Module 6

Code Optimization:

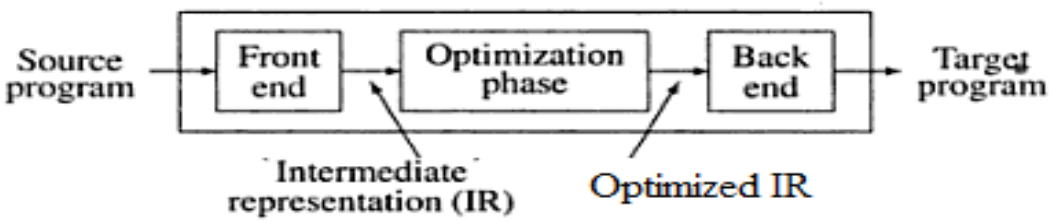
Principal sources of optimization, Optimization of Basic Blocks

Code Generation:

Issues in the design of a code generator. The target machine, A simple code generator

Code optimization

- CO aims at improving the execution efficiency of a program
- This is achieved in 2 ways:
  - 1. Redundancies in a program are eliminated
  - 2. Computations in a program are rearranged or rewritten to make it execute efficiently



Schematic of an optimising compiler

- Optional Phase
- CO must not change the meaning of the program
- 2 points concerning the scope of optimization:
  - 1. CO aims at improving a prgm, rather than improving the algm used in the prgm. Thus replacement of an algm by a more efficient algm is beyond the sope of CO.
  - 2. Efficient code generation for a specific target machine (eg: by fully exploiting its instruction set) is also beyond the scope of CO.
- Compiler was found to consume 40% extra compilation time due to optimization

- The optimized program occupied 25% less storage and executed 3 times faster than unoptimized prgm.

Basic Block

- A **basic block(BB)** is a sequence of pgm statements (s1,s2,…,sn) such that only sn can be a transfer of control statement and only s1 can be the destination of a transfer of control statement
- A BB is a prgm segment with single entry point.
- If control reaches statement s1 during prgm execution, all statements s1,s2…sn will be executed. The 'essentially sequential' nature of a BB simplifies optimization
- Ex. for Basic Block:

```
        a := x*y;
        - - -
        b := x*y;
lab_i : c := x*y;
```

Program Flow Graph (PFG)

- A prgm can be represented in the form of a PFG
- A PFG for a prgm P is a directed graph **Gp = (N, E, n0)** where

N: set of basic blocks in P

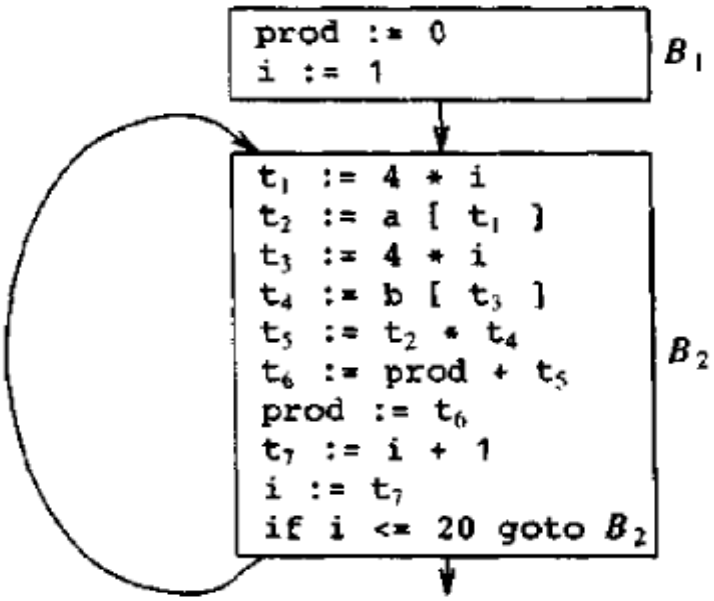
E: set of directed edges (bi,bj) indicating the possibility of control flow from the last statement of bi (the source node) to the first statement of bj (the destination node)

n0: start node of P

- The nodes of PFG are basic blocks.
- There is a directed edge from block B1 to block B2 if B2 can immediately follow B1 in some execution sequence. that is, if
  - 1. There is a conditional or unconditional jump from last statement of B1 to the first statement of B2 or
  - 2. B2 immediately follows B1 in the order of prgm, and B1 does not end in an unconditional jump.
  - Here, B1 is a predecessor of B2 and B2 is a successor of B1.

