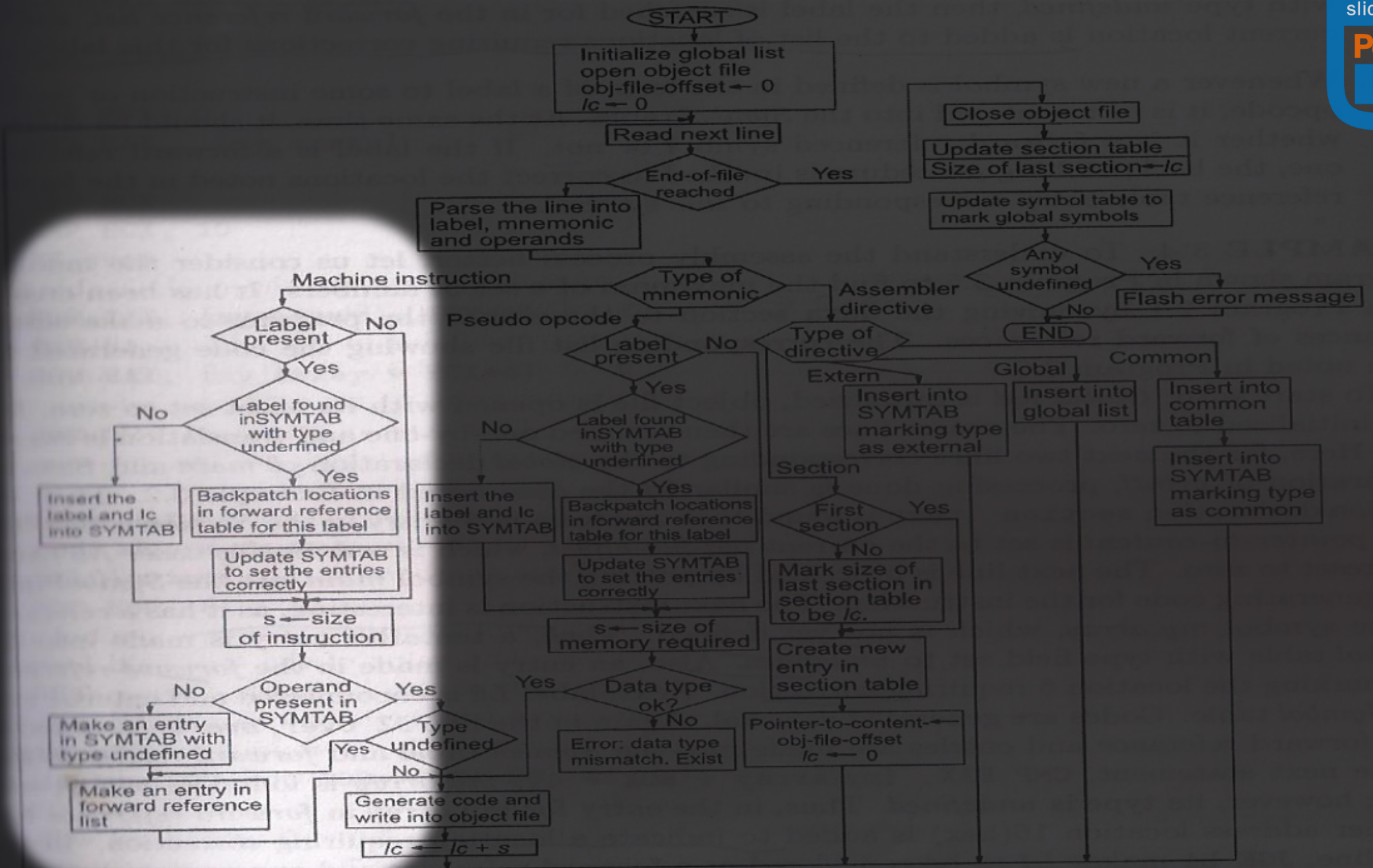


FIGURE 3.15 Single-pass assembler.

