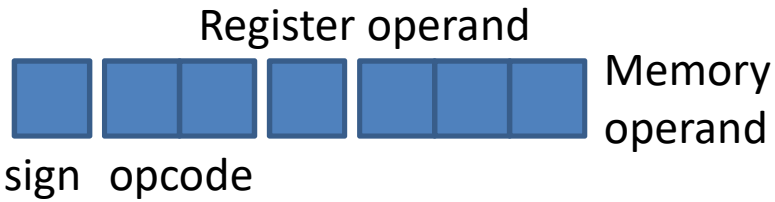
System Software

Dr Binu P Chacko
Associate Professor
Department of Computer Science
Prajyoti Niketan College, Pudukad

Assembly Language Programming

- Machine dependent, low level programming language
- Mnemonic operation codes
- Symbolic operands
- Data declarations



Instruction format

[Label] <opcode> <operand spec> [,<operand spec>…]

- <symbolic name>[±<displacement>][(<index register>)]
- E.g. AREA, AREA+5, AREA(4), AREA+5(4)
- Assembler: a language processor that converts an ALP into a MLP
- It *analyses* the source program, determine symbols corresponding to data or instructions, and use this information to *synthesize* a target program

Mnemonic Operation Codes

Instruction Opcode	assembly Mnemonic	Remarks
00	STOP	Stop execution
01	ADD	Perform addition
02	SUB	Perform subtraction
03	MULT	Perform multiplication
04	MOVER	Move from memory to register
05	MOVEM	Move from register to memory
06	COMP	Compare and set condition code
07	BC	Branch on condition
08	DIV	Perform division
09	READ	Read into register
10	PRINT	Print contents of register

Assembly Language Statements

Imperative statements: action to be performed during the execution of the program. It is translated into machine instruction

E.g. arithmetic operation

Declaration statements

- [Label] DS <constant>

A DS 1

B DS 100

- Different words in the block B is accessed through offsets from it
- [Label] DC ‘<value>’

ONE DC ‘1’

- Initializes a memory word with a value

Assembler directives: instruct the assembler to perform certain actions while assembling a program

START <constant>

END [<operand specification>]

Cont…

ORIGIN <address specification>

- <operand specification> or <constant>
- LC affected
- ORIGIN LOOP+2

<symbol> EQU <address specification>

- <constant> or <symbolic name> $\pm$ <displacement>
- Not affected the LC
- BACK EQU LOOP

LTORG

- Where to place the literals, forward reference, LC affected
- Literal is entered into the literal pool whenever it appears in a statement. At every LTORG statement or END statement, memory is allocated to literals in the literal pool

Assembly Process

```
         START   200
         MOVER   AREG, ='5'        200)  +04  1  211
         MOVEM   AREG, A           201)  +05  1  217
LOOP     MOVER   AREG, A           202)  +04  1  217
         MOVER   CREG, B           203)  +05  3  218
         ADD     CREG, ='1'        204)  +01  3  212
         ........
         BC      ANY, NEXT         210)  +07  6  214
         LTORG
                 ='5'              211)  +00  0  005
                 ='1'              212)  +00  0  001
         .......
NEXT     SUB     AREG, ='1'        214)  +02  1  219
         BC      LT, BACK          215)  +07  1  202
LAST     STOP                      216)  +00  0  000
         ORIGIN  LOOP+2
         MULT    CREG, B           204)  +03  3  218
         ORIGIN  LAST+1
A        DS      1                 217)
BACK     EQU     LOOP
B        DS      1                 218)
         END
                 ='1'              219)  +00  0  001
```

Assembly Scheme

```
         START    101
         READ     N                  101) + 09 0 113
         MOVER    BREG, ONE          102) + 04 2 115
         MOVEM    BREG, TERM         103) + 05 2 116
AGAIN    MULT     BREG, TERM         104) + 03 2 116
         MOVER    CREG, TERM         105) + 04 3 116
         ADD      CREG, ONE          106) + 01 3 115
         MOVEM    CREG, TERM         107) + 05 3 116
         COMP     CREG, N            108) + 06 3 113
         BC       LE,   AGAIN        109) + 07 2 104
         MOVEM    BREG, RESULT       110) + 05 2 114
         PRINT    RESULT             111) + 10 0 114
         STOP                        112) + 00 0 000
N        DS       1                  113)
RESULT   DS       1                  114)
ONE      DC       '1'                115) + 00 0 001
TERM     DS       1                  116)
         END
```

ALP is easy to modify

Cont…

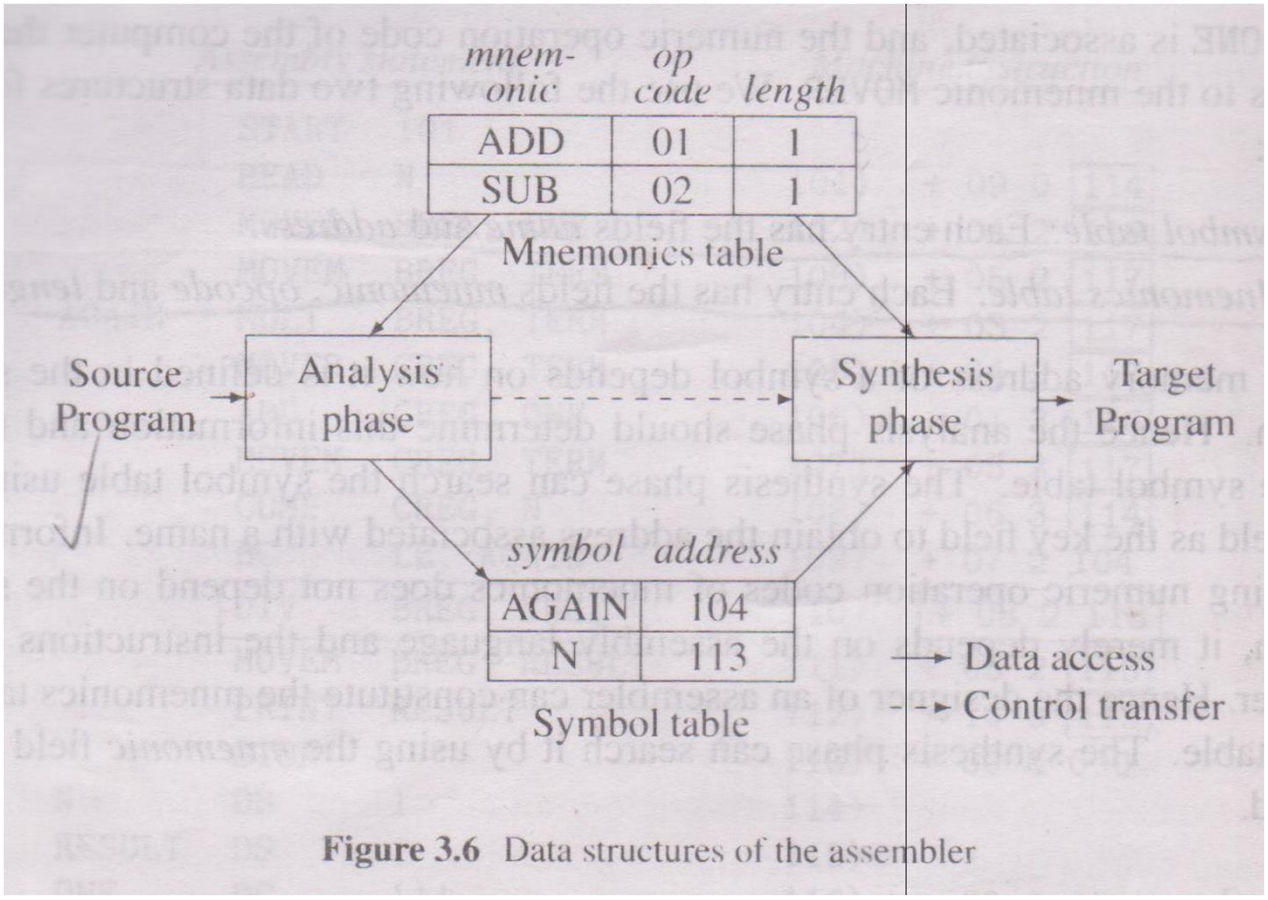


Figure 3.6 Data structures of the assembler

Cont…

Analysis phase

- Build symbol table
- Memory allocation using LC (contains address of the next memory word in the target code)
- LC is initialized with a constant in START
- Separate the contents of label, mnemonic opcode and operand fields
- If a symbol is present in the label field, enter the pair (symbol, <LC>) in the Symbol table
- Check the validity of the mnemonic opcode through a look-up in the Mnemonics table
- Perform LC processing

Synthesis phase

- Obtain the machine opcode from the Mnemonics table
- Obtain the address of memory operand from Symbol table
- Synthesis the machine instruction

Pass Structure of Assembler

- A language processor may make multiple passes over a source program for handling forward references
- Use of a symbol that precedes its definition in a program

Two-pass translation

- First pass (analysis of source program): constructs IR (data structures, IC) of the source program. It performs LC processing and records address of symbols in symbol table, separate the symbol, mnemonic opcode and operand fields
- Second pass: synthesises the target program using IR

Single-pass translation

- LC processing, construction of symbol table and target program
- The problem of forward reference is handled using backpatching
- Assembler builds TII (instruction address, symbol)
- After processing END, symbol table would contain address of all symbols
- Process each entry in TII to complete the concerned instruction

Cont…

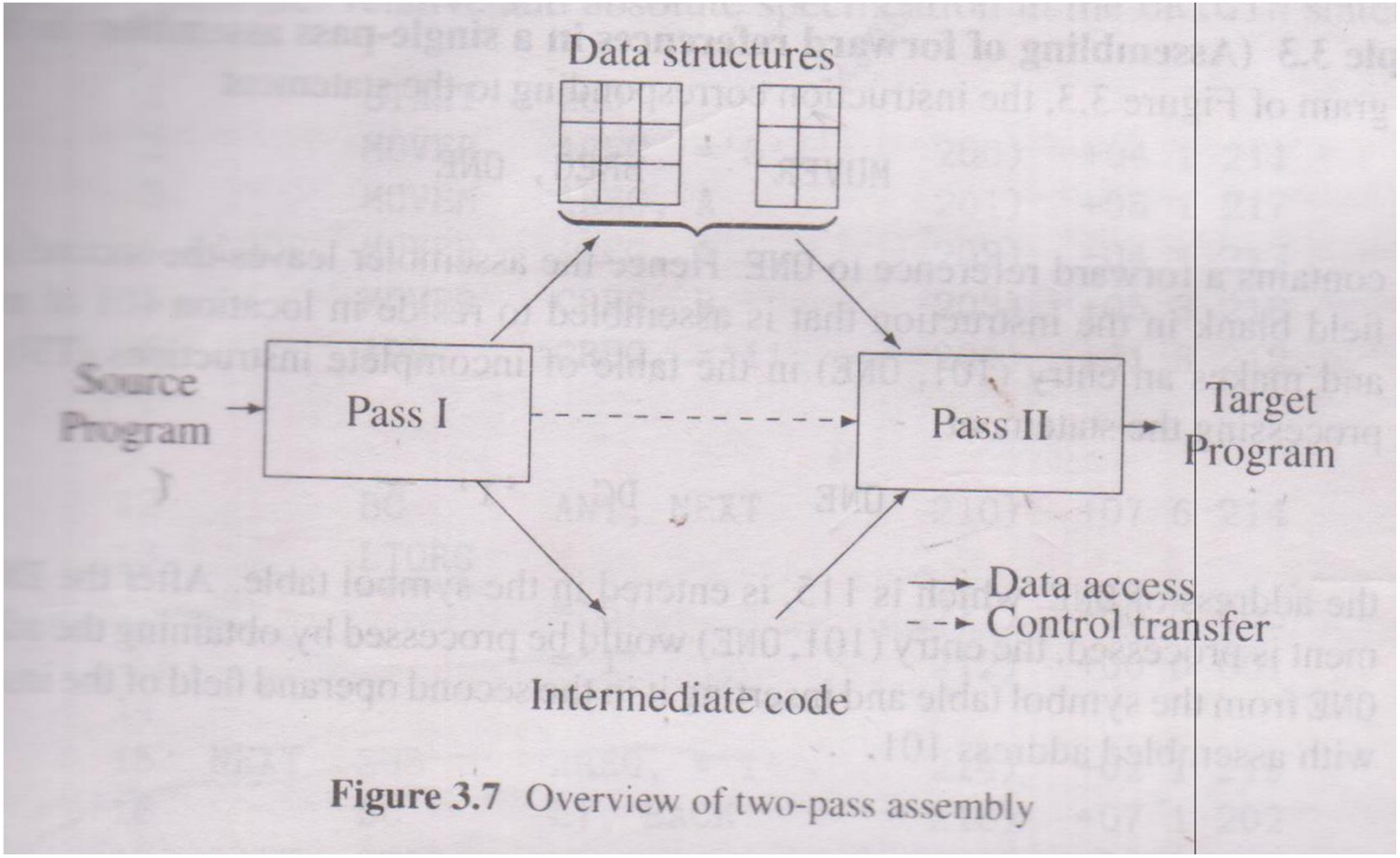


Figure 3.7 Overview of two-pass assembly

Pass I

Data structures

Mnemonic Opcode	Class	Mnemonic Info
MOVER	IS	(04, 1)
DS	DL	R#7
START	AD	R#11

OPTAB

Symbol	Address	Length
LOOP	202	1
NEXT	214	1
LAST	216	1
A	217	1
BACK	202	1
B	218	1

SYMTAB

Assembler directives	
START	01
END	02
ORIGIN	03
EQU	04
LTORG	05

	Value	Address
1	='5'	
2	='1'	
3	='1'	

LITTAB

First	#literals
1	2
3	1
4	0

POOLTAB

Declarative statements	
DC	01
DS	02

OPTAB, SYMTAB, LITTAB and POOLTAB

LC	:	Location counter
littab_ptr	:	Points to an entry in LITTAB
pooltab_ptr	:	Points to an entry in POOLTAB

Details of the intermediate code generated by Pass I are discussed in the next section.

Algorithm 3.1 (Pass I of a two-pass assembler)

1. LC := 0; (This is the default value)
   littab_ptr := 1;
   pooltab_ptr := 1;
   POOLTAB [1].*first* := 1; POOLTAB [1].*# literals* := 0;
2. While the next statement is not an END statement
   (a) If a symbol is present in the label field then
   this_label := symbol in the label field;
   Make an entry (*this_label*, <LC>, –) in SYMTAB.
   (b) If an LTORG statement then
   (i) If POOLTAB [*pooltab_ptr*].*# literals* > 0 then
   Process the entries LITTAB [POOLTAB [*pooltab_ptr*].*first*] ... LITTAB [*littab_ptr*−1] to allocate memory to the literal, put address of the allocated memory area in the *address* field of the LITTAB entry, and update the address contained in location counter accordingly.
   (ii) *pooltab_ptr* := *pooltab_ptr* + 1;
   (iii) POOLTAB [*pooltab_ptr*].*first* := *littab_ptr*;
   POOLTAB [*pooltab_ptr*].*# literals* := 0;

(c) If a START or ORIGIN statement then
LC := value specified in operand field;

(d) If an EQU statement then
(i) *this_addr* := value of <*address specification*>;
(ii) Correct the SYMTAB entry for *this_label* to (*this_label, this_addr*, 1).

(e) If a declaration statement then
(i) Invoke the routine whose id is mentioned in the *mnemonic info* field. This routine returns *code* and *size*.
(ii) If a symbol is present in the label field, correct the symtab entry for *this_label* to (*this_label*, <LC>, *size*).
(iii) LC := LC + *size*;
(iv) Generate intermediate code for the declaration statement.

(f) If an imperative statement then
(i) *code* := machine opcode from the *mnemonic info* field of OPTAB;
(ii) LC := LC + instruction length from the *mnemonic info* field of OPTAB;
(iii) If operand is a literal then
this_literal := literal in operand field;
if POOLTAB [*pooltab_ptr*]. *# literals* = 0 or *this_literal* does not match any literal in the range LITTAB [POOLTAB [*pooltab_ptr*]]. *first* ... LITTAB [*littab_ptr*−1] then
LITTAB [*littab_ptr*]. *value* := *this_literal*;
POOLTAB [*pooltab_ptr*]. *# literals* :=
POOLTAB [*pooltab_ptr*]. *# literals* +1;
littab_ptr := *littab_ptr* + 1;
else (i.e., operand is a symbol)
this_entry := SYMTAB entry number of operand;
Generate intermediate code for the imperative statement.

3. (Processing of the `END` statement)
(a) Perform actions (i)–(iii) of Step 2(b).
(b) Generate intermediate code for the `END` statement.