Program Segment

```
begin
     prod := 0;
     i := 1;
     do begin
         prod := prod + a[i] * b[i];
         i := i + 1
     end
     while i <= 20
end
```



Flow graph for program.

```
(1)    prod := 0
(2)    i := 1
(3)    t1 := 4 * i
(4)    t2 := a [ t1 ]
(5)    t3 := 4 * i
(6)    t4 := b [ t3 ]
(7)    t5 := t2 * t4
(8)    t6 := prod + t5
(9)    prod := t6
(10)   t7 := i + 1
(11)   i := t7
(12)   if i <= 20 goto (3)
```

Three Address Code

Optimizing Transformation

- It is a rule for rewriting a segment of a program to improve its execution efficiency

- Few optimizing transformations commonly used in compilers are:
  - 1. Compile time evaluation
  - 2. Elimination of common subexpression
  - 3. Dead code elimination
  - 4. Frequency Reduction
  - 5. Strength Reduction

1. Compile time Evaluation

- Execution efficiency can be improved by performing certain actions specified in a pgm during compilation itself
- This eliminates the need to perform during execution time, thereby reducing the execution time of the program
  Ex: Constant Folding
- When all operands in an operation are constants, the operation can be performed at compile time
- Ex: a = 100 /25 can be replaced by a = 4 at compilation time itself

2. Elimination of Common Subexpression

- CSE are occurrences of expressions yielding same value.
- Such expressions are also called as **equivalent expressions**
- Ex:

```
                        t := b*c;
a := b*c                a := t;
- - -          ⇒       - - -
x := b*c+5.2;           x := t+5.2;
```

Here CS_i contains the two occurrences of `b*c`. The second occurrence of `b*c` can be eliminated because the first occurrence of `b*c` is always evaluated before the second occurrence is reached during execution of the program. The value computed at the first occurrence is saved in `t`. This value is used in the assignment to `x`.

- The optimization is implemented as follows:
  - Expressions yielding same value are identified. These can be easily identified using quadruples or triples
  - Their equivalence is determined by considering whether their operands have same values in all occurences
  - Such expressions can be eliminated

3. Dead Code Elimination

- Code which can be omitted from a pgm without affecting its results is called **dead code**
- Dead code is detected by checking whether the value assigned in an assignment statement is used anywhere in the pgm

> An assignment `x := <exp>` constitutes dead code if the value assigned to x is not used in the program, no matter how control flows after executing this assignment. Note that *<exp>* constitutes dead code only if its execution does not produce side effects, i.e. only if it does not contain function or procedure calls.

4. Frequency Reduction

- Execution time of a pgm can be reduced by moving code from a part of a prgm which is executed very frequently to another part of the pgm which is executed fewer times
- Ex: Loop optimization – loop invariant code motion

```
for i := 1 to 100 do
begin
   z := i;
   x := 25*a;
   y := x+z;
end;
```

⇒

```
x := 25*a;
for i := 1 to 100 do
begin
   z := i;
   y := x+z;
end;
```

Here `x := 25*a;` is loop invariant. Hence in the optimized program it is computed only once before entering the **for** loop. `y := x+z;` is not loop invariant. Hence it cannot be subjected to frequency reduction.