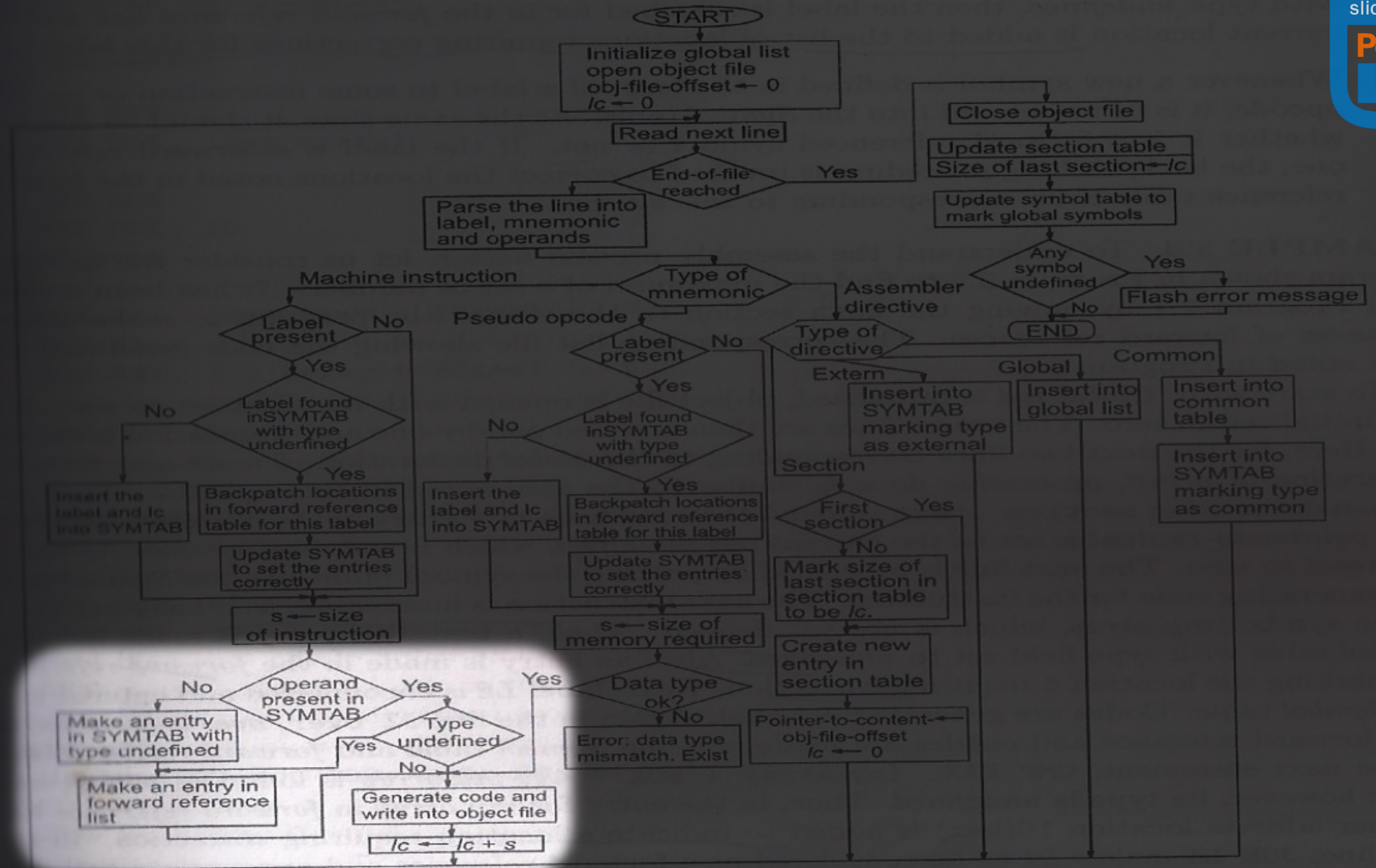


FIGURE 3.15 Single-pass assembler.