Intermediate Code

Address	Mnemonic opcode	operands

- Mnemonic opcode (statement class, code)
- Statement class: IS, DL, AD
- Code: machine opcode for IS, ordinal number within the class for AD/DL

Variant I

- First operand: register (1..4) or condition code (1..6)
- Second operand (operand class, code)
- Operand class: C, S, L

Cont…

	START 200		(AD, 01)	(C, 200)
	READ	A	(IS, 09)	(S, 01)
LOOP	MOVER	AREG, A	(IS, 04)	(1) (S, 01)
	……..			
	SUB	AREG, =‘1’	(IS, 01)	(1) (L, 01)
	BC	GT, LOOP	(IS, 07)	(4) (S, 02)
	STOP		(IS, 00)	
A	DS	1	(DL, 02)	(C, 1)

- A series of (DL, 01) are required for

C	DC	‘5, 3, -7’

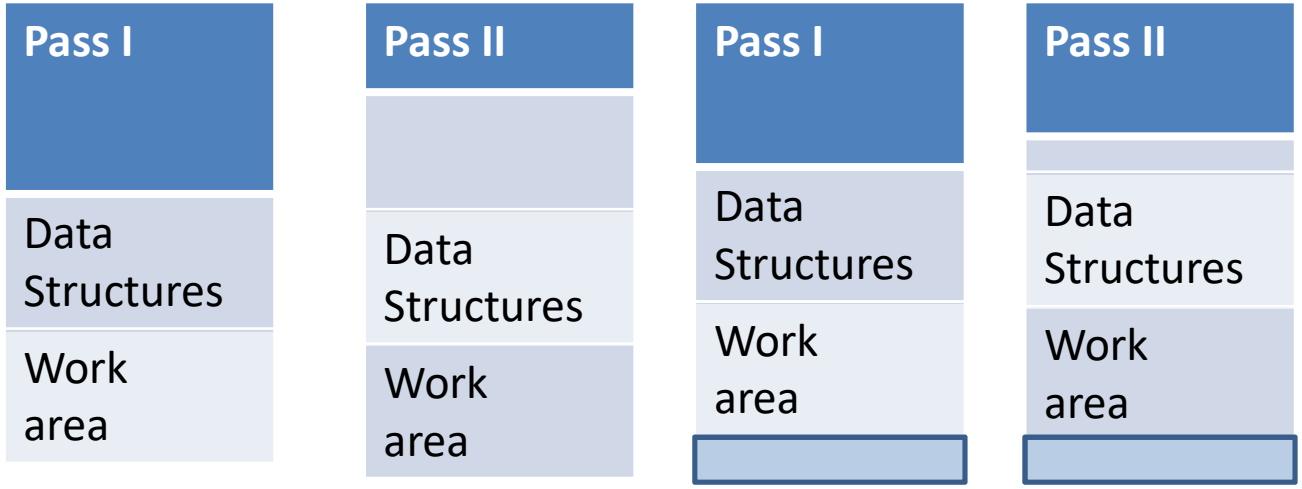
Variant II

- AD/DL statements are processed
- IS: mnemonics and literals are processed

	START	200	(AD, 01)	(C, 200)
	READ	A	(IS, 09)	A
LOOP	MOVER	AREG, A	(IS, 04)	AREG, A
	……..			
	SUB	AREG, =‘1’	(IS, 02)	AREG, (L, 01)
	BC	GT, LOOP	(IS, 07)	GT, LOOP
	STOP		(IS, 00)	
A	DS	1	(DL, 02)	(C, 1)

Comparison

Variant I	Variant II
Do extra work to completely process the operand fields	Reduces the work of Pass I
Simplify the tasks of Pass II	Well-suited if expressions are used in operand fields
IC is compact	IC is less compact

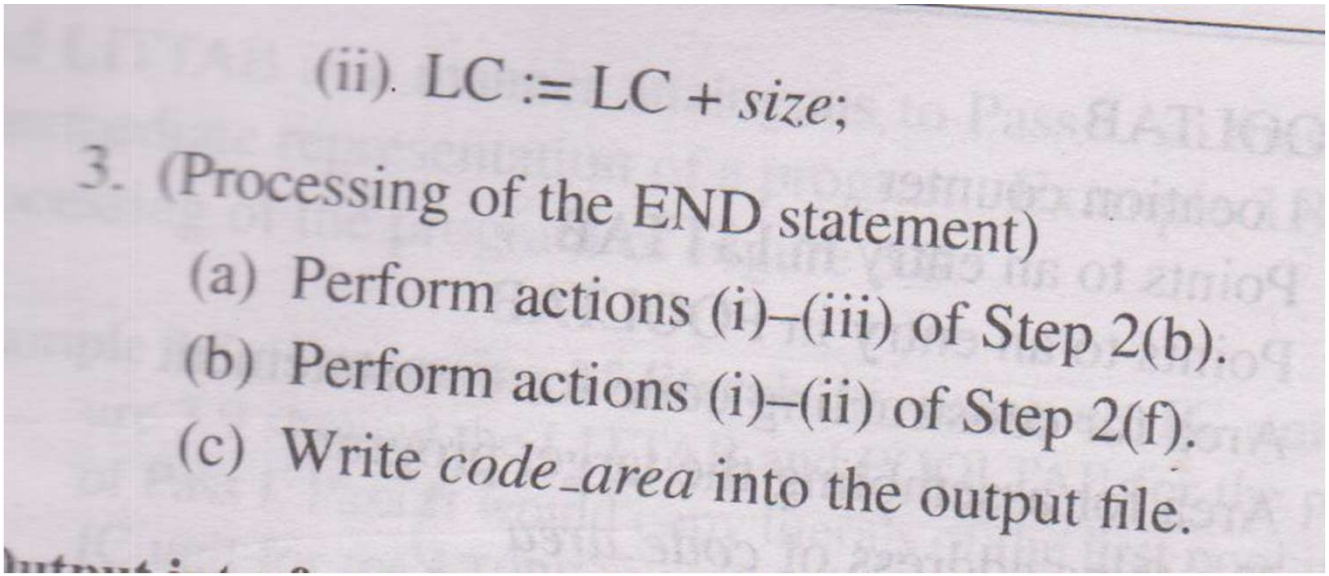


SYMTAB, LITTAB and POOLTAB

LC	:	Location counter
littab_ptr	:	Points to an entry in LITTAB
pooltab_ptr	:	Points to an entry in POOLTAB
machine_code_buffer	:	Area for constructing code for one statement
code_area	:	Area for assembling the target program
code_area_address	:	Contains address of *code_area*

Algorithm 3.2 (Second pass of a two-pass assembler)

1. *code_area_address* := address of *code_area*;
   pooltab_ptr := 1;
   LC := 0;
2. While the next statement is not an END statement
   (a) Clear *machine_code_buffer*;
   (b) If an LTORG statement
      (i) If POOLTAB [*pooltab_ptr*]. # *literals* > 0 then
         Process literals in the entries LITTAB [POOLTAB [*pooltab_ptr*] .*first*] ... LITTAB [POOLTAB [*pooltab_ptr*+1]−1] similar to processing of constants in a DC statement. It results in assembling the literals in *machine_code_buffer*.
      (ii) *size* := size of memory area required for literals;
      (iii) *pooltab_ptr* := *pooltab_ptr* + 1;
   (c) If a START or ORIGIN statement
      (i) LC := value specified in operand field;
      (ii) *size* := 0;
   (d) If a declaration statement
      (i) If a DC statement then
         Assemble the constant in *machine_code_buffer*.
      (ii) *size* := size of the memory area required by the declaration statement;
   (e) If an imperative statement
      (i) Get address of the operand from its entry in SYMTAB or LITTAB, as the case may be.
      (ii) Assemble the instruction in *machine_code_buffer*.
      (iii) *size* := size of the instruction;
   (f) If *size* ≠ 0 then
      (i) Move contents of *machine_code_buffer* to the memory word with the address *code_area_address* + <LC>;


(ii). LC := LC + *size*;

3. (Processing of the END statement)
   (a) Perform actions (i)–(iii) of Step 2(b).
   (b) Perform actions (i)–(ii) of Step 2(f).
   (c) Write *code_area* into the output file.

Error Reporting

- In Pass I: source program need not be preserved until Pass II
- Conserves memory and avoids some amount of duplicate processing
- Only certain errors are reported against the source statement
- Target code could not be included in the listing,
instead memory address of the code is printed
- In Pass II: report each error against the erroneous statement
- Target code against the source statement

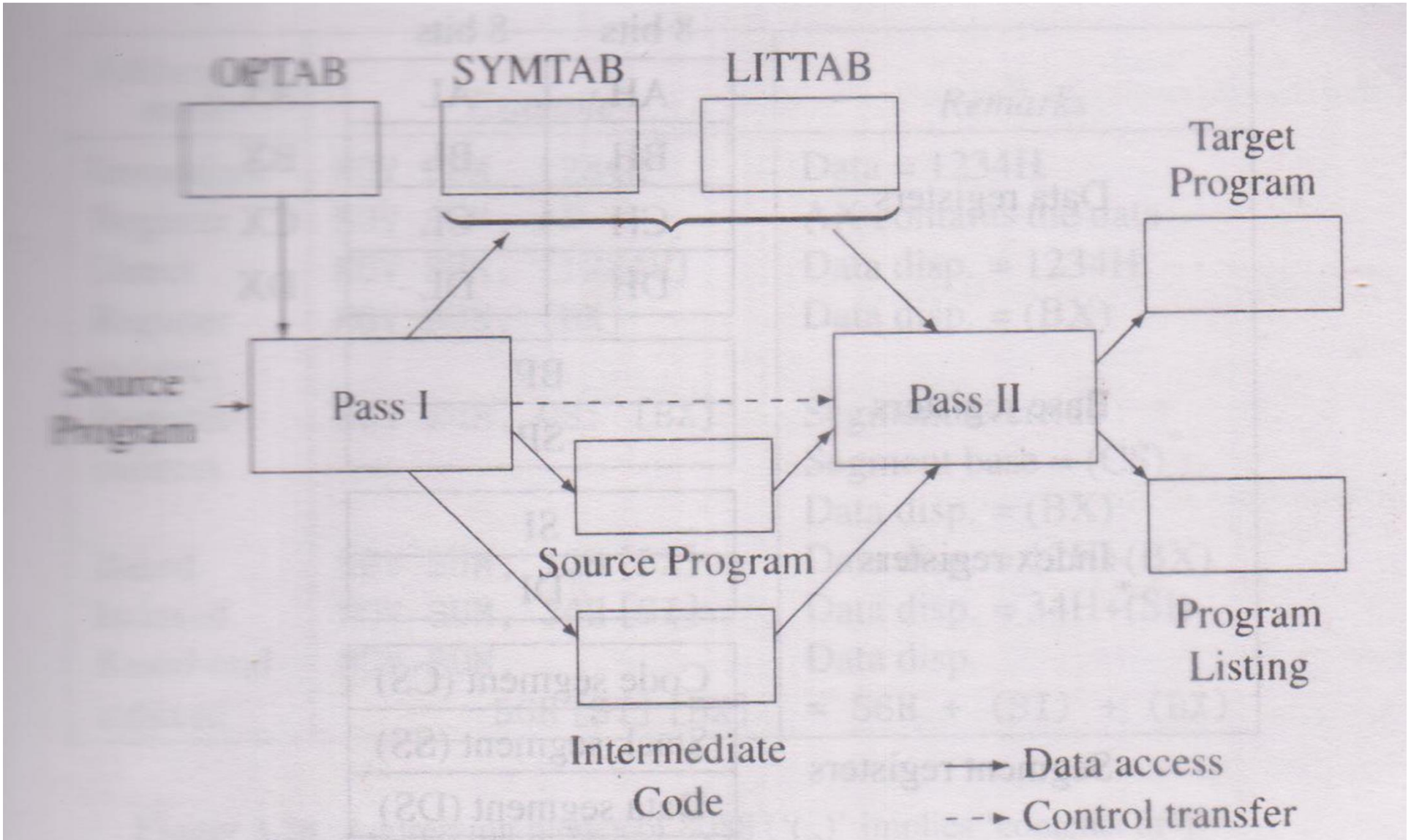


Figure 3.18 Use of data structures and files in a two-pass assembler

Macros

- A facility for extending the programming language
- Macro definition: defines either a new operation or a new method of declaring data
- Macro call: an invocation of the new operation or the new method of declaring data defined in the macro
- Macro expansion: a program generation activity during which macro call is replaced by a sequence of statements
- Lexical substitution: replacement of a character string by another during program generation. To replace formal parameter by actual parameter
- Semantic expansion: generation of statements depending on the requirements of a macro call. Expansion of calls may lead to codes which differ in number, sequence and opcodes of statements

Macro Definition

- Placed at the beginning of the program
- Macro prototype statement declares the name of macro and names and kinds of its formal parameters
- <macro name> [<formal parameter specification> [,…]]
- &<parameter name>[<parameter kind>]
- Model statement is a statement from which an assembly language statement may be generated during macro expansion
- Macro preprocessor statement is used to perform auxiliary functions during macro expansion

Macro Call

- <macro call> [<actual parameter specification> [,…]]
- INCR A, B, AREG

```
MACRO
INCR        &MEM_VAL, &INCR_VAL, &REG
MOVER       &REG, &MEM_VAL
ADD         &REG, &INCR_VAL
MOVEM       &REG, &MEM_VAL
MEND
+ MOVER     AREG, A
+ ADD       AREG, B
+ MOVEM     AREG, A
```

Macro Expansion

- MEC: to implement the flow of control during macro expansion. It points to a model statement that is to be expanded next

Algorithm

1. MEC:= statement number of the first model statement following the prototype statement in the definition of the called macro
2. While the next statement pointed to by MEC is not a MEND statement

   (a) If model statement then

   (i) Expand the statement through lexical substitution

   (ii) MEC:=MEC+1

   (b) Else (i.e., the statement is a preprocessor statement)

   MEC:=value specified in the preprocessor statement
3. Exit from macro expansion

Cont…

- Lexical substitution: an ordinary string is kept as it is. Name of a formal parameter or preprocessor variable (both preceded by &) is replaced by its value
- Positional parameter: &<parameter name>
- Keyword parameter
- INCR_M &MEM_VAL=, &INCR_VAL=, &REG=
- <formal parameter name>=<ordinary string>
- INCR_M MEM_VAL=A, INCR_VAL=B, REG=AREG
- INCR-M INCR_VAL=B, REG=AREG, MEM_VAL=A

Cont…

- INCR_D &MEM_VAL=, &INCR_VAL=, &REG=AREG
- &<parameter name> [<parameter kind>[<default value>]]
- INCR_D MEM_VAL=A, INCR_VAL=B
- INCR_D INCR_VAL=B, MEM_VAL=A
- INCR_D INCR_VAL=B, MEM_VAL=A, REG=BREG
- SUMUP A, B, G=20, H=X

```
        MACRO
        CALC    &X, &Y, &OP=MULT, &LAB=
&LAB    MOVER   AREG, &X
        &OP     AREG, &Y
        MOVEM   AREG, &X
        MEND
CALC    A, B, LAB=LOOP
+   LOOP        MOVER   AREG, A
+               MULT    AREG, B
+               MOVEM   AREG, A
```

Nested Macro Call

- A model statement in a macro may constitute a call on another macro
- COMPUTE X, Y

```
MACRO
COMPUTE     &FIRST, &SECOND
MOVEM       BREG, TMP
INCR_D      &FIRST, &SECOND, &REG=BREG
MOVER       BREG, TMP
MEND
+ MOVEM     BREG, TMP
+ INCR      X, Y
+ MOVER     BREG, TMP
```

```
+ MOVER     BREG, X
+ ADD       BREG, Y
+ MOVEM     BREG, X
```

- Expansion of nested macro call is on LIFO basis

Advanced Macros

Altering expansion time control flow

- AIF (<expression>) <sequencing symbol>

.<ordinary string>

- AGO <sequencing symbol>
- <sequencing symbol> ANOP

```
        MACRO
        CREATE_CONST &X, &Y
        AIF             (T'&X  EQ  B)     .BYTE
&Y      DW              25
        AGO             .OVER
.BYTE   ANOP
&Y      DB              25
.OVER   MEND
```

Cont…

Attributes of formal parameters

- <attribute name>’<formal parameter spec>
- T, L, S

Expansion time variable

- LCL <EV specification>[, <EV specification>….]
- GBL <EV specification>[, <EV specification>….]

&<EV name>

- <EV specification> SET <SET-expression>
- CONSTANTS

```
MACRO
CONSTANTS
        LCL     &A
&A      SET     1
        DB      &A
&A      SET     &A+1
        DB      &A
        MEND
```

Conditional Expansion

- A-B+C

```
                MACRO
                EVAL        &X, &Y, &Z
                AIF         (&X EQ &Y)  .ONLYMOVER
                MOVER       AREG, &X
                SUB         AREG, &Y
                ADD         AREG, &Z
                AGO         .OVER
.ONLYMOVER      MOVER       AREG, &Z
.OVER           MEND
```

- EVAL A, B, C

Expansion Time Loop

```
        MACRO
        CLEAR   &X, &N
        LCL     &M
&M      SET     0
        MOVER   AREG, =‘0’
.MORE   MOVEM   AREG, &X+&M
&M      SET     &M+1
        AIF     (&M NE &N) .MORE
        MEND
```

```
•   CLEAR       B, 3
+   MOVER       AREG, =‘0’
+   MOVEM       AREG, B
+   MOVEM       AREG, B+1
+   MOVEM       AREG, B+2
```

Cont…

```
        MACRO
        CONST10
        LCL         &M
&M      SET     1
        REPT    10
        DC      '&M'
&M      SET     &M+1
        ENDM
        MEND
```

- IRP <formal parameter>, <argument list>

```
MACRO
CONSTS  &M, &N, &Z
IRP     &Z, &M, 7, &N
DC      '&Z'
ENDM
MEND
```

- CONSTS 4, 10

Macro Preprocessor

Tasks in macro expansion

- Recognizing a macro call
- Compare the string found in mnemonic field with name of all macros defined in a program
- MNT stores the name of all defined macros
- Determine values of formal parameters
- PDT – (<formal parameter name>, <default value>)
- APT – (<formal parameter name>, <value>)
- Maintain values of expansion time variables
- EVT – (<expansion time variable name>, <value>)

Cont…

- Organising expansion time control flow
- MDT stores macro definition
- MNT point to the corresponding MDT entry
- MEC points to entries in MDT
- Finding the model statement that defines a sequencing symbol
- SST stores the pair (<sequencing symbol name>, <MDT entry #>) – indicates the statement defining the sequencing symbol
- Perform expansion of a model statement
- It involves lexical substitution for formal parameters and expansion time variables
- MEC points to MDT entry
- Values of formal parameters and expansion time variables are available in APT and EVT
- Use SST to find the model statement that defines a sequencing symbol

Data Structures

- PNTAB, APTAB
- EVNTAB, EVTAB
- SSNTAB, SSTAB
- KPDTAB
- Stores intermediate code of statements in MDT

```
        MACRO
        CLEARMEM    &X, &N, &REG=AREG
        LCL         &M
&M      SET         0
        MOVER       &REG, ='0'
.MORE   MOVEM       &REG, &X+&M
&M      SET         &M+1
        AIF         (&M  NE  &N)    .MORE
        MEND
CLEARMEM    AREA, 10
```

Cont…

PNTAB

X
N
REG

EVNTAB

M

SSNTAB

MORE

MNT

	name	#PP	#KP	#EV	MDTP	KPDTP	SSTP
	CLEARMEM	2	1	1	25	10	5

SSTAB

5	27

MDT

25	(E,1)	SET	0
26		MOVER	(P,3), =‘0’
27		MOVEM	(P,3), (P,1)+(E,1)
28	(E,1)	SET	(E,1)+1
29		AIF	((E,1) NE (P,2)) (S,1)
30		MEND	

KPDTAB

10	REG	AREG

EVTAB

0

APTAB

AREA
10
AREG

Processing of Macro Definition

1. PNTAB_ptr:=1;
   SSNTAB_ptr:=1;
2. Process the macro prototype statement an form the MNT entry for the macro
   (a) name:=macro name; #PP:=0; #KP:=0;
   (b) For each positional parameter
      (i) Enter parameter name in PNTAB[PNTAB_ptr]
      (ii) PNTAB_ptr:=PNTAB_ptr+1;
      (iii) #PP:=#PP+1;
   (c) KPDTP:=KPDTAB_ptr;
   (d) For each keyword parameter
      (i) Enter parameter name and default value (if any), in the entry KPDTAB[KPDTAB_ptr]
      (ii) Enter parameter name in PNTAB[PNTAB_ptr]
      (iii) KPDTAB_ptr:=KPDTAB_ptr+1;
      (iv) PNTAB_ptr:=PNTAB_ptr+1;
      (v) #KP:=#KP+1;

Cont…

3. While not a MEND statement
   (a) If an LCL statement then
   For each expansion time variable declared in the statement:
   Enter name of expansion time variable in EVNTAB.
   #EV:=#EV+1;
   (b) If a model statement then
   (i) If the label field contains a sequencing symbol then
   If the symbol is present in SSNTAB then
   q:=entry number of the symbol in SSNTAB[SSNTAB_ptr];
   else
   Enter the sequencing symbol in SSNTAB[SSNTAB_ptr]
   q:=SSNTAB_ptr;
   SSNTAB_ptr:=SSNTAB_ptr+1;
   SSTAB_ptr:=SSTAB_ptr+1;
   SSTAB[SSTP+q-1]:=MDT_ptr;

Cont…

(ii) Construct an intermediate code for the statement as follows:

- For every occurrence of a parameter in the statement: Replace the parameter b the specification (P, #n), where n is the entry number ocupied by the parameter in the PNTAB

- For every occurrence of an expansion time variable in the statement: Replace the expansion time variable by the specification (E, #n), where n is the entry number occupied by the expansion time variable in the EVNTAB.