5. Strength Reduction

- This optimization replaces the occurrence of a time consuming operation (‘**high strength operation**’) by an occurrences of a faster operation(‘**low strength operation’**).
- Ex: replacement of a multiplication by an addition
- Here, high strength operator * in i*5 occurring inside loop is replaced by a low strength operator + in itemp+5

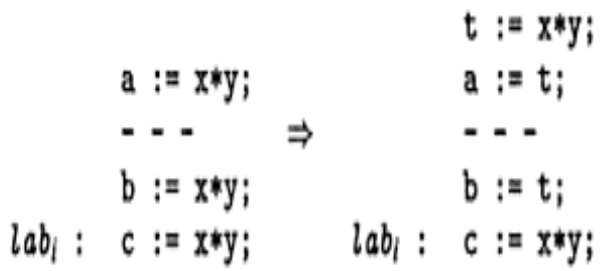
```
                                    itemp := 5;
for i := 1 to 10 do                 for i := 1 to 10 do
begin                               begin
   - - -                               - - -
   k := i*5;                           k := itemp;
   - - -                      ⇒        - - -
                                       itemp := itemp+5;
end;                                end;
```

Local and Global Optimization

- Optimizing a prgm can be classified into :
  - 1. Local Optimization : The optimizing transformations are applied over small segments of a pgm consisting of a few statements
  - 2. Global Optimization : applied over a program unit (that is over a function or procedure)
- Local optimization can be considered as a preparatory phase for global optimization

Local Optimization

- Scope of LO is a basic block.
- A **basic block(BB)** is a sequence of pgm statements (s1,s2,…,sn) such that only sn can be a transfer of control statement and only s1 can be the destination of a transfer of control statement
- Cost of LO is low, bcz the sequential nature of BB simplifies the analysis needed for optimization.
- The benefits by LO are limited bcz certain optimizations(ex: loop optimization) are beyond the scope of LO.
- Ex for Local Optimization in BB:



where lab_i is a label. Local optimization identifies two basic blocks in the program. The first block extends up to the statement `b := x*y;`. Its optimization leads to elimination of the second occurrence of `x*y`. The third occurrence is not eliminated because it belongs to a different basic block. If the label lab_i did not exist, the entire program segment would constitute a single basic block and the third occurrence `x*y` can also be eliminated during local optimization.

Value Numbering

- Value number provides a simple method to determine if 2 occurrences of an expression in a BB are equivalent
- A value number ***vnalpha*** is associated with variable alpha.
- It identifies the last assignment to **alpha** processed so far.
- The value number of a variable **alpha** changes on processing an assignment statement ***alpha* = …**
- Using value numbering:
  - Local Common subexpression elimination
  - Constant propagation
  - Constant Folding

CSE using Value Numbering

- Two expressions ei and ej are equivalent if they are congruent and their operands have same value numbers.
- All expressions in a BB are numbered.
- If statement n is the current statement being processed, is an assignment to variable **alpha** we set ***vnalpha*** to **n**.
- A new field is added to each symbol table entry to hold value number.

- IC we consider here is quadruple.
    - Each operand field in a quadruple holds the pair ***(operand, value number)***
    - A boolean ***flag*** save is associated with each quadruple to indicate whether its value should be saved for use elsewhere in the prgm.
    - ***flag*** is initialized to false in every new quadruple entered in the table
- While forming a quadruple for the expression ***e***, the value number of its operands are copied from the symbol table.
- The new quadruple is now compared with all existing quadruples in IC.
- Existence of a matching quadruple ***qi*** indicates the current occurrence of expression ***e*** has the same value as a previous occurrence represented by ***qi***.
- If a match is found, newly generated quadruple is not entered in IC.
- Instead, result name of ***qi*** is used where e is needed.
- In effect, this occurrence of ***e*** is identified as common subexpression and it is eliminated from the prgm.
- The result name of ***qi*** should now become a compiler generated temporary variable.
- This requirement is noted by setting the save ***flag*** of ***qi*** to true.
- During code generation, this flag is checked to see if the value of ***qi*** needs to be saved in a temporary location.
- Example:

  Program Segment

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

	Operator	Operand 1		Operand 2		Result name	Use flag
		Operand	Value no.	Operand	Value no.		
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*

- For explanation of above example, please refer page 205 (Systems programming and OS – Dhamdhere)

Constant Propagation and constant folding using Value Numbering:

- When an assignment of the form ***var = const*** is encountered, enter const into a table of constants, say in entry n, and associate the value number '-n' with var.
- Constant propagation and folding is implemented while generating a quadruple if each operand is either a constant or has a negative value number.
- **Ex:**

In the following program, variable a is given a negative value number, say '–10', on processing the first statement.

```
a := 27.3;          a := 27.3;
- - -        ⇒     - - -
b := a*3.0;         b := 81.9;
```

This leads to the possibility of constant propagation and folding in the third statement. The value of a, viz. '27.3', is obtained from the 10th entry of constants table. Its multiplication with 3.0 yields 81.9. This value is treated as a new constant and a quadruple for `b := 81.9` is now generated.