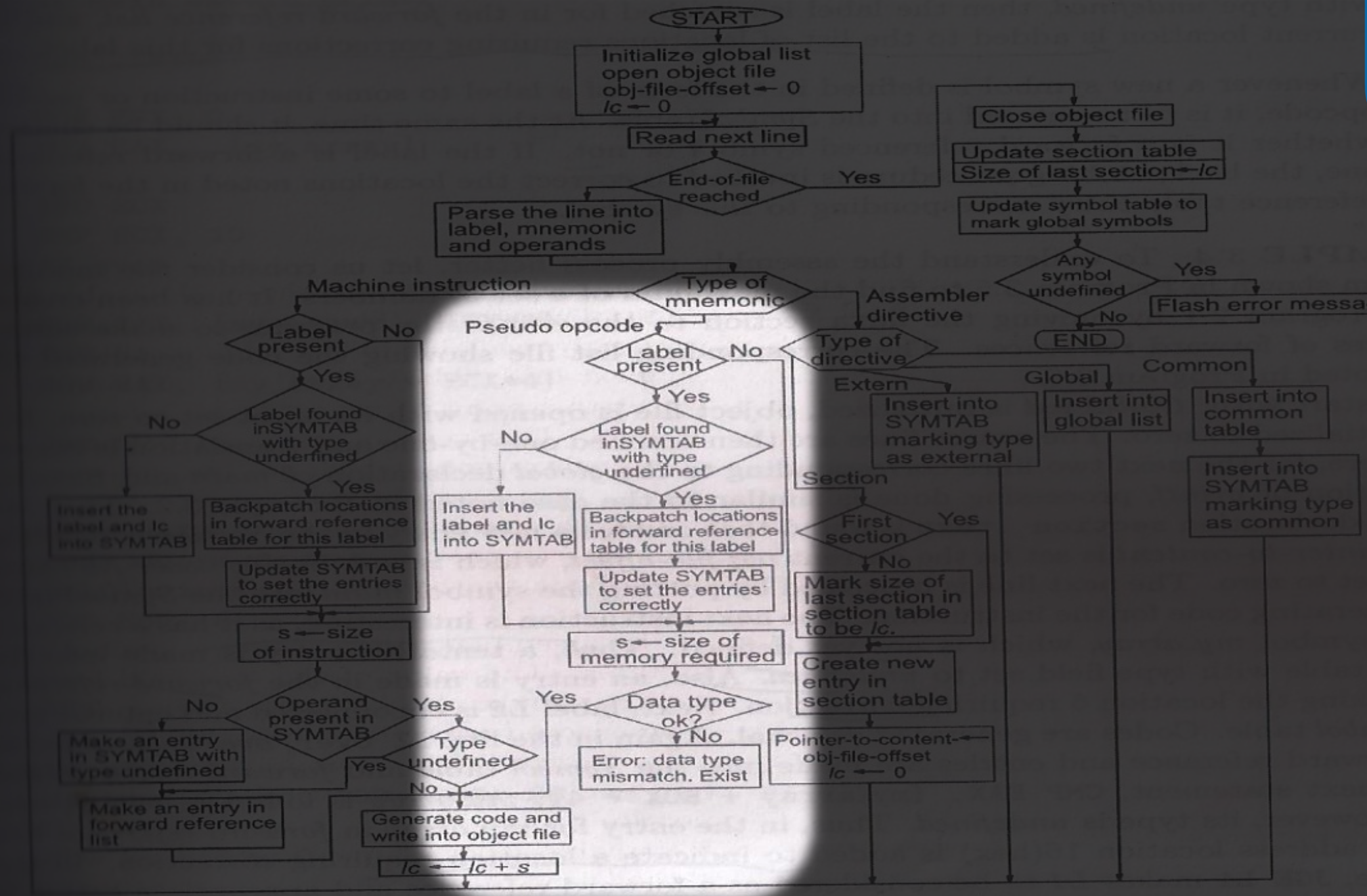


FIGURE 3.15 Single-pass assembler.