- For every occurrence of a sequencing symbol in the statement: Replace the sequencing symbol by the specification (S, #n), where n is the entry number occupied by the sequencing symbol in the SSNTAB.

(iii) Record the intermediate code of the statement in MDT[MDT-ptr]];

(iv) MDT_ptr:=MDT_ptr+1;

Cont…

(c) If a SET, AIF or AGO statement then

(i) If a SET statement then

For every occurrence of an expansion time variable in the statement: Replace the expansion time variable by the specification (E, #n), where n is the entry number occupied by the expansion time variable in EVNTAB.

(ii) If an AIF or AGO statement then

If the sequencing symbol used in the statement is present in SSNTAB then

q:=entry number in SSNTAB;

else

Enter the sequencing symbol in SSNTAB[SSNTAB_ptr]

q:=SSNTAB_ptr;

SSNTAB_ptr:=SSNTAB_ptr+1;

SSTAB_ptr:=SSTAB_ptr+1;

Replace the sequencing symbol by (S, SSTP+q-1)

Cont…

(iii) Record the intermediate code of the statement in MDT[MDT_ptr];

(iv) MDT_ptr:=MDT_ptr+1;

4. (The statement is a MEND statement)

(a) Record the intermediate code of the statement in MDT[MDT_ptr]

(b) MDT_ptr:=MDT_ptr+1;

(c) If SSNTAB_ptr=1 (i.e., SSNTAB is empty) then SSTP:=0

(d) If #KP=0 then KPDTP=0;

Macro Expansion

1. Perform initializations for the expansion of a macro

   (a) MEC:=MDTP field of the MNT entry

   (b) Create EVTAB with #EV entries and set EVTAB_ptr

   (c) Create APTAB with #PP+#KP entries and set APTAB_ptr

   (d) Copy the keyword parameters' default from the entries KPDTAB[KPDTP]…KPDTAB[KPDTP+#KP-1] into the entries APTAB[#PP+1]…APTAB[#PP+#KP]

   (e) Process positional parameters in the actual parameter list and them into the entries APTAB[1]…APTAB[#PP]

   (f) For each keyword parameter in the actual parameter list

   Search name of the keyword parameter in parameter name field of the entries KPDTAB[KPDTP]…KPDTAB[KPDTP+#KP-1]. Let KPDTAB[q] contain a matching entry. Enter the value of the keyword parameter in the cell (if any) in APTAB[#PP+q-KPDTP+1]

Cont…

2. While the statement pointed by MEC is not the MEND statement

(a) If a model statement then

(i) Replace operands of the form (P, #n) and (E, #m) by values in APTAB[n] and EVTAB[m] respectively

(ii) Output the generated statement

(iii) MEC:=MEC+1

(b) If a SET statement with the specification (E, #m) in the label field then

(i) Evaluate the expression in the operand field using values from the APTAB and EVTAB where appropriate and set the resulting value in EVTAB

Cont…

(ii) MEC:=MEC+1

(c) If an AGO statement with (S, #s) in operand field then

MEC:=SSTAB[SSTP+s-1]

(d) If an AIF statement with (S, #s) in operand field then

Evaluate the condition. If the condition is true then

MEC:=SSTAB[SSTP+s-1]

3. Exit from macro expansion

Nested Macro Call

- Examine each statement generated to check whether it is itself a macro call. If so, expand the nested macro call
- Each macro under expansion should have its own set of data structures – stored in a stack
- Expansion nesting counter: to count the number of nested macro calls. It is incremented when a macro call is recognised, and decremented when MEND statement is encountered

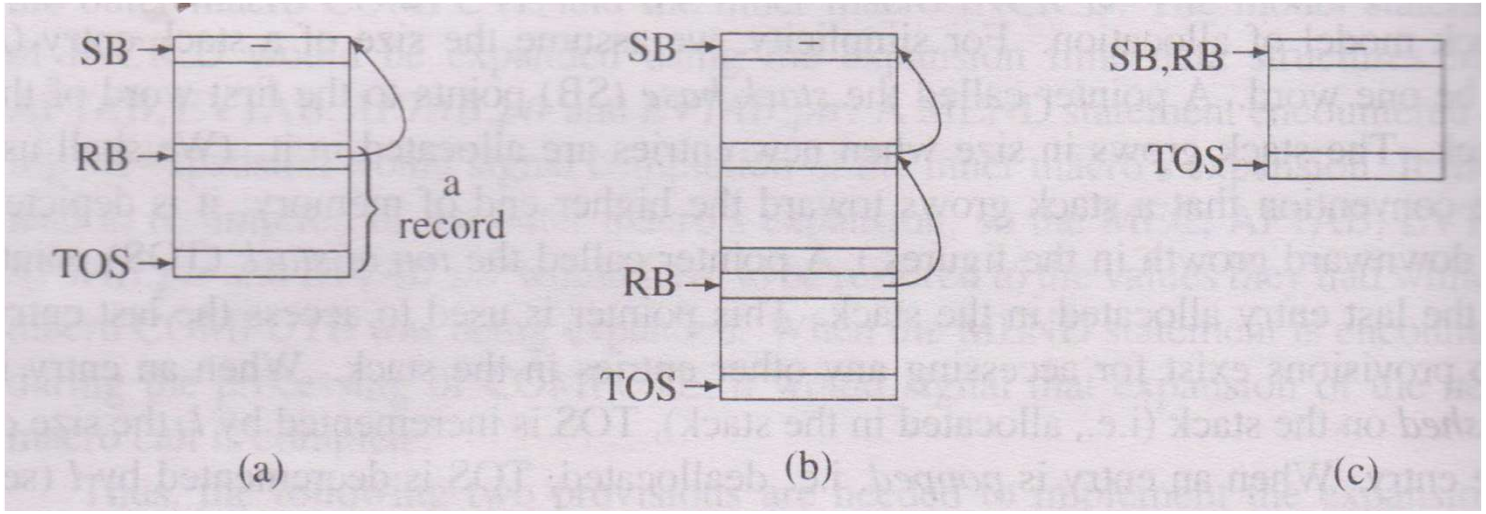
Cont…

Allocation

- TOS:=TOS+1
- TOS*:=RB
- RB:=TOS
- TOS:=TOS+n

Deallocation

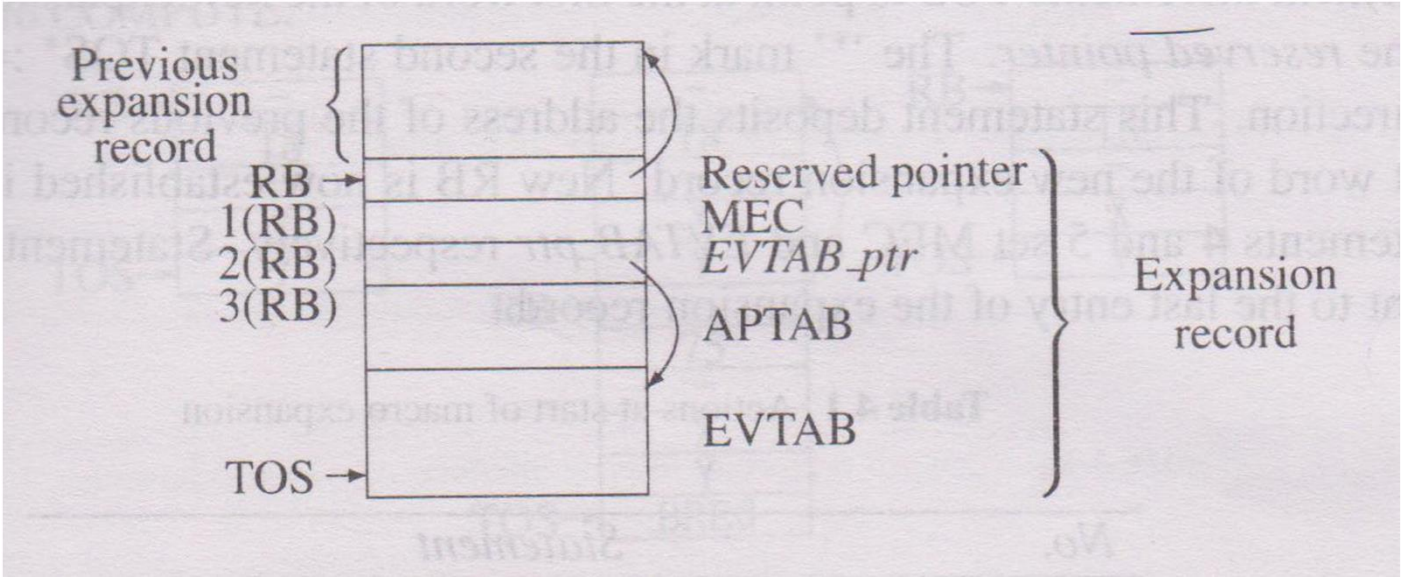
1. TOS:=RB-1
2. RB:=RB*



Cont…

At the start of macro expansion

- TOS:=TOS+1
- TOS*:=RB
- RB:=TOS
- 1(RB):=MDTP entry of MNT
- 2(RB):=RB+3+#eAPTAB
- TOS:=TOS+#eAPTAB+#eEVTAB+2



Macro Assembler

- Macro expansion + translation
- Many functions get duplicated while using macro preprocessor followed by assembler

Pass I

- Macro definition processing
- Enter names and types of symbols in SYMTAB

Pass II

- Macro expansion
- Memory allocation and LC processing
- Processing of literals
- IC generation

Pass III

- Target code generation

Cont…

Pass I

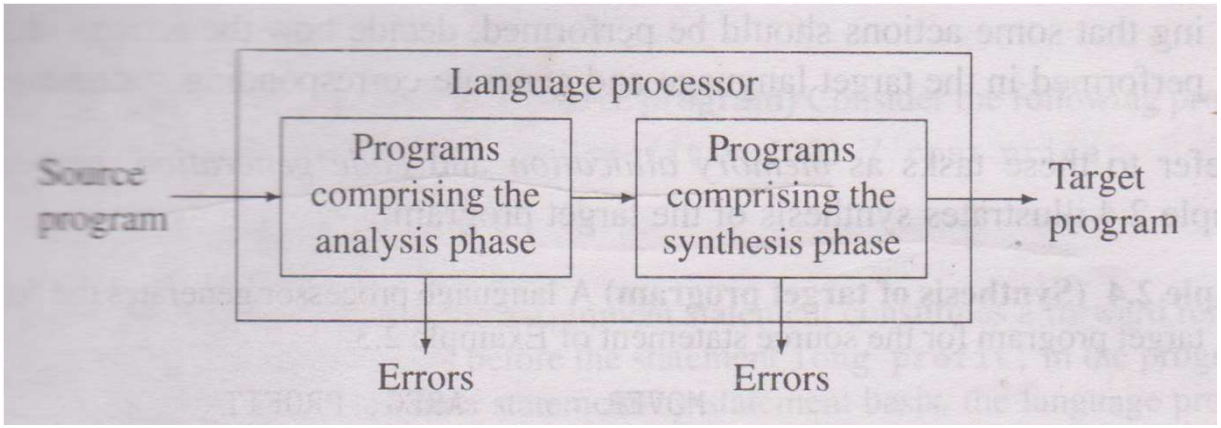
- Macro definition processing
- Macro expansion
- Memory allocation, LC processing and construction of SYMTAB
- Processing of literals
- IC generation

Pass II

- Target code generation

A large imbalance in size exists between the passes

Compiler



- Language processing = Analysis of source program + Synthesis of target program

Components of source program analysis

- Lexical rules govern formation of valid lexical units (operators, identifiers, constants) in the source language
- Syntax rules govern formation of valid statements in the source language
- Semantic rules associate meaning with valid statements

Cont…

- percent = (profit * 100) / price;
- Lexical analysis identifies =, *, / (operators), percent, profit, price (identifiers), 100 (constant)
- Syntax analysis reveals the assignment statement with percent (LHS variable) and (profit * 100) / price as expression (RHS)
- Semantic analysis deduces assignment of (profit * 100) / price to percent

Cont…

Synthesis of the target program

- Memory allocation: how the declared data should be represented in the target language and generate corresponding declaration statements
- Code generation: how the actions should be performed in the target language and generate corresponding statements

```
                MOVER       AREG, PROFIT
                MULT        AREG, HUNDRED
                DIV         AREG, PRICE
                MOVEM       AREG, PERCENT
                ……..
PERCENT         DW          1
PROFIT          DW          1
PRICE           DW          1
HUNDRED         DC          ‘100’
```

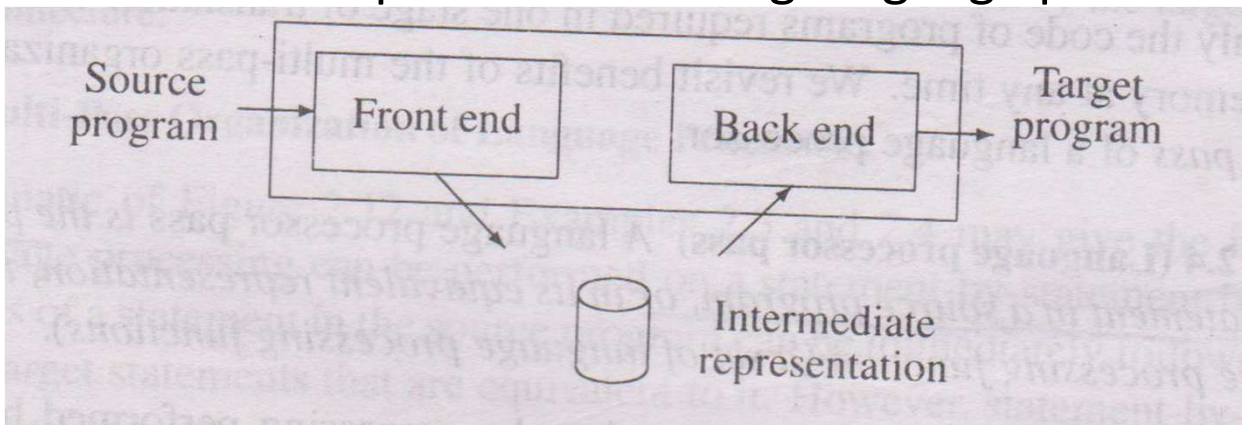
Multi-pass Organisation

- Statement-by-statement processing may not be feasible due to …

Forward reference of a program entity is a reference to the entity before it is declared/defined

Require more memory to store the code of both phases of language processor at the same time

- A language processor pass is the processing of every statement in a source program, or in its equivalent representation, to perform a language processing function
- An intermediate representation is a representation of a source program which reflects the effect of some, but not all, analysis and synthesis functions performed during language processing



Front End

- Performs lexical, syntax and semantic analysis

Functions

- Check the validity of a source statement
- Determine the content of a source statement. i.e., lexical class, syntactic structure of a source statement, meaning of a statement
- Construct a suitable representation of the source statement for use by subsequent analysis functions/synthesis phase. i.e., tables and intermediate code

Cont…

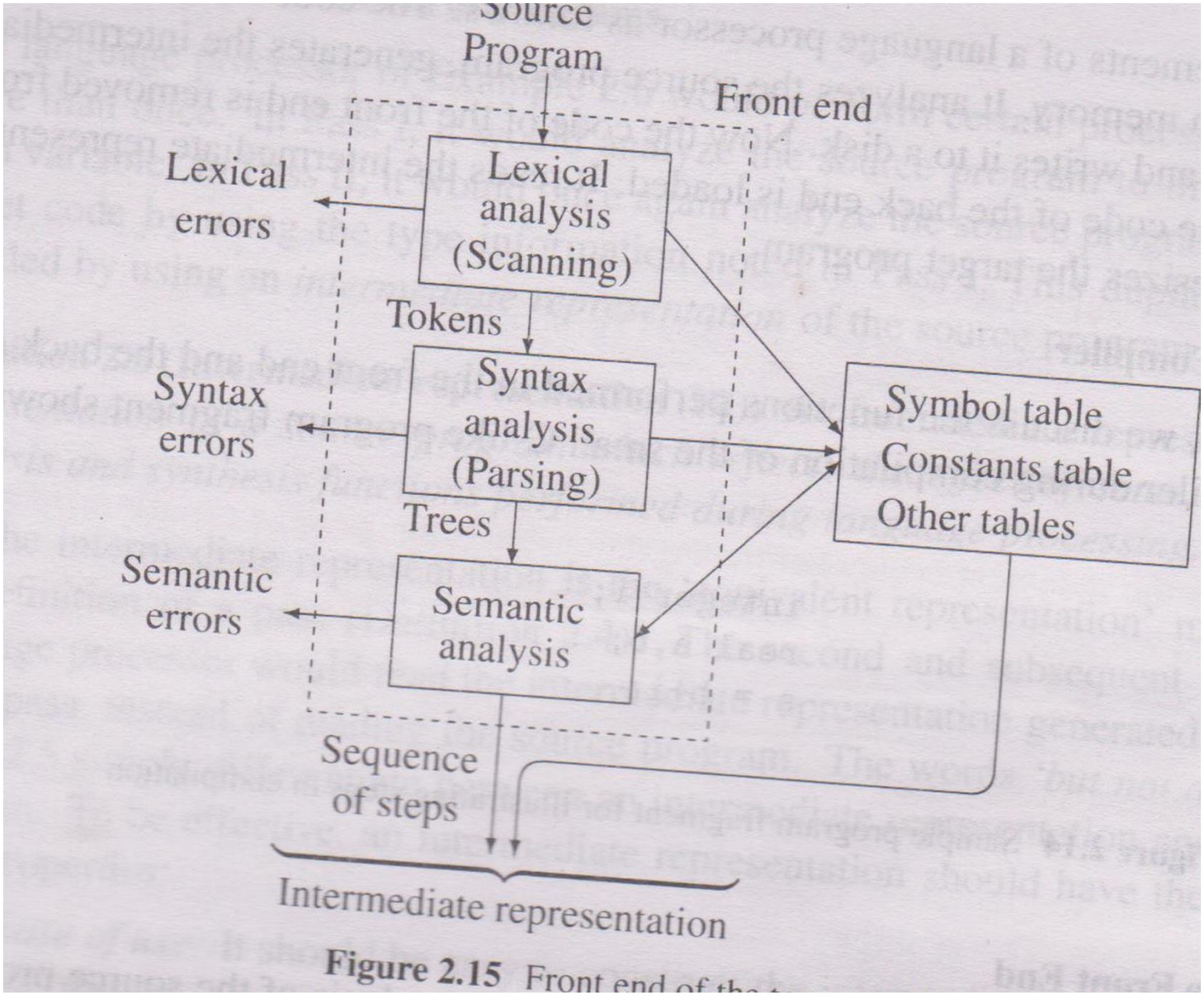


Figure 2.15 Front end of the

Lexical Analysis

- It identifies lexical units, classifies them into lexical classes (operator, identifier, constant) and enters them into relevant tables
- It builds table (for each lexical class) and IC called *tokens* (lexical class, number in class) – descriptor for a lexical unit

int i;

real a, b;