Global Optimization

- Applied over a program unit (that is over a function or procedure)
- More analysis is required to establish the feasibility of optimization
- For this:
  - Control Flow Analysis
  - Data Flow Analysis
- Ex: Global Common Subexpression Elimination

Control Flow Analysis

- Analyses a prgm to collect information concerning its structure (eg: presence of nesting loops in prgm)
- Some control flow concepts:
  - 1. **Predecessors and successors**: If (bi,bj) ε E, bi is a predecessor of bj and bj is the successor of bi.
  - 2. **Paths:** A path is a sequence of edges such that destination node of one edge is the source node of following edge
  - 3. **Ancestors and Descenders:** If a path exists from bi to bj, bi is an ancestor of bj and bj is a desendant of bi
  - 4. **Dominators and post-dominators:** Block bi is a dominator of block bj if every path from n0 to bj passes through bi. bi is a post dominator of bj if every path from bj to an exit node passes through bi

Data Flow Analysis

- Analyse the data in a program to collect information for the purpose of optimization
- These information are called **data flow information**
- It is computed at the entry and exit of each BB in a pgm
- Design of global optimization phase begins with the identification of an appropriate **data flow concept** to support the application of each optimizing transformation

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

Global CSE

- If some expression **x*y** occurs in a set of basic blocks(**SB**) of program **P**, its occurrence in block **bj Ɛ SB** can be eliminated if the following 2 conditions are satisfied for every execution of **P**:
  - 1. Basic block **bj** is executed only after some block **bk Ɛ SB** has been executed one or more times
  - 2. No assignments to **x** or **y** have been executed after the last (or only) evaluation of **x*y** in block **bk**.
  - Condition 1 ensures that **x*y** is evaluated before execution reaches block **bj**
  - Condition 2 ensures that the evaluated value is equivalent to the value of **x*y** in block **bj**
  - The optimization is done by saving the value of x*y in a temporary location in all blocks bk which satisfy condition 1
- Consider a subexpression x*y occurring at program point **pi** in BB **bi**. This occurrence can be eliminated if:
  - 1. Above conditions 1 and 2 are satisfied at entry to **bi**
  - No assignments to **x** or **y** precede the occurrence of **x*y** in **bi**
- Data flow concept used – **Available Expression**

- An expression **e** is available at prgm point **pi** if a value equivalent to its value is always computed before prgm execution reaches **pi**
- The availability of an expression at entry or exit of BB bi is computed using the following rules:
  - 1. Expression e is available at the **exit** of bi if: **(Avail_outi)**
  - bi contains an evaluation of e which is not followed by assignments to any operand of e
  - The value of e is available at entry to bi and bi does not contain assignments to any operands of e
  - 2. Expression e is available at **entry** to bi if it is available at the exit of each predecessor of bi in Gp **(Avail_ini)**
- **Evali** : **true** only if exprsn **e** is evaluated in **bi** and none of its operands are modified following the evaluation
- **Modifyi** : **true** only if some operand of **e** is modified in **bi**

$$Avail_in_i = \prod_{b_j \in pred(b_i)} Avail_out_j$$
$$Avail_out_i = Eval_i + Avail_in_i \,.\, \neg Modify_i$$

where $\prod_{b_j \in pred(b_i)}$ is the boolean 'and' operation over all predecessors of b_i.