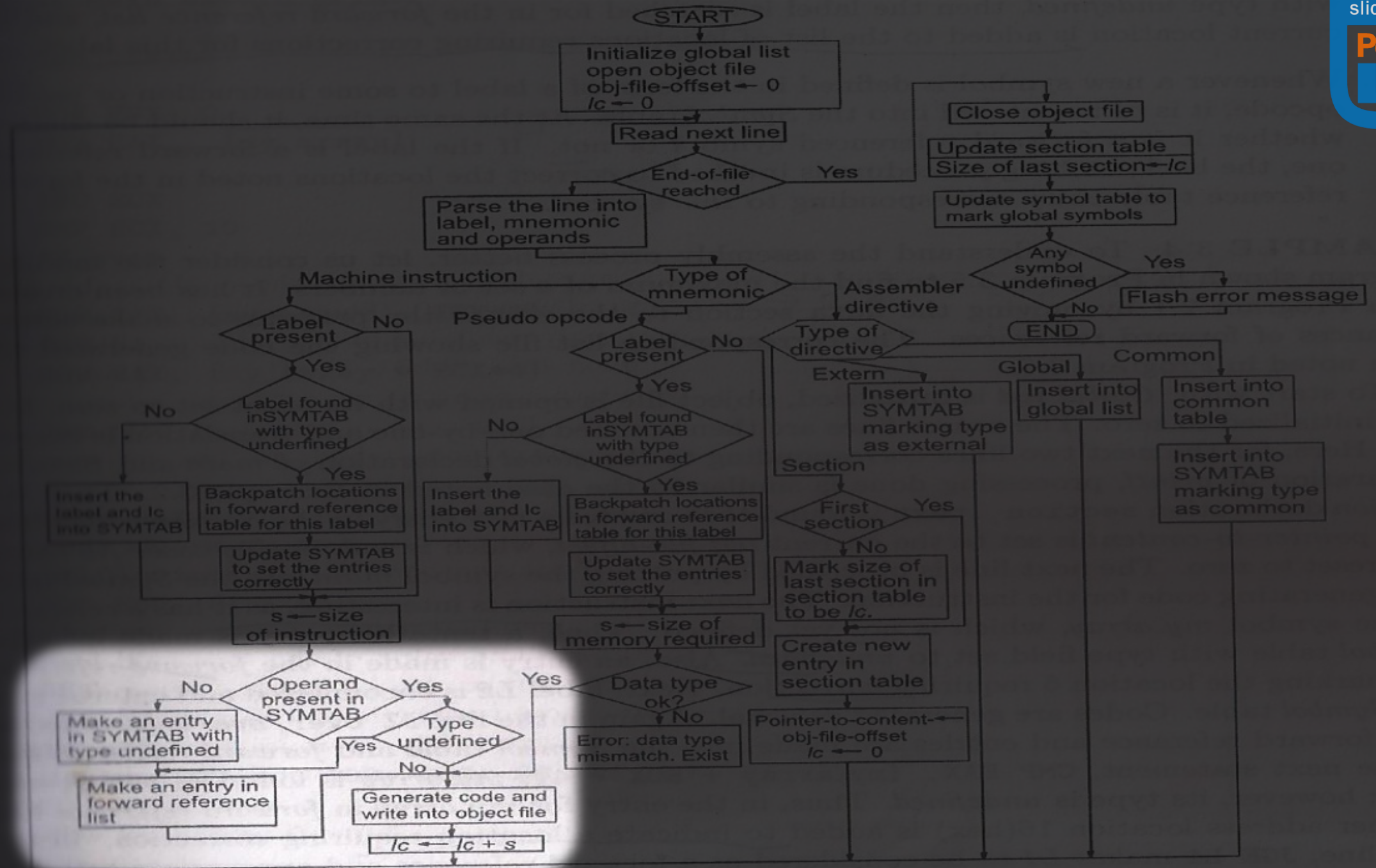


FIGURE 3.15 Single-pass assembler.