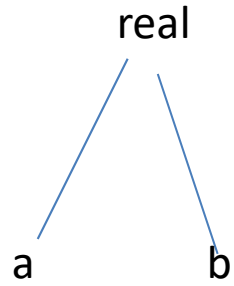
a = b + i;

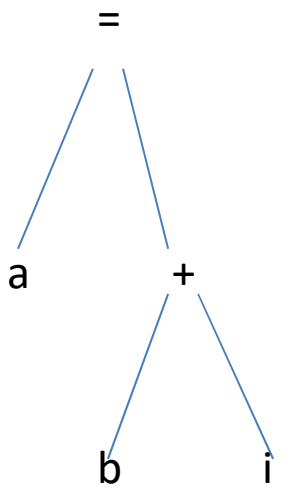
Id#2	Op#5	Id#3	Op#3	Id#1	Op#10

	Symbol	Type	Length	Address
1	i			
2	a			
3	b			

Symbol table

Syntax Analysis





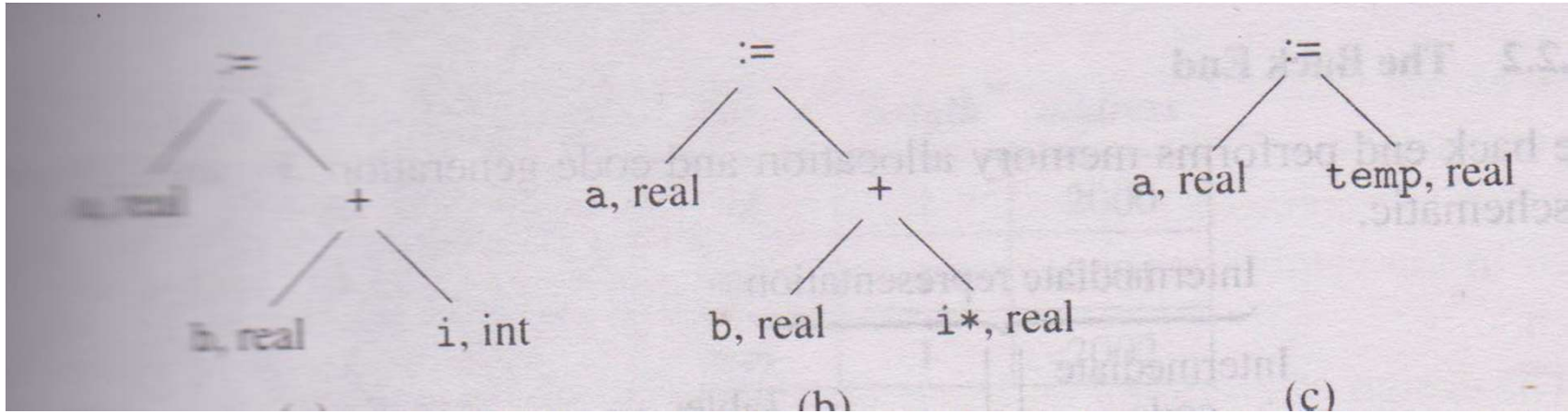
- Processes tokens to determine its grammatical structure and builds an IC (tree)

Semantic Analysis

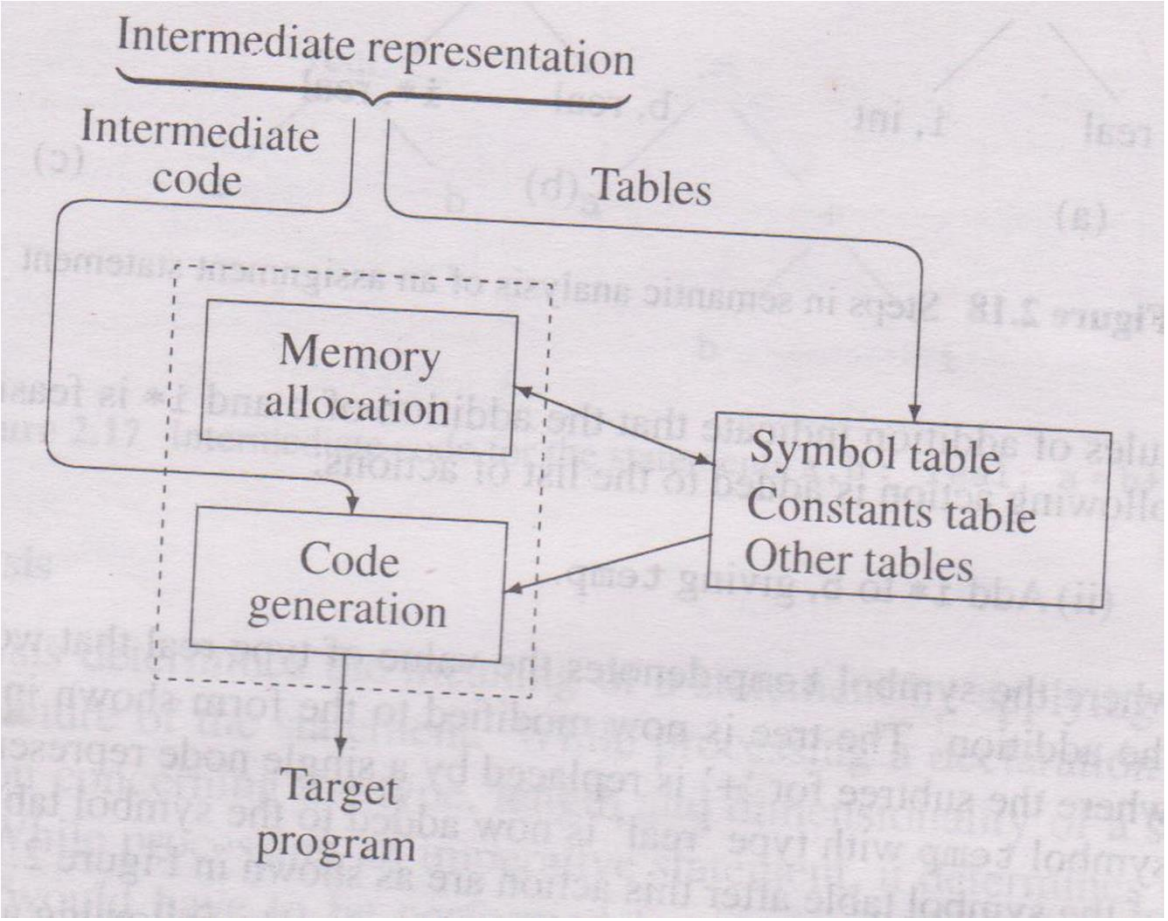
- Process the tree to determine the meaning of a statement by applying the semantic rules to the structure of the statement
- It adds type, length and dimensionality of a symbol to symbol table (for declarative statement)
- It determines the sequence of actions to be performed for implementing the meaning and represents them in IC (for imperative statement)
- IR (by analysis phase) = updated tables + sequence of actions

Cont…

	Symbol	Type	Length	Address
1	i	int		
2	a	real		
3	b	real		
4	i*	real		
5	temp	real		



Back End


Intermediate representation
Intermediate code
Tables
Memory allocation
Code generation
Symbol table
Constants table
Other tables
Target program

Cont…

- Memory allocation: Compute memory requirement of a variable from its type, length and dimensionality

	Symbol	Type	Length	Address
1	i	int	1	2000
2	a	real	1	2001
3	b	real	1	2002

- Code generation: values of i* and temp are in AREG

```
CONV_R      AREG, I
ADD_R       AREG, B
MOVEM       AREG, A
```

Grammar

- A grammar G of a language L_G is a quadruple (Σ, SNT, S, P) where

Σ is the alphabet of L_G, i.e., set of terminal symbols

SNT is the set of nonterminal symbols

S is the distinguished symbol

P is the set of productions

Classification of Grammars

- Type-0 (phrase structure) grammar

Productions are of the form $\alpha ::= \beta$

where α and β can be strings of terminal and NT symbols

- Type-1 (context sensitive) grammar

Production has the form $\alpha A \beta ::= \alpha \pi \beta$

π can be replaced by A only when it is enclosed by strings α and β in a sentential form

- Type-2 (context free) grammar

Production is of the form $A ::= \pi$

Suited for programming languages. E.g. Algol-60, Pascal

- Type-3 (linear or regular) grammar

A ::= tB|t OR A ::= Bt| t suited for lexical units

Classified into left linear and right linear depending on the NT in RHS alternative appears at the extreme left or extreme right

- Operator grammars: Productions of an operator grammar don't contain two or more consecutive NT symbols in any RHS alternative. They are separated by one or more terminal symbols known as operator

Scanning

- Scanner (Generated by LEX) is a program that performs scanning. It identifies lexical units, enters each lexical unit into relevant table and constructs a token
- Finite State Automaton (FSA) is a triple (S, Σ, T) where

S is a finite set of states, one of which is the initial state s_{init}, and one or more of which are final states

Σ is the alphabet of source symbols

T is a finite set of state transitions defining transitions out of states in S on encountering symbols in Σ

- Deterministic finite state automaton (DFA) is a FSA none of whose states has two or more transitions for the same source symbol. It reaches a unique state for every source string input to it
- Each transition of DFA can be represented by the triple (old state, source symbol, new state). Alternatively, the transitions of DFA can be represented in the form state transition table which has one row for each state s_i in S and one column for each symbol *symb* in Σ.

Cont…

- Type-3 rule <integer> ::= d|<integer>d

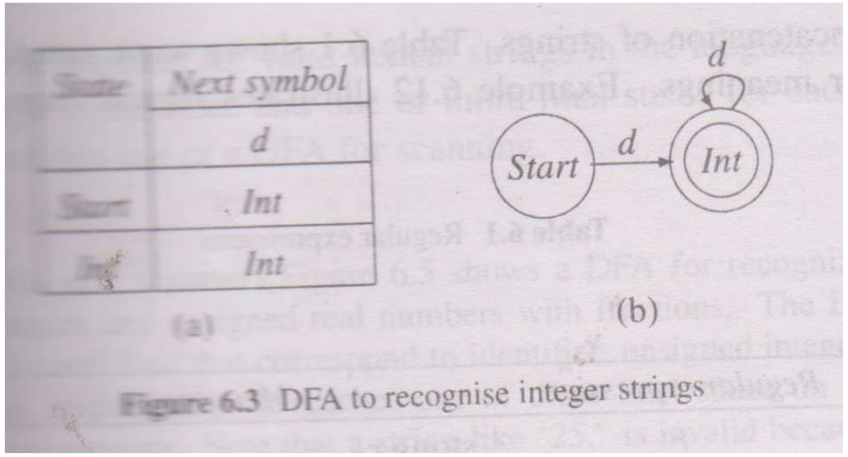


Figure 6.3 DFA to recognise integer strings

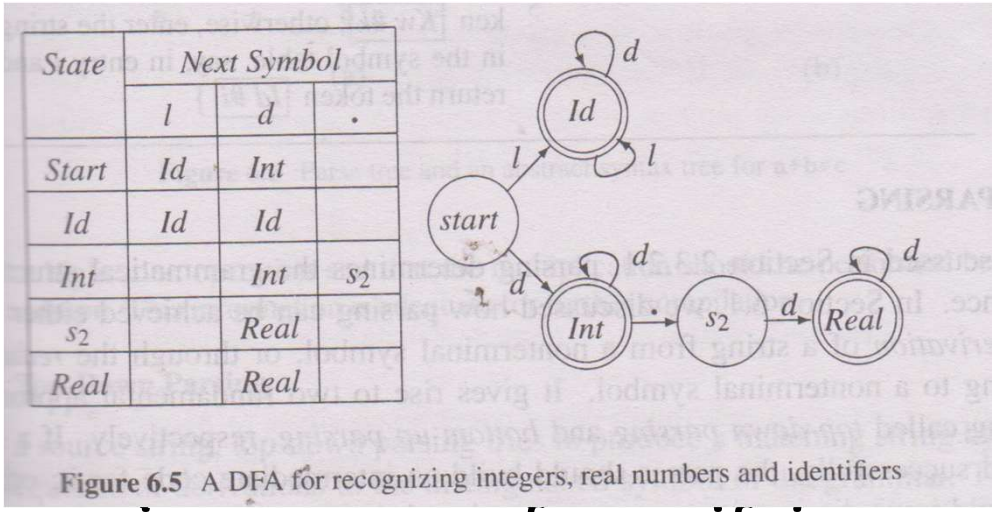


Figure 6.5 A DFA for recognizing integers, real numbers and identifiers

- Regular Expression: A concise means of specifying lexical units
- Integer : $[+|-](d)^{+}$
- Real number : $[+|-](d)^{+}.(d)^{+}$
- Real number with optional fraction: $[+|-](d)^{+}.(d)^{*}$
- Identifier : $l(l|d)^{*}$

Cont…

- Specification of a scanner

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

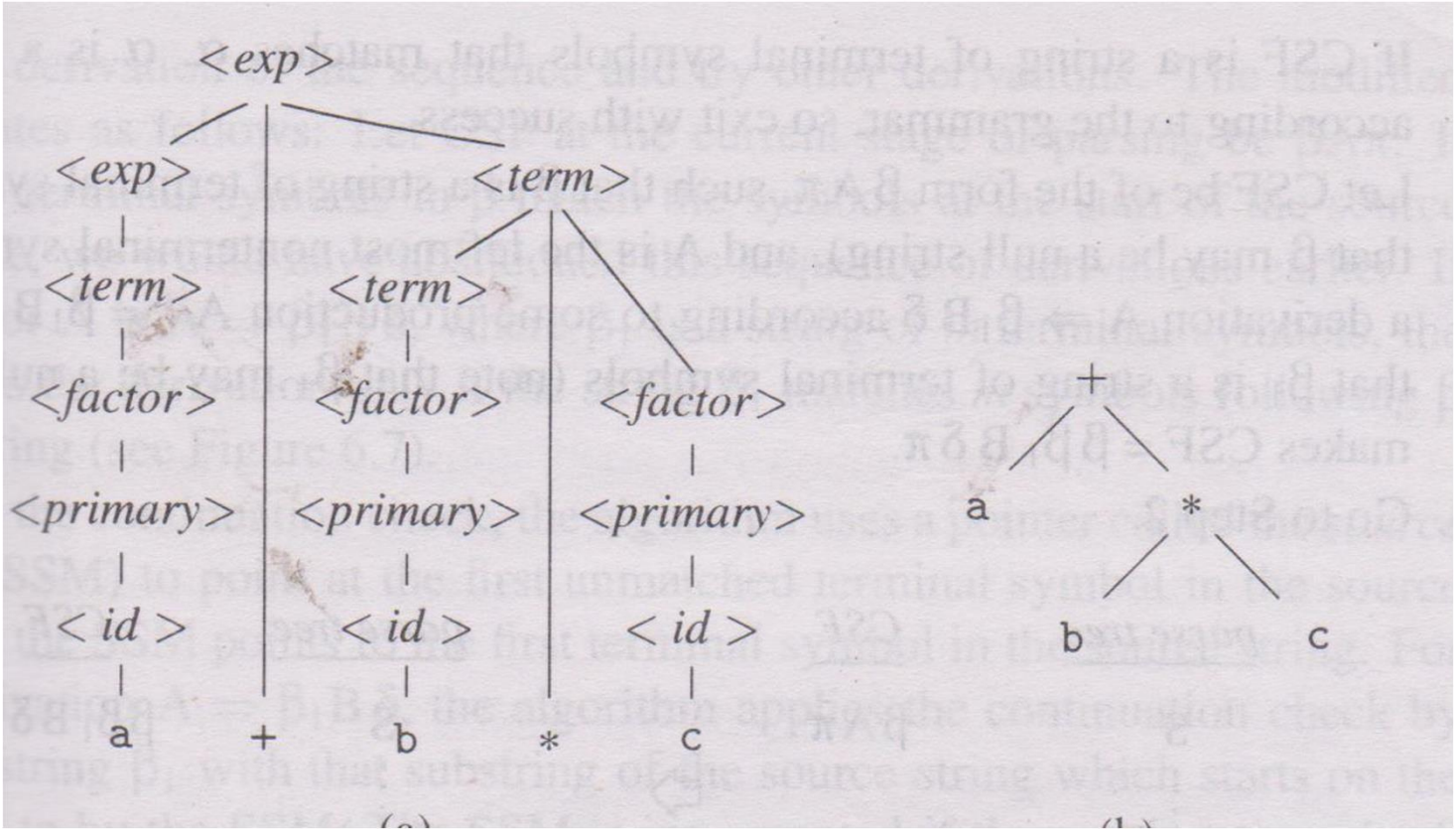
Parsing

- Determines grammatical structure of a sentence
- Achieved through derivation of a string from a NT symbol (top-down parsing), OR through the reduction of a string to a NT symbol (bottom-up parsing)
- If parsing is successful, parser builds an intermediate code (parse tree or AST); otherwise issues diagnostic messages
- Parse tree depicts the steps in parsing of a source string according to a grammar. It is a poor IR for a source string, because much of the information contained in it is not useful for subsequent processing
- AST represents the structure of a source string in a more economical manner. A source string may have different AST in different compilers, but its parse tree is unique