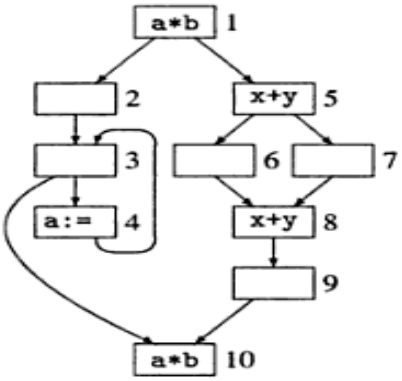
- After solving the system of equations for all blocks, an evaluation of expression **e** can be eliminated from a block **bi** if:
  - 1. **Avail_ini = true** and
  - 2. The evaluation of **e** in **bi** is not preceded by an assignment to any of its operands

Example:

- Available expression analysis for PFG gives the following results:

a*b : *Avail_in* = *true* for blocks 2,5,6,7,8,9
Avail_out = *true* for blocks 1,2,5,6,7,8,9,10

x+y : *Avail_in* = *true* for blocks 6,7,8,9
Avail_out = *true* for blocks 5,6,7,8,9



Global Dead Code Elimination

- Data flow concept – **Live Variable**
- A variable **var** is said to be live at a prgm point **pi** if the value contained in it at **pi** is likely to be used during subsequent execution of the prgm
- If var is not live at pi which a definition var = …, it can be considered as dead code and can be eliminated from the prgm
- The liveness ppty of a variable can be determined as follows:
  - 1. Variable **v** is live at the entry of **bi (Live_ini)** if:

    a. **bi** contains a use of **v** which is not preceded by assignment to **v** or

    b. **v** is live at exit of **bi** and **bi** does not contain assignment to **v**
  - 2. Variable **v** is live at exit of **bi (Live_outi)** if it is live at entry of some successor of **bi** in **Gp**
- Data flow information concerning live variables can be collected using:

$Live_in_i$	:	*var* is live at entry of b_i
$Live_out_i$	:	*var* is live at exit of b_i
Ref_i	:	*var* is referenced in b_i and no assignment of *var* precedes the reference
Def_i	:	An assignment to *var* exists in b_i

$$Live_in_i = Ref_i + Live_out_i \,.\, \neg Def_i$$
$$Live_out_i = \Sigma_{b_j \in succ(b_i)} Live_in_j$$

where $\Sigma_{b_i \in succ(b_i)}$ is the boolean 'or' operation over all successors of b_i.

- With the initialization **Live_ini = Live_outi =false** for all bi
- **Available expression:**
    - **forward data flow concept:** bcz availability at the exit of a node determines availability at the entry of its successors
    - **All path concept:** bcz availability at the entry of a BB requires availability at the exit of all predecessors
- **Live variable:**
    - **Backward data flow concept:** bcz availability at the entry of a block determines availability at the exit of its predessor
    - **Any path concept:** bcz liveness at the entry of one successor is sufficient to ensure liveness at exit of a block

Machine Dependent and Machine Independent Optimization

Machine Independent Optimization

- **Common Subexpression Elimination**
- **Dead Code Elimination**
- **Loop Invariant Code Motion**

Machine Dependent Optimization

- **Register Allocation:**
    - Register allocation refers to the choosing of which values are kept in registers during program execution.
    - Values stored in registers can be accessed more quickly than values stored in other types of memory.
    - Since computers have only a limited number of registers, the efficient use of these registers is important in producing good code.
    - Can be done using: Graph Coloring
- **Instruction Level Parallelism:**
    - Overlap the execution of instructions to improve performance
    - Each step (pipe stage or pipe segment) in the pipeline completes a part of an instruction.
    - Different steps complets different parts of different instructions in parallel.

- The time required between moving an instruction one step down the pipeline is processor cycle.
- The length of the processor cycle is determined by the time required for the slowest pipe stage.
- Designers goal:
    - To balance the length of each pipeline stage.
    - Reduce the average execution time per instruction.
- In RISC Architecture, every instruction(4 bytes) can be implemented in atmost 5 clock cycles. Cycles include:

1.Instruction fetch cycle(IF)

- Send the PC address to memroy and fetch the current instruction from memory.update the PC with next sequential PC by adding 4.