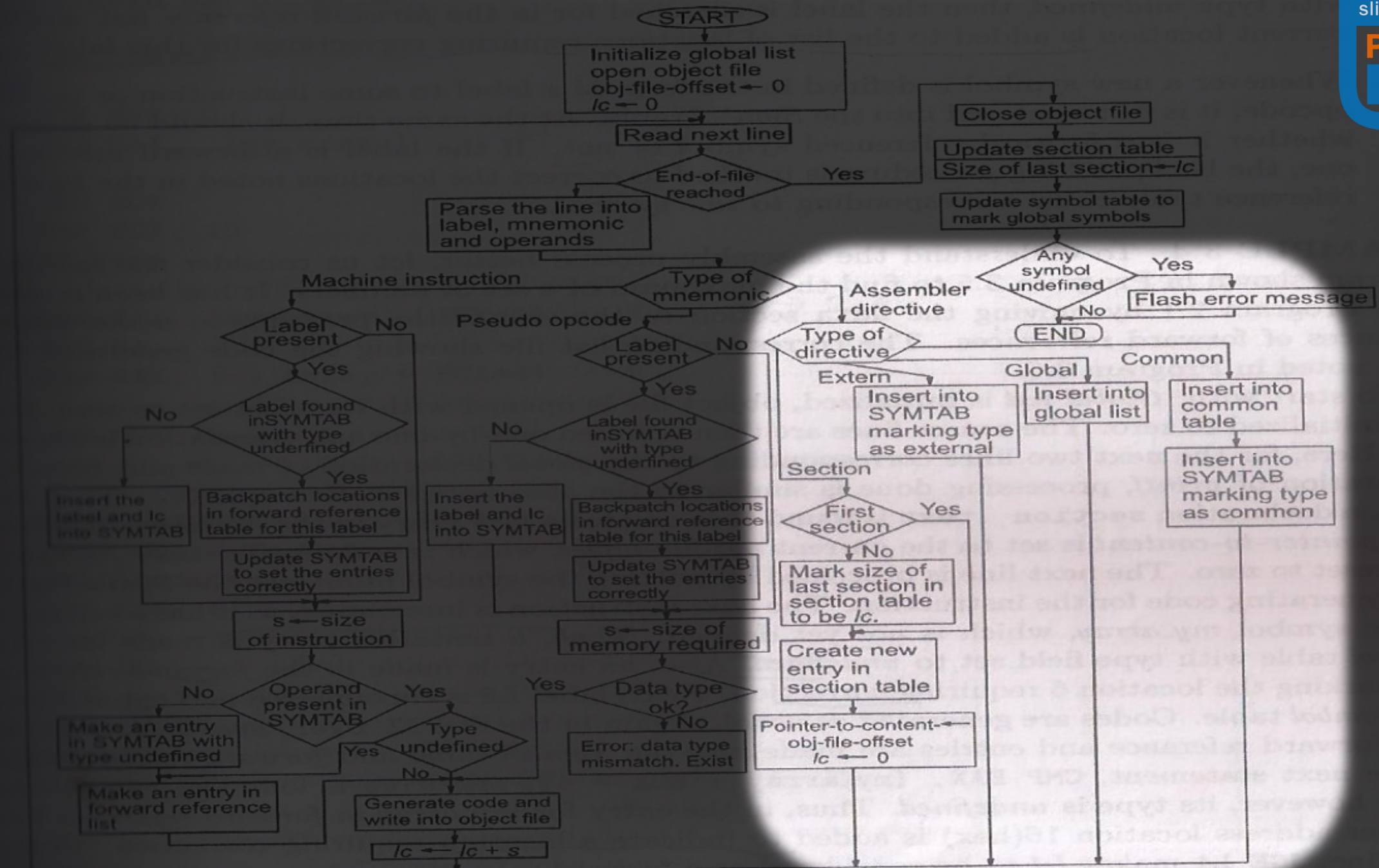


FIGURE 3.15 Single-pass assembler.

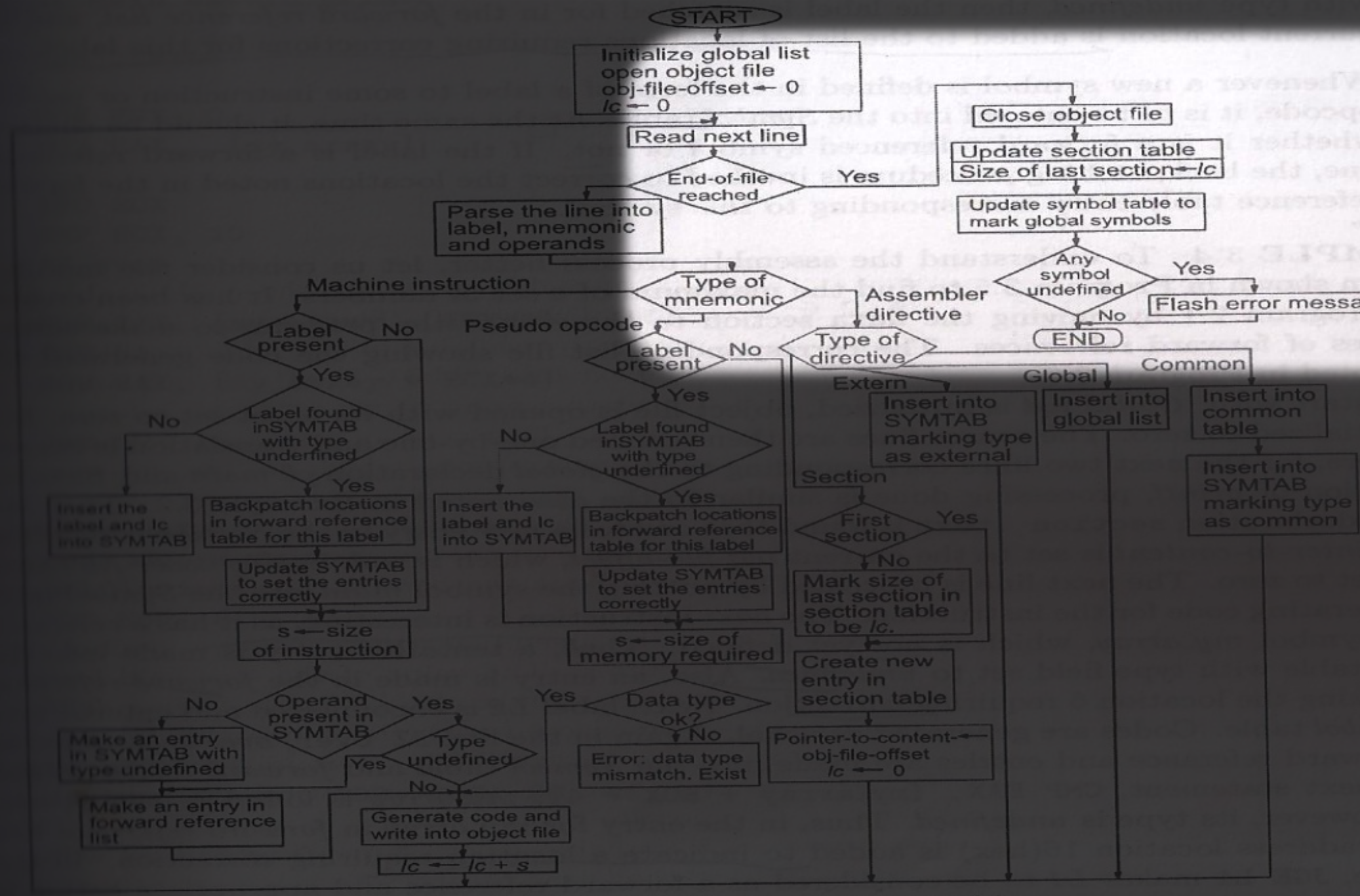


FIGURE 3.15 Single-pass assembler.