Cont…

- <exp> ::= <exp> + <term>|<term>
- <term> ::= <term> * <factor>|<factor>
- <factor> ::= <factor> ^ <primary>|<primary>
- <primary> ::= <id>|<const>|(<exp)
- <id> ::= <letter>|<id><letter>|<id><digit>
- <const> ::= [+|-]<digit>|<const><digit>
- <letter> ::= a|b|…|z
- <digit> ::= 0|1|…|9

Cont…


<exp>
<exp>
<term>
<term>
<term>
<factor>
<factor>
<factor>
<primary>
<primary>
<primary>
< id >
< id >
< id >
a
+
b
*
c
+
a
*
b
c

Top-Down Parsing

- $S \Rightarrow \ldots \Rightarrow \ldots \Rightarrow \alpha$
- It is also know as LL parsing, since we make a derivation for the leftmost NT at any stage

Procedure

- SSM points to first unmatched symbol in the source string
- Prediction making mechanism makes prediction for the leftmost NT symbol in the CSF
- Matching and backtracking mechanism implements the continuation check by using SSM

Example

- S ::= E
- E ::= T + E | T
- T ::= V * T | V
- V ::= <id>

Prediction	CSF
-	E
E ⇒ T + E	T + E
T ⇒ V	V + E
V ⇒ <id>	<id> + E
E ⇒ T	<id> + T
T ⇒ V * T	<id> + V * T
V ⇒ <id>	<id> + <id> * T
T ⇒ V	<id> + <id> * V
V ⇒ <id>	<id > + <id> * <id>

Recursive Descent Parser

- No backtracking. It uses a set of mutually-recursive procedures to perform parsing

E ::= T{+T}*

T ::= V{*V}*

V ::= <id>

```
procedure proc_E : (tree_root);
/* This procedure constructs an AST for E and returns a pointer to its root */
var
    a, b : pointer to a tree node;
begin
    proc_T(a);
    while (nextsymb = '+') do
            nextsymb := next source symbol;
            proc_T(b);
            a := treebuild('+', a, b);
    /* Builds an AST and returns pointer to its root */
    tree_root := a;
    return;
end proc_E;
```

Cont…

```
procedure proc_T (tree_root);
/* This procedure constructs an AST for T and returns a pointer to its
    root */
var
    a, b : pointer to a tree node;
begin
    proc_V(a);
    while(nextsymb=‘*’) do;
            nextsymb := next source symbol;
            proc_V(b);
            a := treebuild (‘*’, a, b);
    tree_root := a;
    return;
end proc_T;
```

Cont…

```
procedure proc_V(tree_root);
/* This procedure constructs an AST for V and returns a
   pointer to its root */
var
   a : pointer to a tree node;
begin
   if (nextsymb = <id>) then
         nextsymb := next source symbol;
         tree_root := treebuild (<id>, -, -);
   else print “Error”;
   return;
end proc_V;
```

LL(1) Parser

Non terminal	Source symbol			
	<id>	+	*	-|
E	E ⇒TE’			
E’		E’ +TE’		E’⇒ε
T	T ⇒VT’			
T’		T’⇒ ε	T’⇒ *VT’	T’⇒ ε
V	<id>			

Bottom-up Parsing

- Parser constructs a parse tree by applying a sequence of reduction to a source string
- Source string is valid if it can be reduced to S. If not, an error is detected and reported
- Parser performs as many *reductions* as possible at the current position of SSM. When no reductions are possible at the current position, SSM will advance by one symbol (*shift* action). Parsing thus consists of shift and reduce actions applied in a left-to-right manner. Hence bottom-up parsing is also known as LR parsing or shift-reduce parsing

Operator Precedence Matrix

Left operator	Right operator				
	+	*	(	)	-|
+	.>	<.	<.	.>	.>
*	.>	.>	<.	.>	.>
(	<.	<.	<.	=	
)	.>	.>		.>	.>
|-	<.	<.	<.		=

Operator Precedence Parsing

- Function: newnode(operator, l_operand_pointer, r_operand_pointer) creates a node with appropriate pointer fields and returns a pointer to the node
- Input: An expression string enclosed between ‘|-’ and ‘-|’

1. TOS := SB-1; complete := false

   Push ‘|-’ on the stack

   Set SSM to point at the second source symbol

2. If current source symbol is an operand then

{Build a node for the operand and store its pointer in TOS entry}

x := newnode(source symbol, null, null)

TOS.operand_pointer := x

Advance SSM by one character

Cont…

3. While (not complete)
4. While TOS operator .> current operator
   {Perform an appropriate reduce action. Build AST for reduced string. Check whether more reduce actions are needed}
   x := newnode(TOS operator, TOSM.operand_pointer, TOS.operand_pointer)
   Pop an entry off the stack
   TOS.operand_pointer := x
5. If TOS operator <. Current operator then
   {Perform a shift action}
   Push the current operator on the stack
   Advance SSM by one character
6. If current source symbol is an operand then
   {Build a node for the operand and store its pointer in TOS entry}
   x := newnode(source symbol, null, null)
   TOS.operand_pointer := x
   Advance SSM by one character

Cont…

7. If TOS operator = current operator, then

    If TOS operator = ‘|-’, then complete := true {Parsing successful}

    If TOS operator = ‘(‘, then

        {Current operator must be ‘)’, hence drop the parentheses}

        temp := TOS.operand_pointer

        Pop an entry off the stack

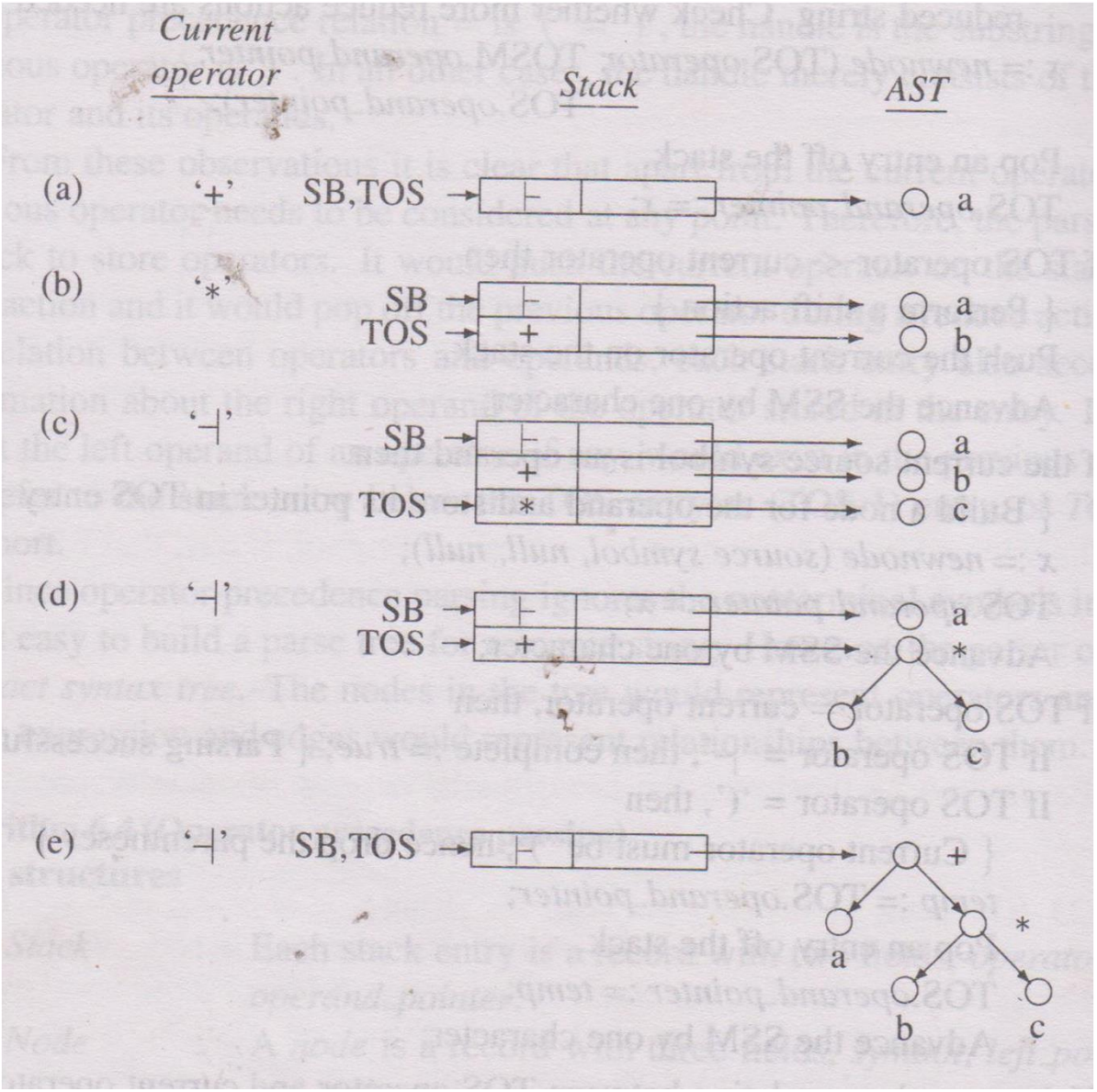
        TOS.operand_pointer := temp

        Advance SSM by one character

8. If the precedence relation between TOS operator and current operator is undefined then

    Report error; complete := true

Cont…


Current operator
Stack
AST
(a)
'+'
SB,TOS
a
(b)
'*'
SB
TOS
+
a
b
(c)
'⊣'
SB
+
TOS
*
a
b
c
(d)
'⊣'
SB
TOS
+
a
*
b
c
(e)
'⊣'
SB,TOS
+
a
*
b
c

Memory Allocation

- Activity of performing memory binding
- Memory binding is an association between memory address attribute of a data item (type, dimensionality) and the address of a memory area

Types of variables (dynamic memory allocation)

- Automatic: variable is created automatically when the block is entered during program execution and destroyed when the block is exited
- Program controlled: variable is created and destroyed through the execution of functions such as *calloc* and *free*

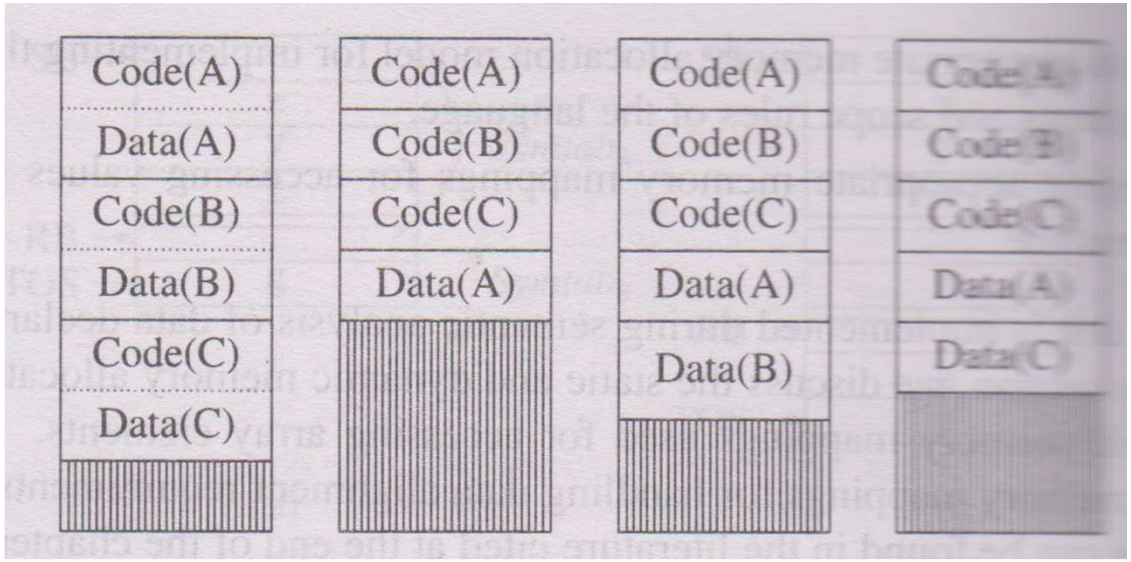
Tasks

- Determine memory requirement – implemented during semantic analysis
- Use appropriate memory allocation model – static and dynamic
- Develop appropriate memory mappings - array

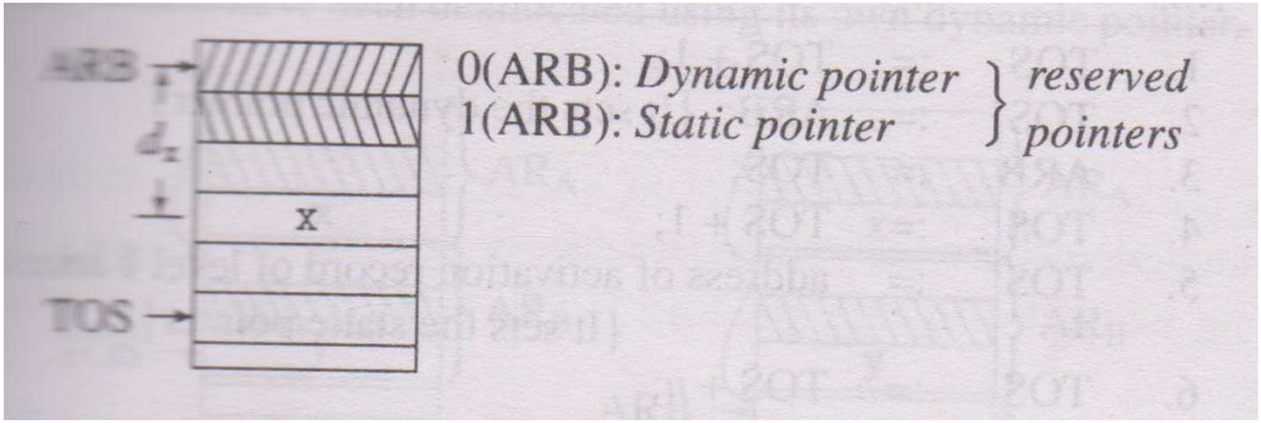
Memory Allocation Models

- Static memory allocation: memory is allocated to a variable before the execution of a program begins. i.e., performed during compilation. E.g. Fortran
- Dynamic memory allocation: memory bindings are established and destroyed during execution of a program. e.g. C, C++, Java
- Automatic dynamic allocation is implemented using stack. Entry and exit of a block is on LIFO basis
- Program controlled dynamic allocation is implemented using heap. Allocate or deallocate memory at any time during execution
- Adv: Recursion is easy to implement
- Suitable for data structures whose size varies dynamically

Cont…



2 to n+1



- Each activation record accommodates variables in an active block
- ARB (register) contains start address of TOS record
- Dynamic pointer performs memory allocation/deallocation. It points to the previous activation record
- Static pointer assists in accessing nonlocal variables

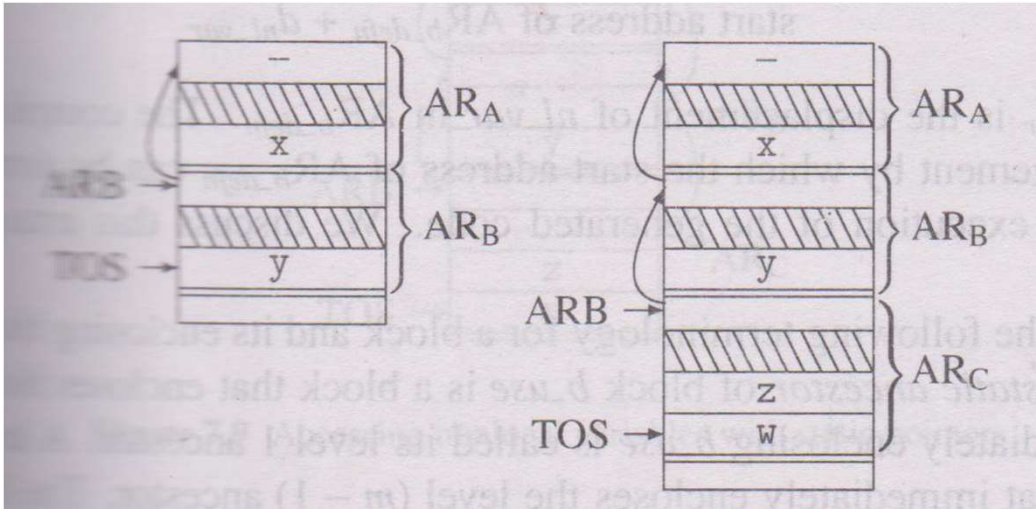
Cont…

Actions at block entry

- TOS=TOS+1
- TOS*=ARB Sets dynamic pointer
- ARB=TOS
- TOS=TOS+1
- TOS*=address of activation record of level 1 ancestor
  Sets static pointer
- TOS=TOS+n

Actions at block exit

- TOS=ARB-1
- ARB=ARB*



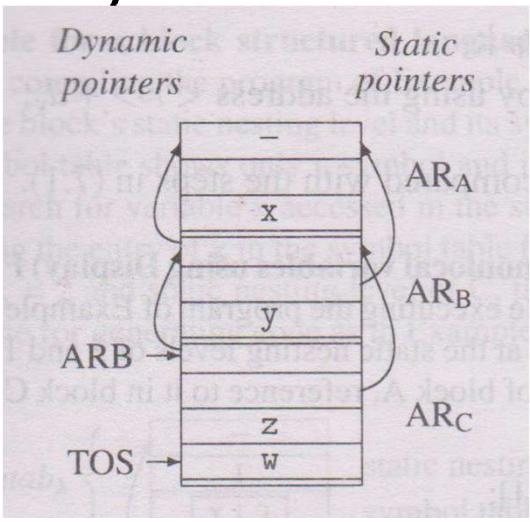
Cont…

- Access local variable
- <ARB>+d_v or d_v(ARB) – store this value in symbol table entry of variable v
- Access nonlocal variable