2.Instruction decode/register fetch cycle(ID)

- Decode the instruction and read the registers corresponding to register source in PARALLEL.

3.Execution/Effective address cycle(EX)

- ALU operates on operands perpared in the prior cycle.
- Perform one of 3 operations(depending on instruction type) :
    - 1. MEMORY REFERENCE:
        - EA=BASE REGISRE+OFFSET
    - 2. REGISTER-REGISTER ALU INSTRUCTION:
        - Performs the operation specified by the ALU opcode on the values read from the register.
    - 3. REGISTER-IMMEDIATE ALU INSTRUCTION:
        - Performs the operation specified by the ALU opcode on the first value read from register and the sign extended immediate.

4.Memory Access(MEM)

- For load-> read on memory by using EA

5.Write-Back(WB): Write the result into register, whether it comes from memory system or from ALU.

Basic Issues in Code Generation

- Major Issues:
  - 1. Determination of an evaluation order for the operators in an expression
  - 2. Selection of instructions to be used in target code
  - 3. Use of registers and handling of partial results

1. Determination of an evaluation order for the operators in an expression

- Evaluation order for the operators depends on operator precedence
- An operator which has higher precedence than its neighbours must be evaluated first.
- For this we can use: **Expression trees**

2. Selection of instructions to be used in target code

- Depends on:
  - 1. The type and length of each operand
  - 2. The addressability of each operand (that is, where the operand is located and how it can be accessed).
- For this we can use – **operand descriptor**
- Operand descriptor contains information regarding type, length and addressability of each operand
- Choice of an instruction can be made by analysing an operator and the descriptors of its operator

3. Use of registers and handling of partial results

- A partial result is the value of some subexpressions computed while evaluating an expression
- For efficiency, partial results are stored in CPU Registers as far as possible

- But some of them have to be moved to memory if the number of results exceeds the number of available CPU registers
- The issue is:
  - when and how to move partial results between memory and registers
  - How to know which partial result is contained in a register
- For this we can use – **Register Descriptor**

Operand Descriptor

- It contains following fields:
  - 1. **Attributes**: Contains the subfields type, length and miscellaneous information
  - 2. **Addressability**: Specifies where the operand is located and how it can be accessed. It has two subfields:
    - a) **Addressability Code**: - Takes the values:

      M (operand in memory)

      R (operand in register)

      AR(Address in register)

      AM(Address in Memory)

    - b) **Address**: - Address of a CPU register or memory word
- An operand descriptor is built for id's, constants and partial results
- **descriptor for id** – when id is reduced during parsing
- **descriptor for partial result** – immediately after code is generated
- Operand descriptors are stored in an array called ***Operand_descriptor***

The code generated for the expression a*b is as follows:

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

Three operand descriptors are used during code generation. Assuming a, b to be integers occupying 1 memory word, these are:

1	(int,1)	M, addr(a)	Descriptor for a
2	(int,1)	M, addr(b)	Descriptor for b
3	(int,1)	R, addr(AREG)	Descriptor for a*b

```
%%
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), (addressability_code, address))
                                of operand;
    return i;
 }
```

Skeleton of the code generator

Register Descriptor

- It has 2 fields:
  - 1. Status: Contains ***free or occupied*** to indicate register status
  - 2. Operand descriptor #: If ***status=occupied***, this field contains the descriptor # for the operand contained in the register.

- Register descriptors are stored in an array called ***Register_descriptor.***
- One register descriptor exist for each CPU register.
- Code generated for the expression a*b is:

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

- The register descriptor for AREG after generating above code:

Occupied	#3

This indicates that register AREG contains the operand described by descriptor #3.

Expression Trees

- So far we assumed that operators are evaluated in the order determined by bottom up parser.
- This evaluation order may not lead to the most efficient code for an expression
- So compiler backend must analyze an expression to find the best evaluation order for its operators
- Expression tree is an AST which depicts the structure of an expression
- This representation simplifies the analysis of an expression to determine the best evaluation order
- To determine the best evaluation order for the operators in an expression, 2 step procedure is used:
- 1. Associates a **register requirement label(RR Label)** with each node in the expression.
    - **RR Label** indicates the number of CPU registers required to evaluate the subtree rooted at that node without moving a partial result to memory.
    - Labelling is performed in a bottom up pass of expression tree.