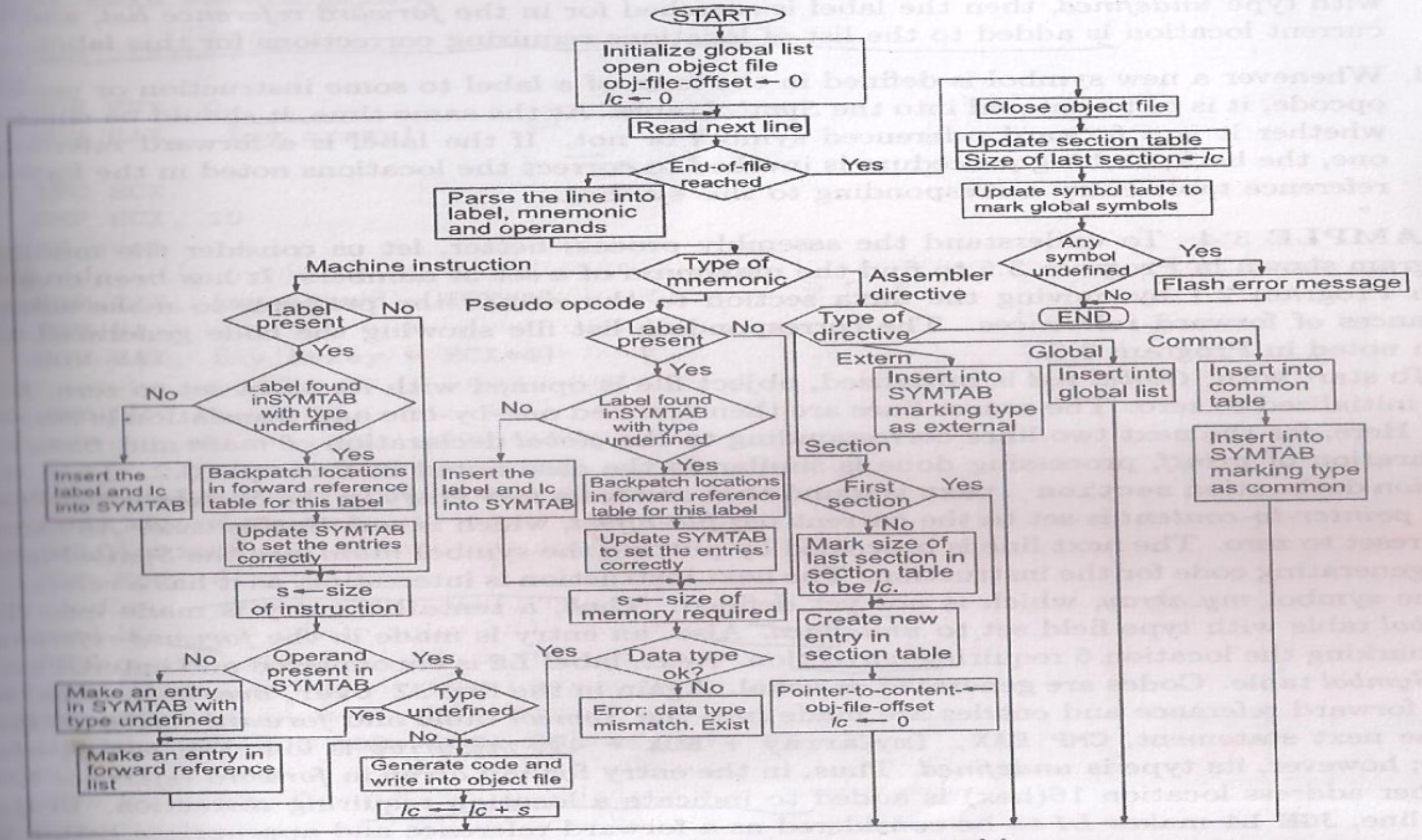


FIGURE 3.15 Single-pass assembler.