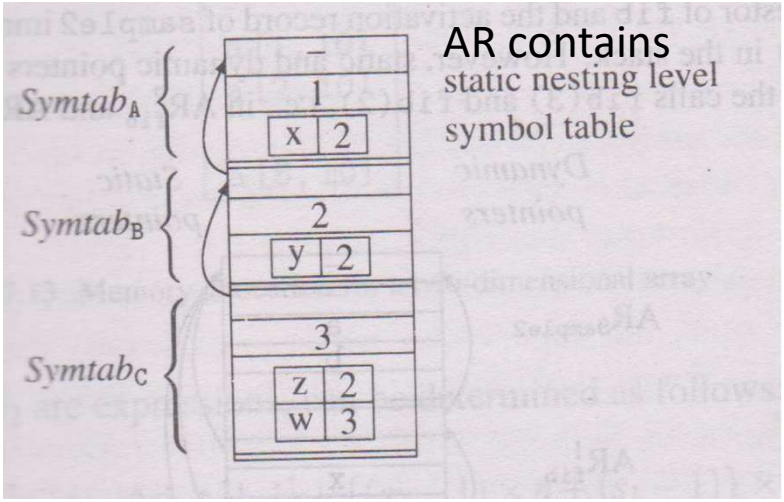
Start address of AR_{b_defn}+ d_{nl_var}

- r=ARB, where r is some CPU register
- Repeat following next step 3 m times

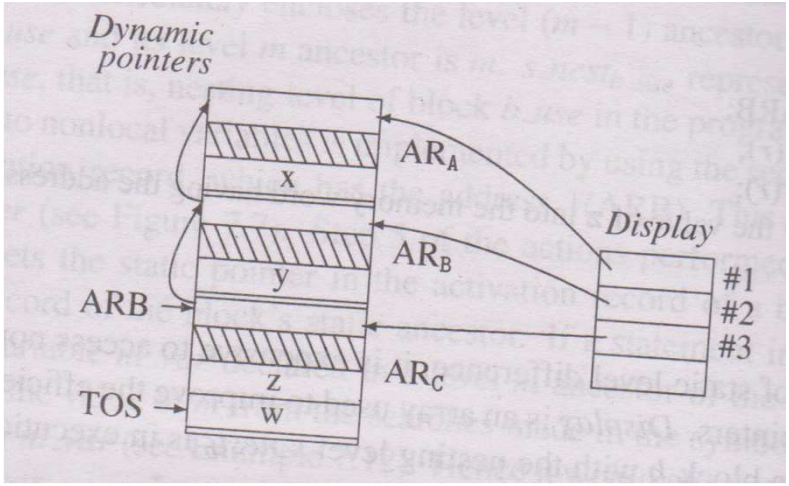
  To access *nl_var* from level m ancestor



- r=1(r), i.e., load static pointer of the activation record to which register r is pointing into CPU register r
- Access *nl_var* by using the address <r>+ d_{nl_var}

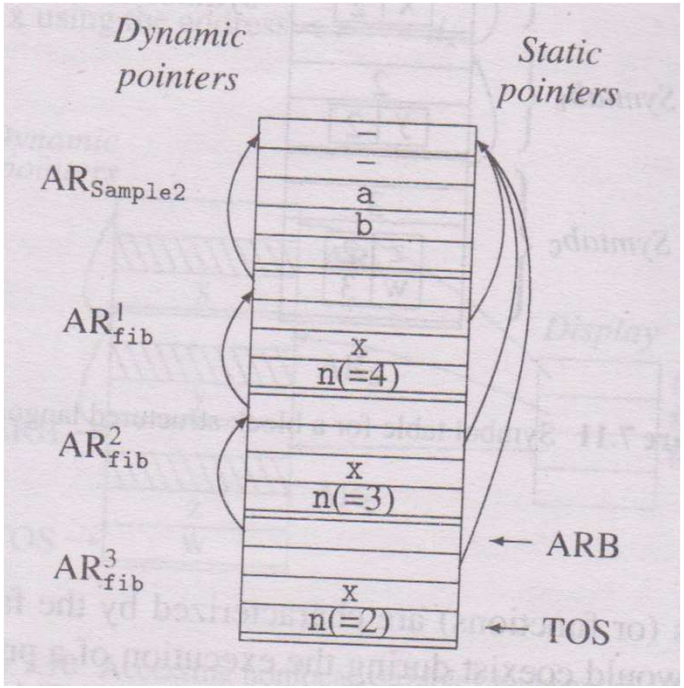


Display



- An array used to improve the efficiency of nonlocal access.
- Display[1]=address of AR of level (s_nest_b-1) ancestor of b
- Display[s_nest_b]=address of AR_b
- To access variable v (defined in ancestor block b') from block b
- r=Display[$s\_nest_{b'}$]
- Access variable v by using the address <r>+d_v

Recursion


Dynamic pointers
Static pointers
AR Sample2
a
b
AR 1 fib
x
n(=4)
AR 2 fib
x
n(=3)
AR 3 fib
x
n(=2)
ARB
TOS

Array

- Ad. $a[s_1,s_2]$=Ad. $A[0,0]+(s_2 \times range_1 + s_1) \times k$

where $range_1=u_1-l_1+1$, and k is the memory requirement of an element

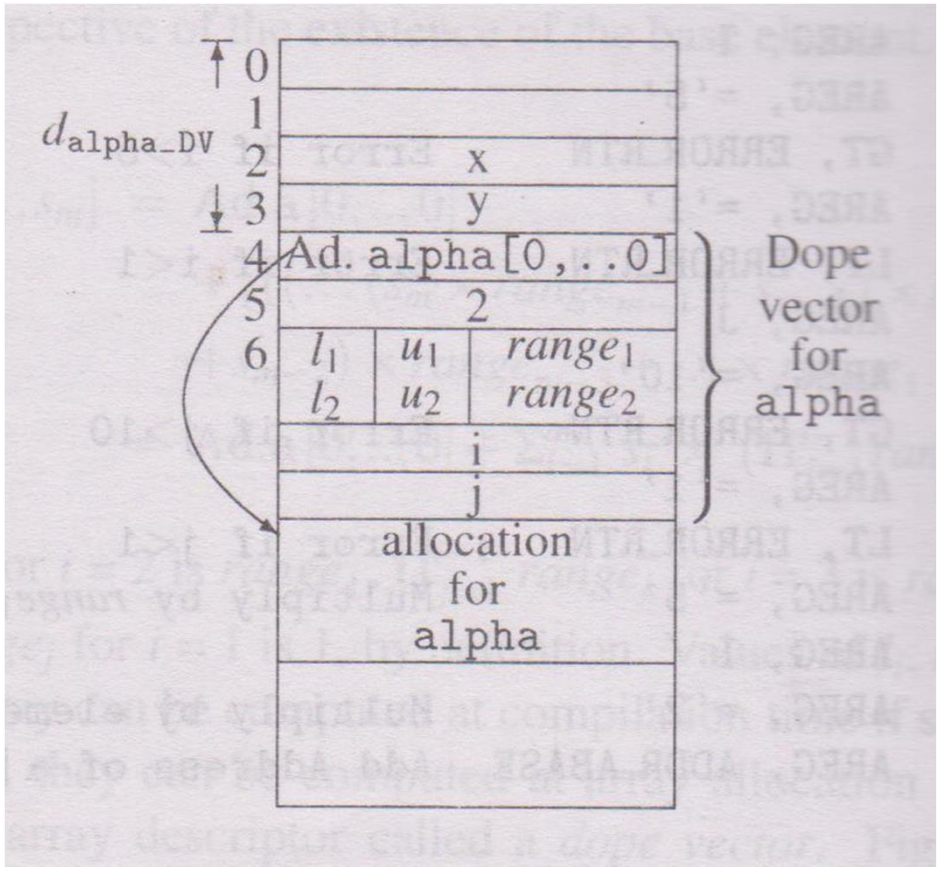
- Dope vector

Ad. a[0,…,0]		
No. of dimensions (m)		
l_1 l_2 : l_m	u_1 u_2 : u_m	$range_1$ $range_2$: $range_m$

Target code for array reference

```
MOVER   AREG, I
COMP    AREG, =‘5’
BC      GT, ERROR_RTN
COMP    AREG, =‘1’
BC      LT, ERROR_RTN
MOVER   AREG, J
COMP    AREG, =‘10’
BC      GT, ERROR_RTN
COMP    AREG, AREG, =‘1’
BC      LT, ERROR_RTN
MULT    AREG, =‘5’
ADD     AREG, I
MULT    AREG, =‘2’
ADD     AREG, ADDR_ABASE
```

a[1:5, 1:10]



Compilation of Expressions

Operand descriptors

- Built for each identifier, constant and partial result
- Attributes (type, length)
- Addressibility [addressibility code(M,R,AR,AM), Address] – specifies where and how operand can be accessed

Register descriptors

- Built for every CPU register
- Status (free or occupied) – register status
- Operand descriptor #: descriptor # of the operand contained in the register

Cont…

- a*b

Operand_descriptor[1]

Attributes	Addressibility
(int, 1)	M. addr(a)

Operand_descriptor[2]

Attributes	Addressibility
(int, 1)	M, addr(b)

- Code generation

```
MOVER      AREG, A
MULT       AREG, B
```

- Operand_descriptor[3]

Attributes	Addressibility
(int, 1)	R. addr(AREG)

For Partial result

- Register descriptor for AREG

Status	Operand_descriptor #
Occupied	#3

Code Generator

- %%

```
E : E + T        {$$ = codegen('+', $1, $3)}
  | T            {$$ = $1}
T : T * F        {$$ = codegen('*', $1, $3)}
  | F            {$$ = $1}
F : id           {$$ = build_descriptor($1)}
```

- %%

```
build_descriptor(operand)
{   i = i + 1;
    operand_descr[i] = ((type, (addressibility_code, address)) of
    operand;
    return i;     }
```

Partial Result

- a*b+c*d

Attributes	Addressibility
(int, 1)	M. addr(temp[1])

Partial result of a*b moved to memory

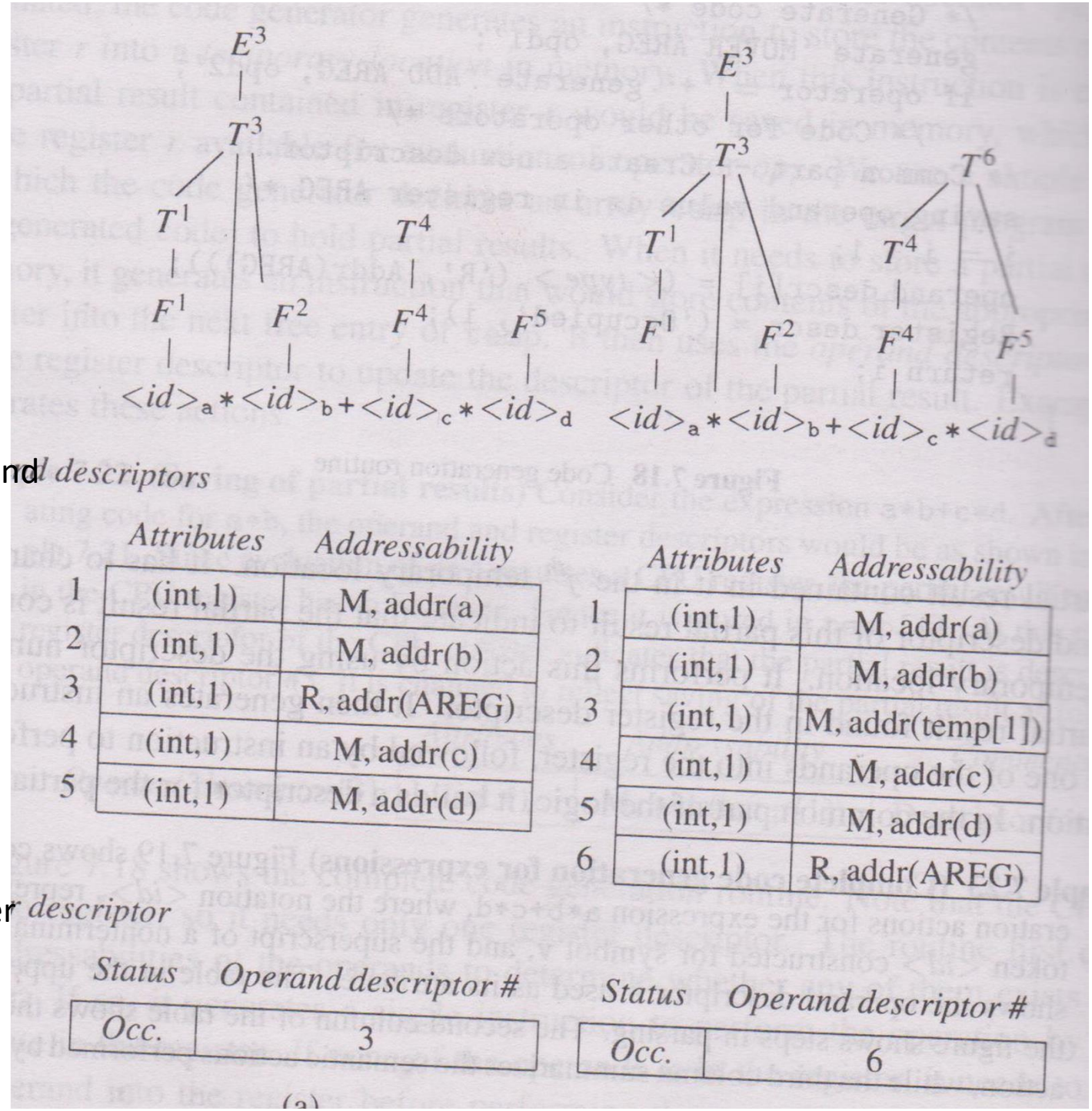
Operand_descriptor[3]

```
codegen(operator, opd1, opd2)
{   if opd1.addressibility_code = 'R'
        if operator = '+' generate 'ADD AREG, opd2'
    else if opd2.addressibility_code = 'R'
        if operator = '+'   generate 'ADD AREG, opd1'
    else
        if Register_descr.status = 'Occupied'
            generate('MOVEM AREG, Temp[j]')
            j = j + 1
            Operand_descr[Register_descr.Operand_descriptor#] =
            (<type>, (M, Addr(Temp[j])))
        generate 'MOVER AREG, opd1'
        if operator = '+' generate 'ADD AREG, opd2'
    i = i + 1
    Operand_descr[i] = (<type>, ('R', Addr(AREG)))
    Register_descr = ('Occupied', i)
    return i        }
```

Cont…

- $<id>_a$ -> F^1 — Build descriptor # 1
- F^1 -> T^1
- $<id>_b$ -> F^2 — Build descriptor # 2
- $T^1 * F^2$ -> T^3 — Generate MOVER AREG, A
  MULT AREG, B
  Build descriptor # 3
- T^3 -> E^3
- $<id>_c$ -> F^4 — Build descriptor # 4
- F^4 -> T^4
- $<id>_d$ -> F^5 — Build descriptor # 5
- $T^4 * F^5$ -> T^6 — Generate MOVEM AREG, TEMP_1
  MOVER AREG, C
  MULT AREG, D
  Build descriptor # 6
- $E^3 + T^6$ -> E^7 — Generate ADD AREG, TEMP_1

Cont…



operand descriptors

	Attributes	Addressability
1	(int,1)	M, addr(a)
2	(int,1)	M, addr(b)
3	(int,1)	R, addr(AREG)
4	(int,1)	M, addr(c)
5	(int,1)	M, addr(d)

	Attributes	Addressability
1	(int,1)	M, addr(a)
2	(int,1)	M, addr(b)
3	(int,1)	M, addr(temp[1])
4	(int,1)	M, addr(c)
5	(int,1)	M, addr(d)
6	(int,1)	R, addr(AREG)

register descriptor

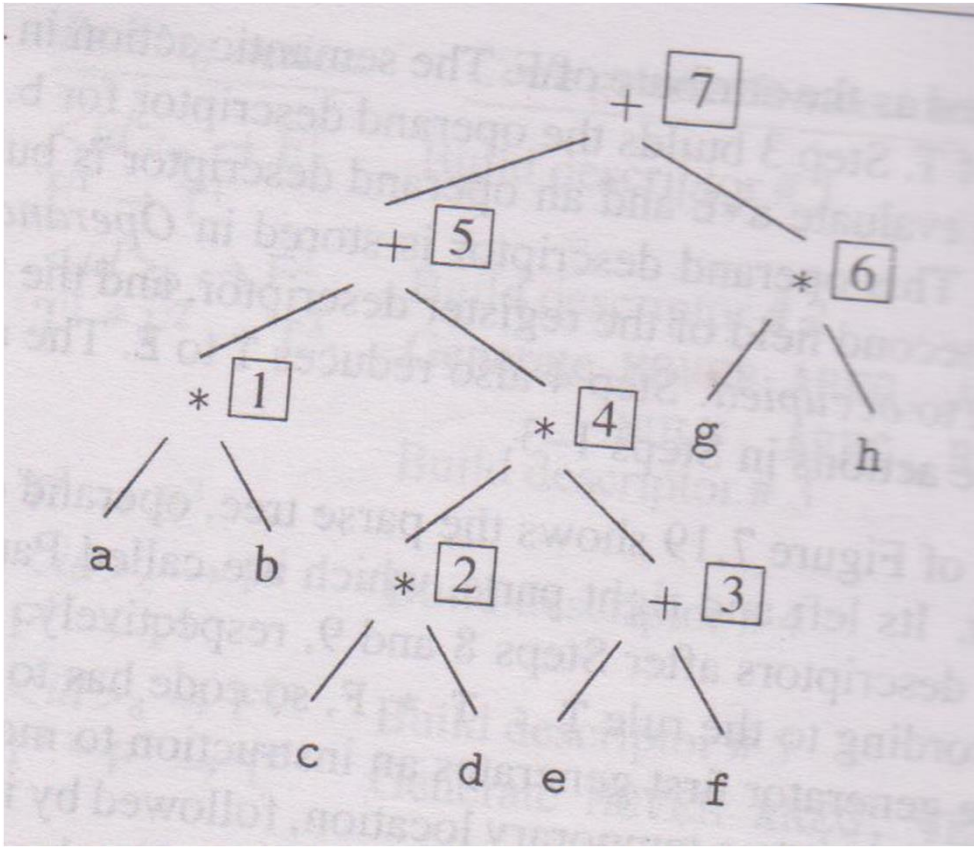
Status	Operand descriptor #
Occ.	3

Status	Operand descriptor #
Occ.	6

(a)