- 2. Analyses the RR labels of the child nodes of a node to determine the order in which they should be evaluated
    - Performed during top down pass

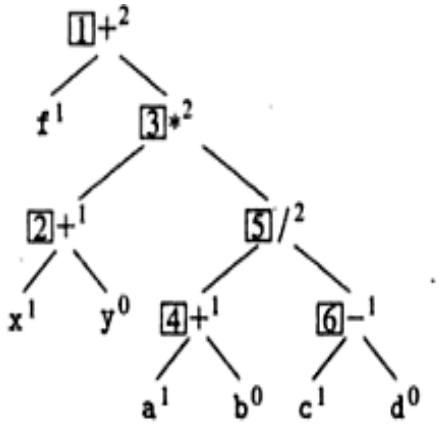
Evaluation order for operators

Algorithm 6.1 (Evaluation order for operators)

1. Visit all nodes in an expression tree in post order (i.e., such that a node is visited *after* all its children).
   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 is not a leaf node then
   If $RR(l_child_{n_i}) \neq RR(r_child_{n_i})$ then
   $RR(n_i) := \max(RR(r_child_{n_i}), RR(l_child_{n_i}))$;
   else $RR(n_i) := RR(l_child_{n_i}) + 1$;
2. Perform the procedure call *evaluation_order* (*root*) (See Fig. 6.23), which prints a postfix form of the source string in which operators appear in the desired evaluation order.

```
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;
```

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



Evaluation order: 6,4,5,2,3,1

Example 6.25 Figure 6.24 shows the expression tree for the string `f+(x+y)*((a+b)/(c-d))`. The boxed numbers indicate operator positions in the source string. The RR label of each node is shown as the superscript of the operand or operator at that node. The evaluation order according to algorithm 6.1 is [6], [4], [5], [2], [3], [1]. This evaluation order is determined as follows: When *evaluation_order* is called with the root node, i.e. operator [1], as the parameter, it decides that the right child of the node should be evaluated before its left child since RR([3]) > RR(node for `f`). This leads to a recursive call with node [3] as the parameter. Here RR([5]) > RR([2]) leads to a recursive call with node [5] as the parameter. At node [5] also decision is made to visit its right child before its left child. This leads to the postfix string `cd-ab+/xy+*f+` in which operators appear in the evaluation order mentioned above.

Generating an instruction

- When an operator **opi** is reduced by the parser, the function **codegen** is called.
- If the descriptors indicate that one operand is in register and the other is in memory, a single instruction is generated
- If both operands are in memory, two instructions will be generated
  - 1. Move one operand into a register
  - 2. Instruction to evaluate opi

Saving partial results

- When an operator opi is to be evaluated,:
  - if all registers are occupied, a register r is freed by copying its contents into a temporary location in memory.
  - Then r is used to evaluate opi
- An array called **temp** is declared to hold partial results
- The partial results saved last is always used before the results saved earlier. So **stack** (**LIFO data structure**) can be used to organize temporary locations
- The use of stack permits reuse of temporary locations, which reduces the number of temporary locations required

Code Generation Routine

```
codegen(operator, opd1, opd2)
{
   if opd1.addressability_code = 'R'
         /* Code generation -- case 1 */
      if operator = '+' generate 'ADD AREG, opd2';
         /* Analogous code for other operators */
   else if opd2.addressability_code = 'R'
         /* Code generation -- case 2 */
      if operator = '+' generate 'ADD AREG, opd1';
         /* Analogous code for other operators */
   else
         /* Code generation -- case 3 */
      if Register_descr.status = 'Occupied'
         /* Save partial result */
         generate  ('MOVEM AREG, Temp[j]');
         j = j + 1;
         Operand_descr[Register_descr.Operand_descriptor#]
                  = (<type>, (M, Addr(Temp[j])));
```

```
            /* Generate code */
            generate 'MOVER AREG, opd1';
            if operator = '+' generate 'ADD AREG, opd2';
              /* Analogous code for other operators */

   /* Common part -- Create a new descriptor
   Saying operand value is in register AREG */
   i = i + 1;
   operand_descr[i] = (<type>,('R', Addr(AREG)));
   Register_descr = ('Occupied', i);
   return i;
}
```