PROGRAM 3.6: List file showing code generated of the program to find the maximum

```
 1                                          global main
 2                                          extern printf
 3
 4                                          section .text
 5                                          main:
 6 00000000 B900000000                          MOV ECX, 0
 7 00000005 A1[00000000]                        MOV EAX,  [my_array]
 8                                          L2:
 9 0000000A 41                                  INC ECX
10 0000000B 81F90A000000                        CMP ECX, 10
11 00000011 7412                                JZ over
12
13 00000013 3B048D[00000000]                    CMP EAX, [my_array + ECX*4]
14 0000001A 7D07                                JGE L1
15 0000001C 8B048D[00000000]                    MOV EAX, [my_array + ECX*4]
16                                          L1:
17 00000023 EBE5                                JMP L2
18
19                                          over:
20 00000025 50                                  PUSH EAX
21 00000026 68[28000000]                        PUSH dword format
22 0000002B E8(00000000)                        CALL printf
23 00000030 81C408000000                        ADD ESP, 8
24
25
26 00000036 C3                                  RET
27                                          section .data
28                                          my_array:
29 00000000 0A000000140000001E-                 dd 10, 20, 30, 100, 200, 56, 45,
30 00000009 0000006400000C800-                    67, 89, 77
31 00000012 0000380000002D0000-
32 0000001B 0043000000590000000-
33 00000024 4D000000
34                                          format:
35 00000028 25640A00                            db '%d', 10, 0
36
```

Name	Type	Location	Size	Section-id	Is-global
printf	external				
main	label	0		1	-
my_array	undefined				
L2	label	10		1	-
over	undefined				
L1	undefined				

(a)

Name	Offsets(hex) to be corrected
my_array	06, 16, 1F
over	12
L1	1B

(b)

FIGURE 3.16 Partial (a) symbol table, and (b) forward reference list.

Name	Type	Location	Size	Section-id	Is-global
printf	external				
main	label	0		1	true
my_array	variable	0	40	2	false
L2	label	10		1	false
over	label	37		1	false
L1	label	35		1	false
format	variable	40	3	2	false

(a)

Name	Offsets(hex) to be corrected
my_array	06, 16, 1F
over	12
L1	1B
format	27

(b)

FIGURE 3.17 (a) Symbol table, and (b) Forward reference list.

Load and Go Assembler

- Assemblers produce code directly into memory and code is ready for execution is Load and go type.
- No loader needed
- For small programs favorable, where for each small modification, it is desirable to check the results.
- Time saved
- Drawbacks:
- User program needs to be resassembled each time it is run.