Expression Tree

- a*b+c*d*(e+f)+g*h



Cont…

```
MOVER  AREG, A
MULT   AREG, B
MOVEM  AREG, TEMP_1
MOVER  AREG, C
MULT   AREG, D
MOVEM  AREG, TEMP_2
MOVER  AREG, E
ADD    AREG, F
MULT   AREG, TEMP_2
ADD    AREG, TEMP_1
MOVEM  AREG, TEMP_3
MOVER  AREG, G
MULT   AREG, H
ADD    AREG, TEMP_3
```

```
MOVER  AREG, A
MULT   AREG, B
MOVEM  AREG, TEMP_1
MOVER  AREG, C
MULT   AREG, D
MOVEM  AREG, TEMP_2
MOVER  AREG, E
ADD    AREG, F
MULT   AREG, TEMP_2
ADD    AREG, TEMP_1
MOVEM  AREG, TEMP_1
MOVER  AREG, G
MULT   AREG, H
ADD    AREG, TEMP_1
```

Using a stack of temporary locations

Intermediate Code

Postfix notation

- a + b * c + d * e ↑ f
- a b c * + d e f ↑ * +
- Operators can be evaluated in the order in which they appear in the postfix string
- Order of operators in the postfix string matches their bottom-up evaluation order

Postfix string is processed in LR manner

- If the next symbol is an operand

  Build operand descriptor and push into the stack
- If the next symbol is an operator with arity k

  k operand descriptors are popped off the stack and generate instructions to evaluate the operator. Push the descriptor of the partial result into the stack

Triples & Quadruples

- Triples

triple #	Operator	Operand 1	Operand 2
1	*	b	c
2	+	a	1
3	↑	e	f
4	*	d	3
5	+	2	4

- Quadruples

quaduple #	Operator	Operand 1	Operand 2	Result name
1	*	b	c	t_1
2	+	a	t_1	t_2
3	↑	e	f	t_3
4	*	d	t_3	t_4
5	+	t_2	t_4	t_5

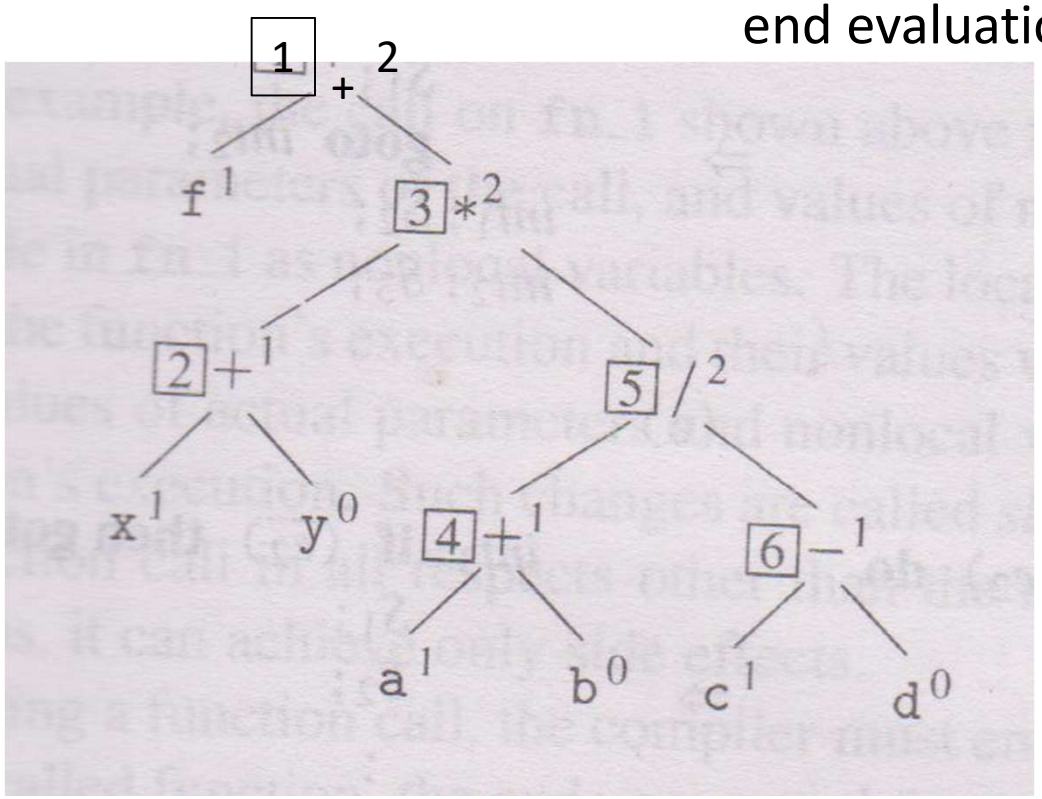
Expression Tree

- AST that represents the structure of an expression
- To determine a desirable evaluation order

1. Make a bottom-up traversal of the expression tree
   for each node n_i
   (a) if n_i is a leaf node then
   if n_i is the left operand of its parent then $RR(n_i) = 1$
   else $RR(n_i) = 0$
   (b) if n_i isn't a leaf node then
   if $RR(l\_child_{ni}) \neq RR(r\_child_{ni})$ then
   $RR(n_i)=max(RR(l\_child_{ni}), RR(r\_child_{ni}))$
   else $RR(n_i)=RR(l\_child_{ni})+1$
2. Call procedure evaluation_order

Cont…

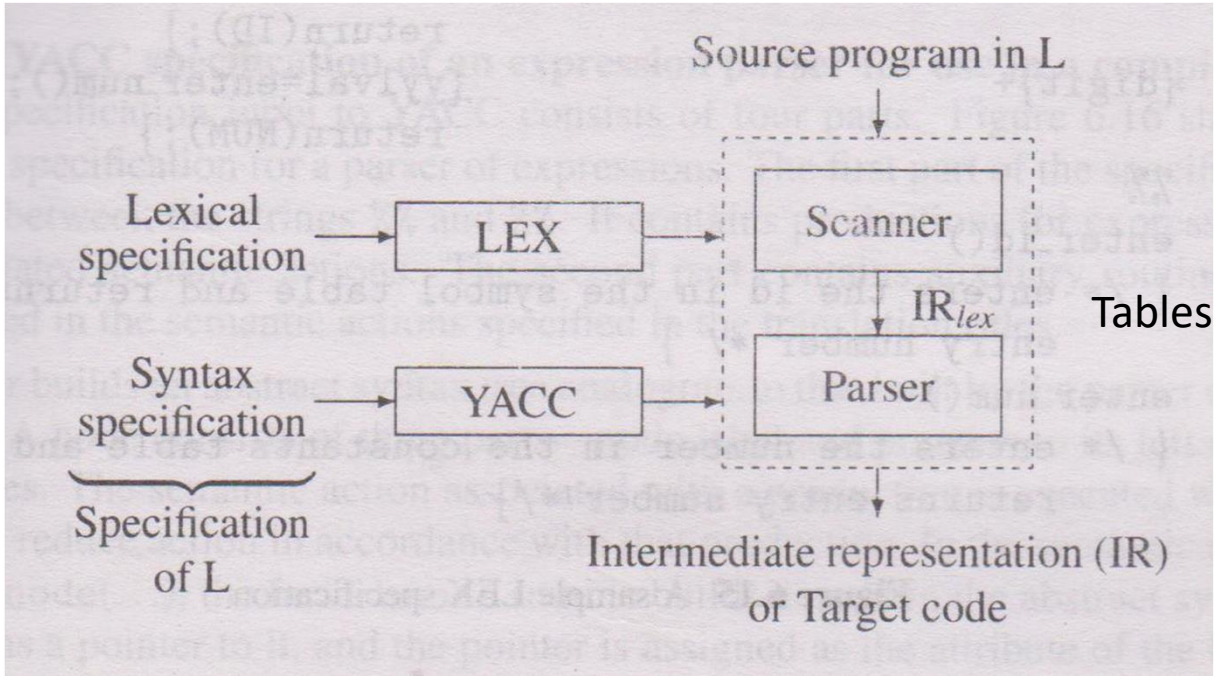
```
procedure evaluation_order(node)
    if node is not a leaf node then
        if RR(l_child_node) ≤ RR(r_child_node) then
            evaluation_order(r_child_node)
            evaluation_order(l_child_node)
        else
            evaluation_order(l_child_node)
            evaluation_order(r_child_node)
    print node
end evaluation_order
```



f+(x+y)*((a+b)/(c-d))

Language Processor Development Tools

- LEX, YACC
- Specification input consists of Translation rules: <string specification> {<semantic action>}
- When a string in the source program matches <string specification>, the <semantic action> will be executed



LEX

- It generates scanner (a C program) of language L and also incorporates the code for semantic actions
- When the scanner is executed, it would recognize lexical units and invoke code for semantic action

```
%{
letter                                  [A-Za-z]
Digit                                   [0-9]
}%
%%          Translation rules
begin                                   {return(BEGIN);}
End                                     {return(END);}
":="                                    {return(ASGOP);}
{letter} ({letter}|{digit})*            {yylval = enter_id();  return(ID);}
{digit}+                                {yylval = enter_num();  return(NUM);}
%%          Semantic actions
enter_id()
{/* enters the id in the symbol table and returns entry number */}
enter_num()
{/* enters the number in the constant table and returns entry number */}
```

Symbols used

LEX specification

YACC

- It generates LALR(1) bottom-up parser from productions of a grammar
- For a shift action, it would invoke the scanner to obtain the next token and continue the parse by using that token
- While performing a reduce action in accordance with a production, it would perform semantic action (build IR or target code)

```
 %%        Productions
E : E + T          {$$ = build_node('+',$1,$3)}
  | T              {$$ = $1}
T : T * V          {$$ = build_node('*', $1,$3)}
  | V              {$$ = $1}
V : id             {$$ = build_node($1,nil,nil)}
%%        semantic actions
build_node(node_label,pointer_1, pointer_2)
{ /* Build a node and return a pointer to it */ }
```

Cont…

- a+b*c
- Abstract syntax tree

build_node(id#1,nil,nil);

build_node(id#2,nil,nil);

build_node(id#3,nil,nil);

build_node(*,n_2,n_3);

build_node(+,n_1,n_4);

Control Structures

- Control transfer (conditional/unconditional goto), conditional execution, iteration control, procedure calls

```
                                                {instructions for ē1}
if (e1) then        if (ē1) then int1;          BC ...., int1 {Branch if true}
    S1;             S1;                         {instructions for S1}
else                goto int2;                  BC   ANY, int2 {Unconditional branch}
    S2;         int1:S2;                    int1:{instructions for S2}
S3              int2:S3;                    int2:{instructions for S3}
```

```
while (e2) do                   int3:if (ē2) then goto int4;
    S1;                                 S1;
    :                                   :
    Sn;                                 Sn;
endwhile                            goto int3;
<next statement>                int4:<next statement>
```

Function & Procedure Calls

Components

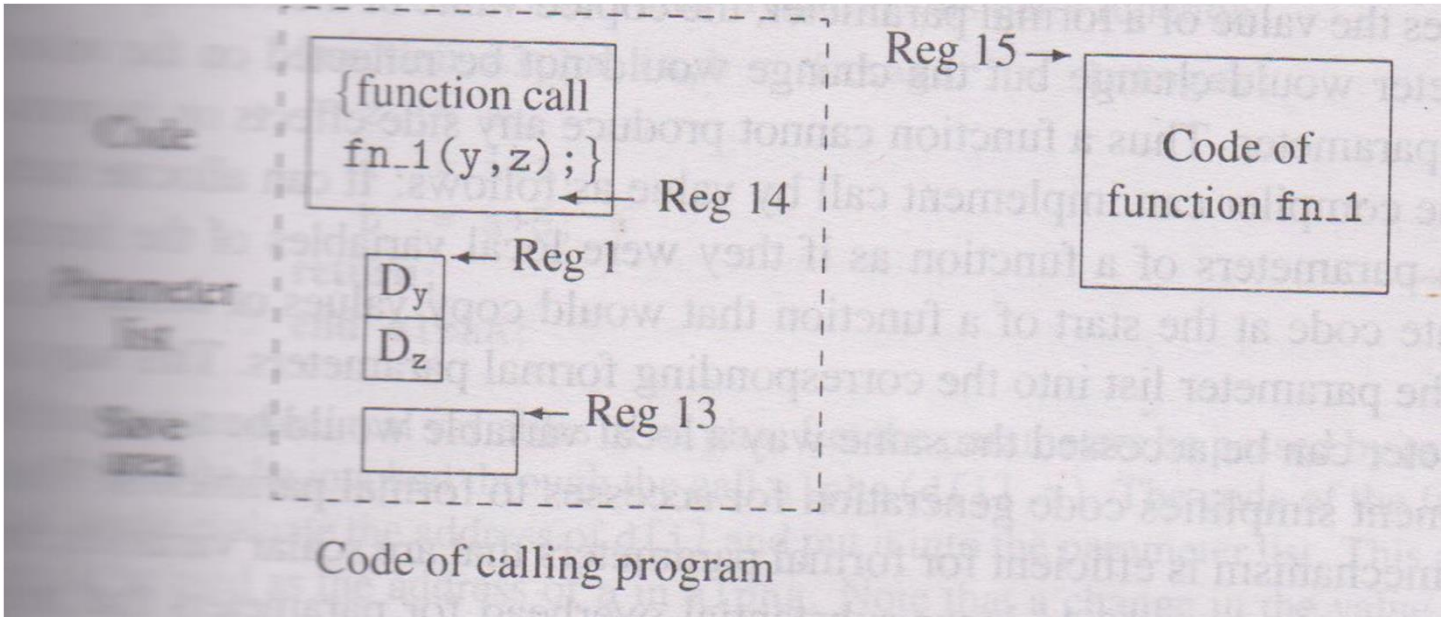
- Parameter list: consists of a descriptor for each actual parameter corresponding to formal parameter p, ($D_{p)}$
- To compile a reference to p in the function, compiler generates code to access D_p and use it to access the actual parameter
- Save area: to save the contents of CPU registers before begin the execution of called function
- Calling conventions

Calling Conventions

Static memory allocation

- Addresses of function, parameter list (r_{par_list}) and save area would be contained in registers
- $(d_{Dp})_{par\_list}$ – displacement of D_p in the parameter list. It is stored in symbol table entry of formal parameter p
- Address of D_p: $<r_{par\_list}> + (d_{Dp})_{par\_list}$

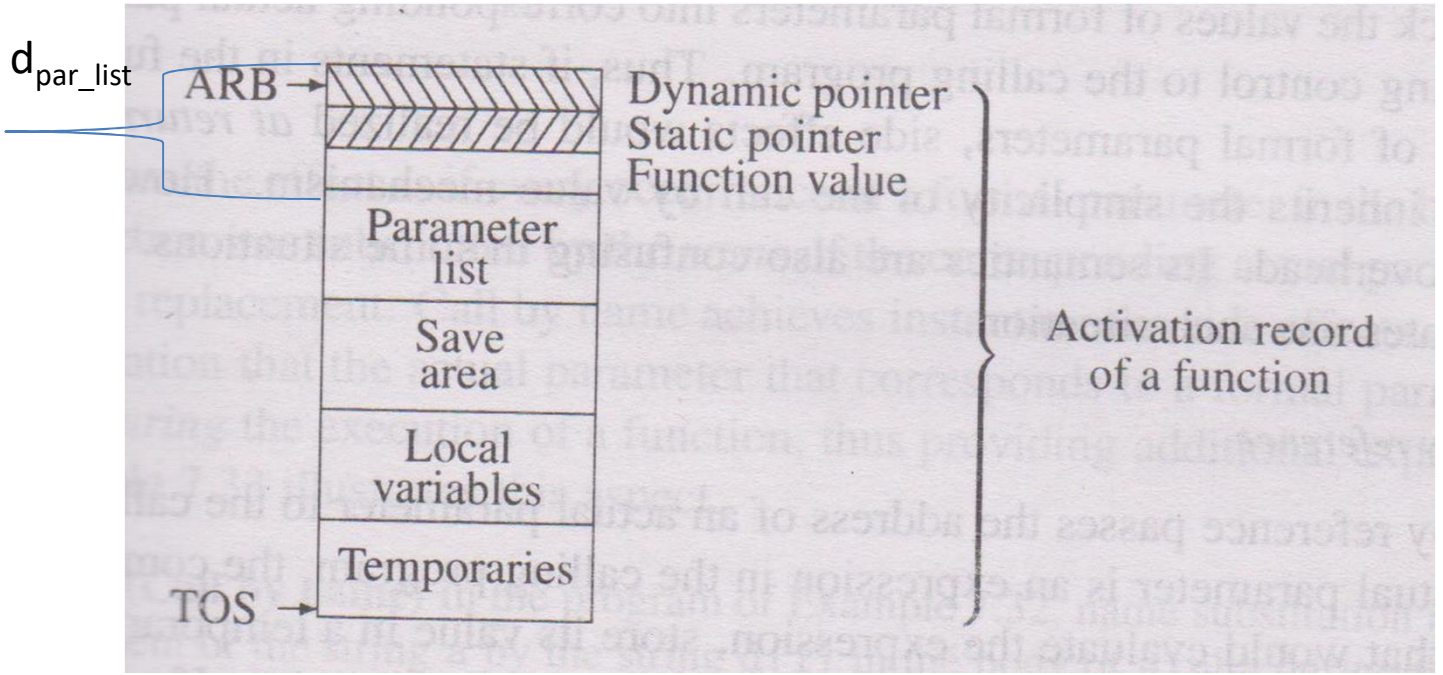
Register No	Purpose
0	To hold return value
1	Address of parameter list
13	Address of save area
14	Return address
15	Address of function



Cont…

Dynamic memory allocation

- Address of D_p : <ARB> + $(d_{Dp})_{AR}$

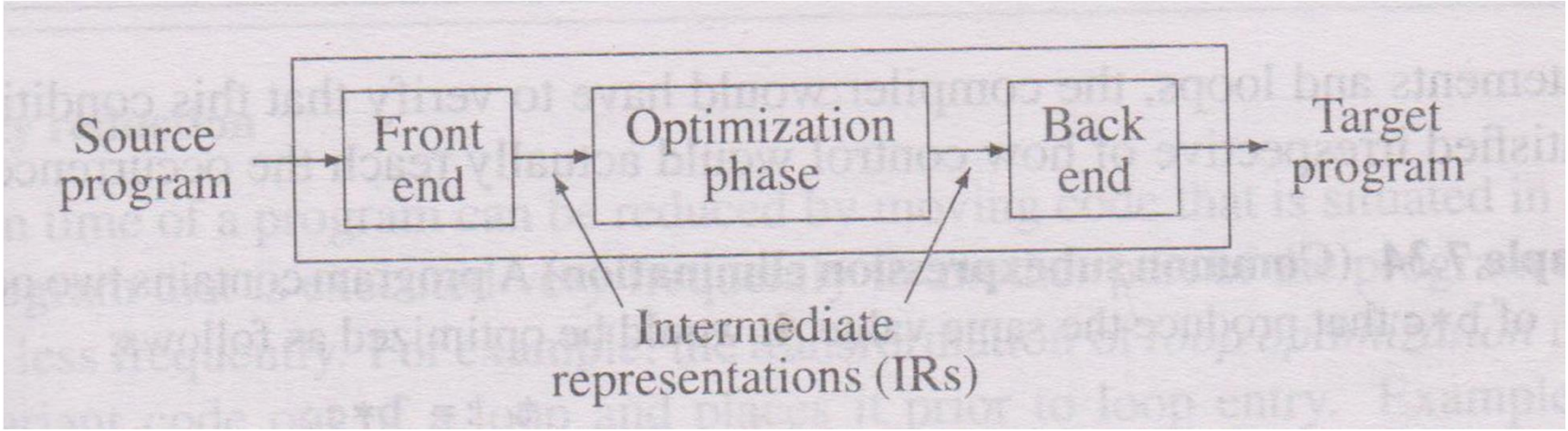


Parameter Passing Mechanisms

- Call by value (C): No side effect on parameters
- Allocate memory to formal parameters and generate code at the start of a function to copy values of actual parameters into formal parameters
- Call by value-result (Ada): copy back the values of formal parameters to actual parameters
- Side effects would be realized at return, incurs more overhead
- Call by reference (Pascal): passes address of an actual parameter to called function
- Side effect would be produced instantaneously
- Call by name (Algol-60): substitute name of formal parameter by actual parameter through string replacement
- Produces instantaneous side effects

Code Optimization

- To improve execution efficiency of a program through
  - Elimination of redundancies in a program
  - Rearrangement of computations in a program
- Optimization phase analyzes IR, performs optimizations and generates an IR of the optimized program. Higher cost of compilation
- Optimization techniques are independent of both PL and target machine



Optimizing Transformations

- A rule for rewriting a segment of a program without changing its meaning to improve the execution efficiency
- Compile time evaluation: need not generate code

  A = 45/11 (constant folding)
- Elimination of common subexpression: occurrences of expressions in a program that yield the same value
- Dead code: code that can be omitted from a program without affecting its results
- Frequency reduction: loop optimization
- Strength reduction: replaces time consuming operation in an expression by a faster operation (only to integer operands)

```
for I = 1 to 10 do
begin
    .....
    k = I * 5;
    .......
end       ;
```

```
temp = 5;
for I = 1 to 10 do
begin
.........
k = temp;
                .........
temp = temp + 5;
end;
```

Local Optimization

- Scope of local optimization is a basic block – sequential section of code segment
- Basic block: it is a sequence of program statements (s_1, s_2, …, s_n) such that only s_n can be a transfer of control statement and only s_1 can be the destination of a transfer of control statement
- Value number provides a simple means of checking an instance of common subexpression
- Value number of each variable is initialized with 0 in symbol table
- When compiler encounters an assignment statement for the variable, it stores the serial number of that statement as the new value number
- Constant propagation assign value number –n (entry number of a constant in constant table) to the variable

stmt no.	*statement*
14	`g := 25.2;`
15	`x := z+2;`
16	`h := x*y+d;`
..	...
34	`w := x*y;`

Symbol table

Symbol	...	*Value number*
y		0
x		15
g		14
z		0
d		5
w		0

Quadruples table

		Operand 1		*Operand 2*			
	Operator	*Operand*	*Value no.*	*Operand*	*Value no.*	*Result name*	*Use flag*
20	:=	g	–	25.2	–	t_{20}	*f*
21	+	z	0	2	–	t_{21}	*f*
22	:=	x	0	t_{21}	–	t_{22}	*f*
23	*	x	15	y	0	t_{23}	*ft*
24	+	t_{23}	–	d	5	t_{24}	*f*
	⋮						
57	:=	w	0	t_{23}	–	t_{57}	*f*

Global Optimization

- It is performed over a procedure/function
- A program is represented in the form of control flow graph
- A CFG of a program P is a directed graph $G_P = (N, E, n_o)$ where

N : set of blocks in P

E : set of directed edges where an edge (b_i, b_j) indicates that control can flow from the last statement of block b_i to the first statement of block b_j during program's execution

n_o : start node of P

Control Flow Analysis

- To collect information concerning the structure of a program
- Predecessors (b_i) and successors (b_j) of an edge (b_i, b_j) in E
- Paths: sequence of edges
- Ancestors (b_i) and descendants (b_j): if a path exist from node b_i to node b_j
- Dominators and post-dominators: node b_i is a dominator of node b_j if every path from n_o to b_j passes through b_i. b_i is a post-dominator of b_j if every path from b_j to an exit node (no successors) of the program graph passes through b_i
- Loops and nesting of loops

Data Flow Analysis

- To collect information concerning use of data. i.e., values of variables and expressions in the program (data flow information)

Data flow concept	Optimization in which used
Available expression	Common subexpression evaluation
Live variable	Dead code elimination
Reaching definition	Constant and variable propagation

- Available expression: an expression *e* is available at a program point p_i if a value that is equivalent to its value would have been computed before program execution reaches p_i

Cont…

- Expression *e* is available at exit of basic block b_i if

  b_i contains an evaluation of e that isn't followed by assignments to any operands of e, ($Eval_i$), or

  the value of e is available at entry to b_i and b_i doesn't contain assignments to any operands of e
- Expression *e* is available at entry to b_i if it is available at the exit of each predecessor of b_i in G_P
- Global properties (data flow equations): Data flow analysis is the process of solving these equations

$Avail\_in_i = \prod_{\text{all bj in pred(bi)}} Avail\_out_j$ (all paths concept)

$Avail\_out_i = Eval_i + Avail\_in_i \,.\, \neg Modify_i$ (forward data flow concept)

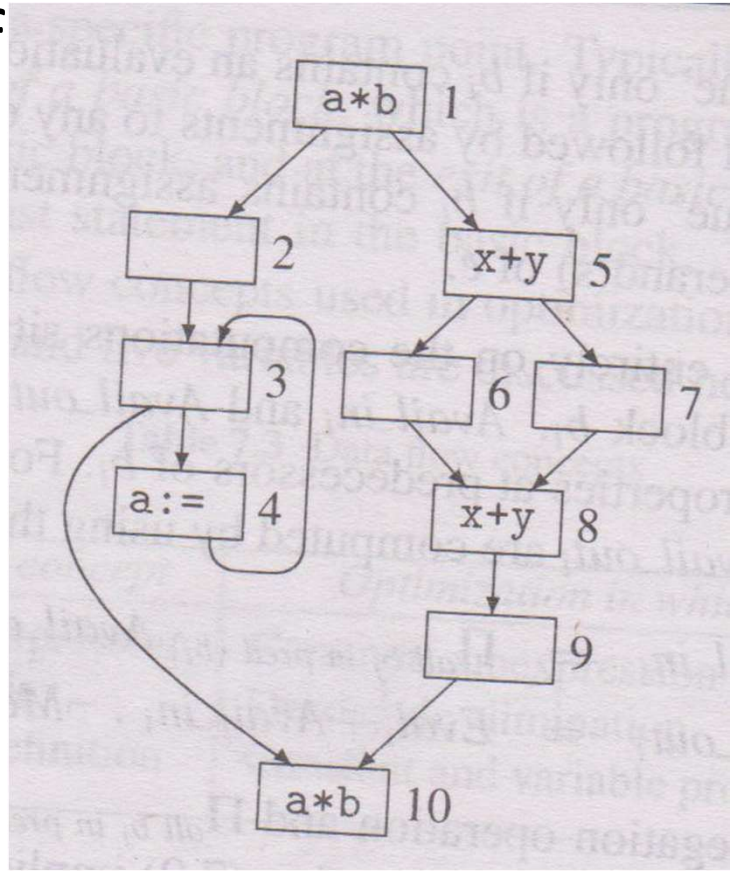
- Local properties: $Eval_i$, $Modify_i$ ('true' if b_i contains assignment to some operand of e)

Cont…

- Evaluation of expression e is eliminated from a block b_i if

$Avail\_in_i$ = true, and

Evaluation of e in b_i isn't preceded by an assignment to any of its operands



Live Variable

- Variable *var* is live at the entry of basic block b_i if ($Live\_in_i$)

  b_i contains a use of *var* that isn't preceded by assignment to *var*, (Ref_i) or

  var is live at the exit of b_i and b_i doesn't contain assignment to *var*

- *var* is live at exit of b_i if it is live at the entry of some successor of block b_i ($Live\_out_i$)

$Live\_in_i = Ref_i + Live\_out_i.\neg Def_i$ (backward data flow concept)

$Live\_out_i = \Sigma_{\text{all bj in succ(bi)}} Live\_in_i$ (any path concept)

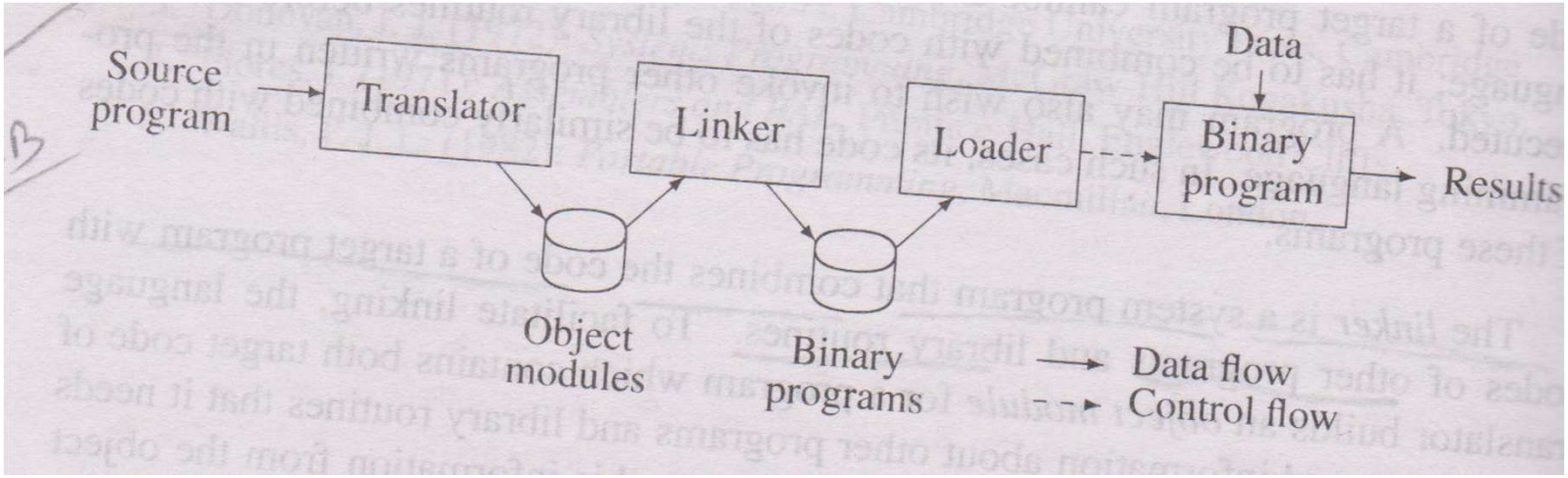
Linkers & Loaders

- Linker is a system program that combines the code of a target program with codes of other programs and library routines
- Object module contains both target code and information about other programs and library routines that it needs to invoke
- Linker extracts information from the object module, locates needed programs and routines and combines them with the target code of the program to produce a program in the machine language called binary program that can execute without the assistance of any other program
- Loader is a system program that loads a binary program into memory for execution
- Steps in program execution: Translation, Linking, Relocation (action of changing memory addresses used in the program code so that it can execute correctly in the allocated memory area), Loading
- Translated origin: address of first memory word of the target program
- Execution start address: address of the instruction from which execution of program must begin

Cont…

Reasons for changing the origin of program

- Same set of translated addresses may have been used in many object modules
- OS may require that a program should execute from a specific area of memory



Program Relocation

- It is the action of modifying the addresses used in the address sensitive instructions of a program such that the program can execute correctly from the designated area of memory
- Relocation factor of program P

  $relocation\_factor_P = l\_origin_P - t\_origin_P$

  $l_{symb} = t_{symb} + relocation\_factor_P$
- If linked origin ≠ translated origin, then relocation by linker
- If load origin ≠ linked origin, then relocation by loader
- Absolute loaders don’t perform relocation. i.e., load origin = linked origin

Linking

- It is the action of putting the correct linked addresses in those instructions of a program that contain external references
- ENTRY statement lists public definitions in a program unit that may be referenced by other program units
- EXTRN statement lists the symbols defined in another program to which external references are made in the program unit

```
        ORIGIN  500
        ENTRY   TOTAL
        EXTRN   MAX, ALPHA
        READ    A           500)  + 09  0  540
LOOP                        501)
        .......
        MOVER   AREG, ALPHA 518)  + 04  1  000
        BC      ANY, MAX    519)  + 06  6  000
        ..........
        BC      LT, LOOP    538)  + 06  1  501
        STOP                539)  + 00  0  000
A       DS      1           540)
TOTAL   DS      1           541)
        END
```

```
        START   200
        ENTRY   ALPHA
        ........
ALPHA   DS   25          231) + 00 0 025
        END
```

Cont…

- A binary program is a machine language program comprising a set of program units SP such that for every P_i in SP

  P_i has been relocated to the memory area whose starting address matches its linked origin, and

  Each external references in P_i has been resolved

Object module consists of

- Header contains translated origin, size and execution start address
- Machine language program
- Relocation table (RELOCTAB) contains translated address of address sensitive instruction
- Linking table (LINKTAB) contains symbol, type (PD or EXT) and translated address

Relocation Algorithm

- Machine language program is loaded into work area and relocates address sensitive instructions in it by processing entries of RELOCTAB

1. program_linked_origin = <link origin> from linker command
2. For each object module mentioned in the linker command

(a) t_origin = translated origin of the object module

(b) relocation_factor = program_linked_origin – t_origin

(c) Read machine language program contained in program component of object module into the work _area

(d) Read RELOCTAB of the object module

(e) For each entry in RELOCTAB

(i) translated_address = address found in RELOCTAB entry

(ii) address_in_work_area = address of work area + translated_address – t_origin

(iii) add relocation_factor to operand address found in the word that has the address address_in_work_area

(f) program_linked_origin = program_linked_origin + OM_size

Linking Scheme

- Linker processes LINKTABs of all object modules and copy the information about PD found in them into NTAB
- NTAB: symbol (name of an external reference or object module), linked_address (linked address of the symbol or linked origin of object module)
- Pass I (step 2): algorithm reads program component of each object module into the work area, computes the linked origin of each object module and linked address of each PD, and enters this information in NTAB
- Pass II (step 3): algorithm resolves each external reference by computing address of the word in work area and inserting linked address of symbol in it

Algorithm

1. Program_linked|_origin = <link origin> from linker command
2. For each object module mentioned in the linker command

(a) t_origin = translated origin of the object module

OM_size = size of object module

(b) relocation_factor = program_linked_origin – t_origin

(c) Read machine language program contained in program component of object module into work_area

(d) Read LINKTAB of the object module

(e) Enter (object module name, program_linked_origin) in NTAB

(f) For each LINKTAB entry with type=PD

name=symbol field of the LINKTAB entry

linked_address=translated_address+relocation_factor

Enter(name, linked_address) in a new entry of NTAB

(g) program_linked_origin=program_linked_origin+OM_size

Cont…

3. For each object module mentioned in the linker command

   (a) t_origin=translated origin of the object module

   program_linked_origin=linked_address from NTAB

   (b) For each LINKTAB entry type=EXT

   (i) address_in_work_area=address of work area+program_linked_origin-<link origin> in linker command+translated_address-t_origin

   (ii) Search the symbol found in the symbol field of the LINKTAB entry in NTAB and note its linked address. Copy this address into the operand address field in the word that has the address address_in_work_area

Self Relocating Programs

- Program can be loaded in any area of memory. It would perform its own relocation and execute correctly in that memory area
- Components required: a table containing information about address sensitive instructions
- Relocating logic – code to perform relocation of address sensitive instructions. Start address of the relocating logic is specified as the execution start address of the program
- Relocatable program: Program can relocated by a linker or loader to have linked address or load address that matches the start address of the specified area of memory. E.g., object module
- Non-relocatable program: Program can't be executed in any memory area other than the area starting from its translated origin. E.g., hand-coded machine language program

Overlays

- Part of a program that has same load origin as some other parts of the program
- Overlay structured program: A program containing overlays. Memory requirement can be reduced
- It consists of a permanently resident part (root) and a set of overlays
- Root is loaded in memory and given control of execution. Whenever it refer to a function or data located in an overlay, it invokes overlay manager (organizes loading of the overlay)
- A program designer designs the overlay structure and conveyed to the linker

Dynamic Linking

- Linker is invoked when an unresolved external reference is encountered during the execution of a program
- Procedure: Each program is first processed by static linker.
- Static linker links each external reference to a dummy module
- Dummy module calls the dynamic linker and pass name of external symbol to it
- Dynamic linker maintains a table containing public definitions and their load address. If the external symbol is present in the table, use its load address. Otherwise search the library of object modules and find the object module containing public definition of that symbol. Link this object module to the binary program and add information about the symbol to the table
- Benefits: modules that aren't invoked during execution of a program needn't be linked at all
- If the module is already available in the memory as part of the execution of another program, same copy can be used, thus saving the memory
- Dynamically Linked Libraries (DLL) use some of these features

Loaders

Absolute loader

- It loads a binary program containing the following into memory for execution
- Header record showing load origin, length, and load time execution start address of the program
- Sequence of binary image records containing program's code. Each record contains a part of the program's code – sequence of bytes, load address of first byte of this code, number of bytes
- Absolute loader loads the binary image record into the specified location. At end it transfers control to execution start address of the program

Bootstrap loader loads OS (bootstrap loading) when power is switched on. It loads loader also and passes control to it

Relocating Loader

- It loads a program containing the following into a designated area of memory, relocates it, and passes control to it for execution
- Header record showing linked origin, length, and linked execution start address of the program
- Sequence of binary image records containing program's code. Each record contains a part of the program's code – sequence of bytes, linked address of first byte of this code, number of bytes
- A table giving linked addresses of address sensitive instructions in the program
- Relocating loader is invoked with a file containing the program and its load origin