Object File Format

Intel hex format

Blocked object code

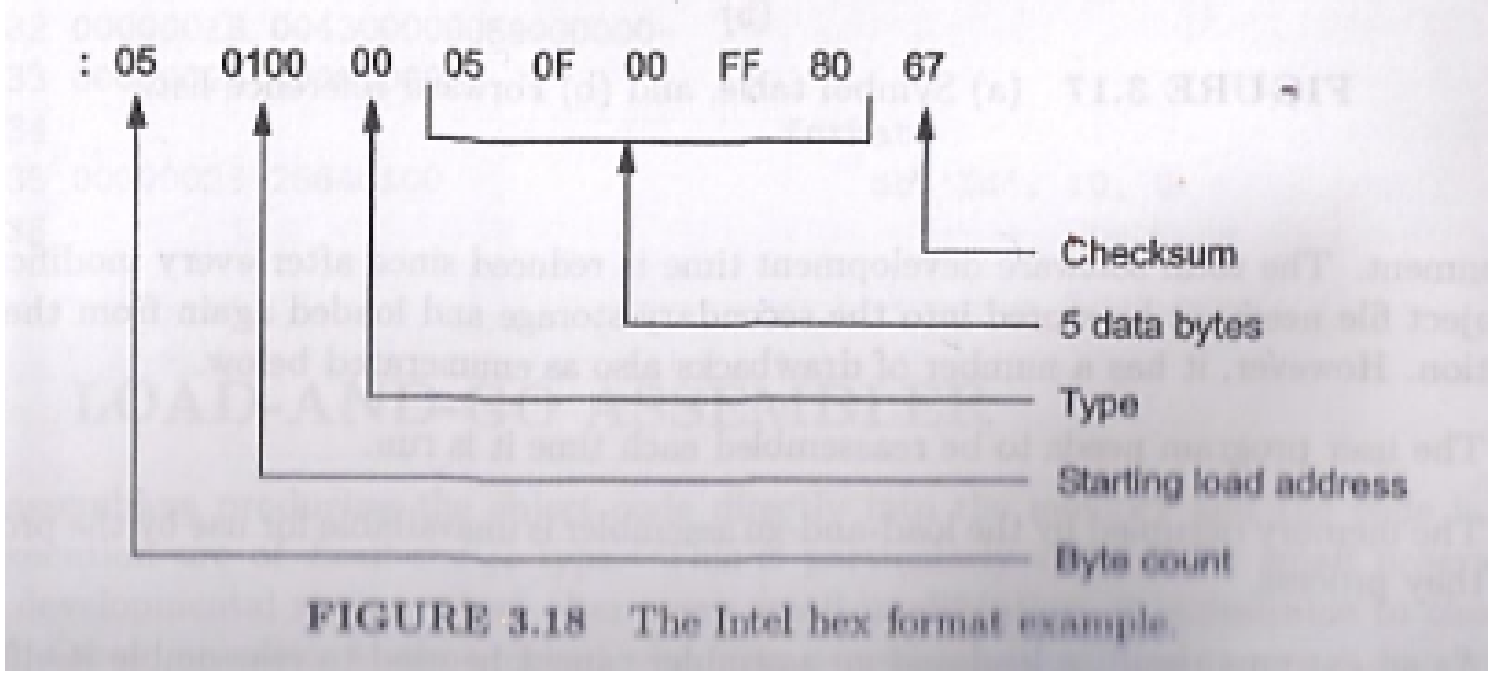


FIGURE 3.18 The Intel hex format example.

Object File Format

- Obj file: The obj format is the one used in assemblers MASM, TASM etc, typically fed to 16 bit DOS linkers to produce .EXE files. Default extension is .obj
- Win32 format: Used in Microsoft Win32 object files, suitable for passing Microsoft linkers such as Visual C++. Default extension is .obj
- Coff format: The coff object files are suitable for linking with DJGPP linker.
- Elf format: Used by Linux and Unix System V, including Solaris x86, UnixWare and SCO Unix. Default output file extension is .o. It allows programs to be viewed as collection of Sections

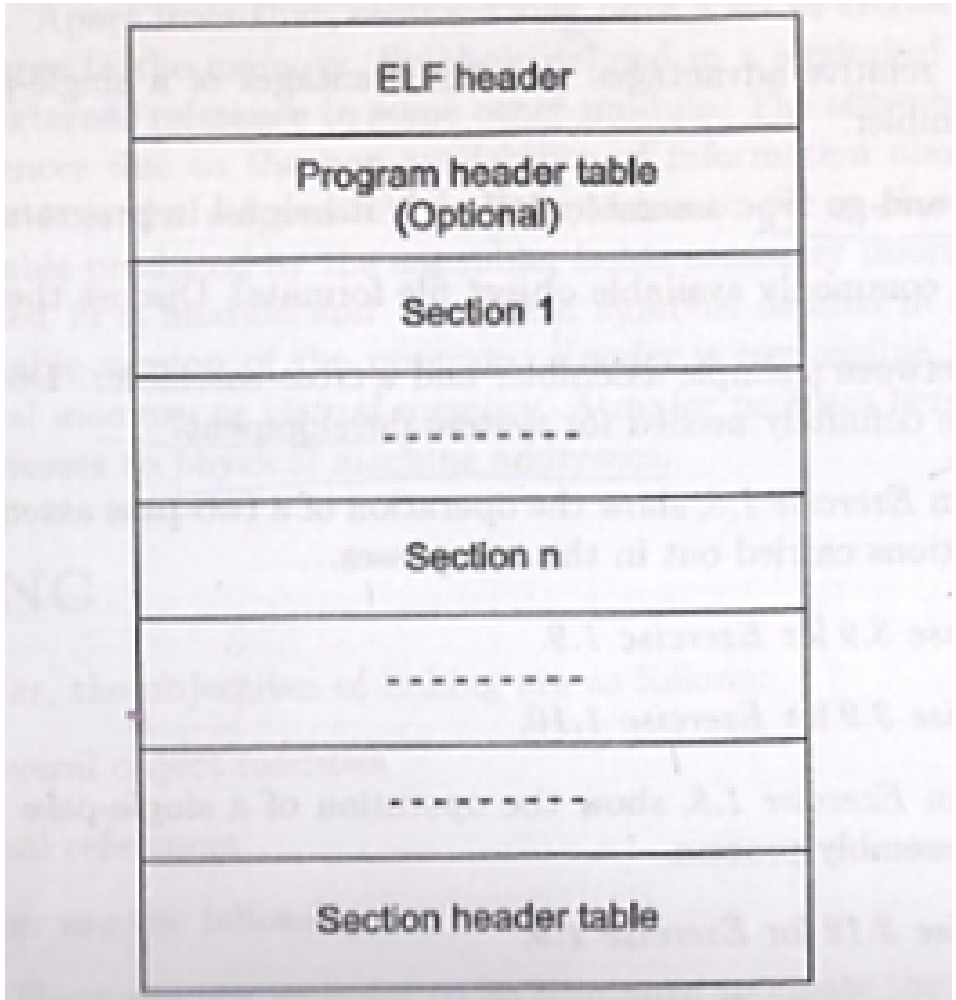


FIGURE 3.19 Elf object file format.

Aout Format: The aout format generates a.out object files used by early Linux systems.
Very Simple, does not contain any special directives or symbols