Example: a*b + c*d

- Code generation steps: Superscript of NT shows operand descriptor # and <id>v represents token <id> constructed for symbol v

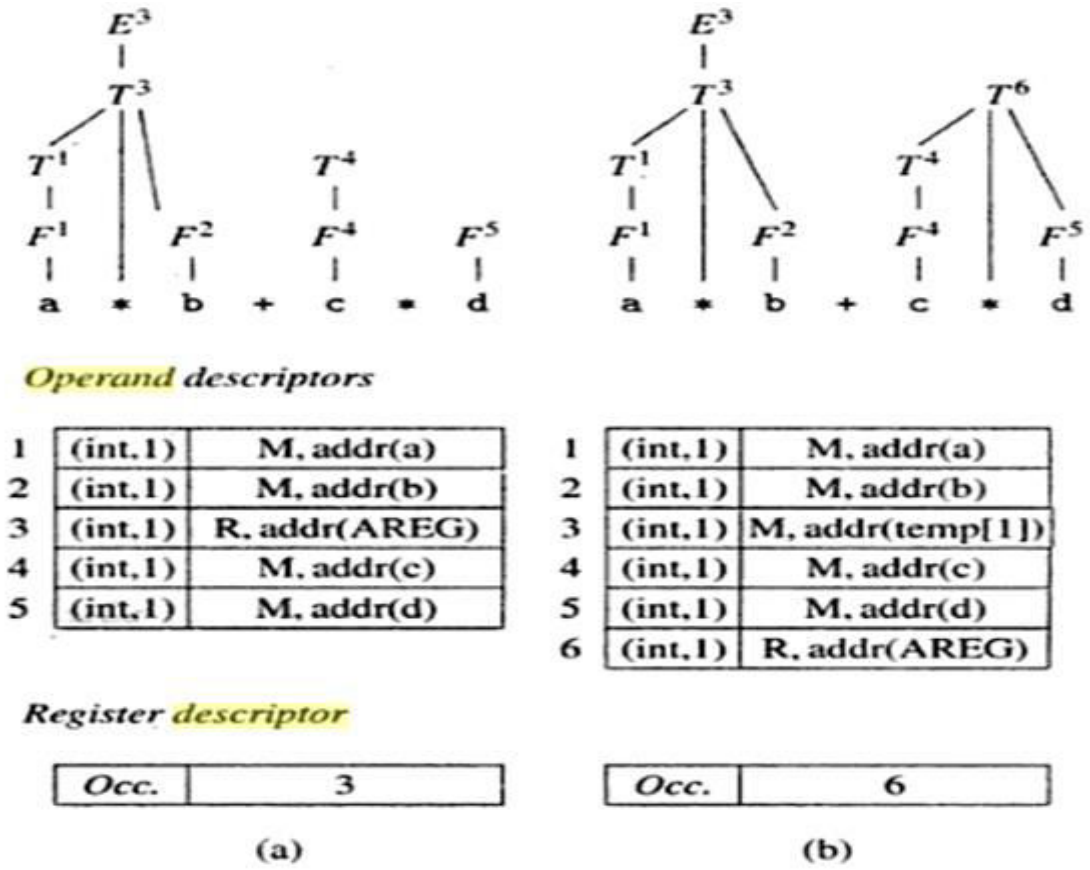
Code generation actions for a*b+c*d

Step no.	*Parsing action*	*Code generation action*
1.	$<id>_a \rightarrow F^1$	Build descriptor # 1
2.	$F^1 \rightarrow T^1$	–
3.	$<id>_b \rightarrow F^2$	Build descriptor # 2
4.	$T^1 * F^2 \rightarrow T^3$	Generate MOVER AREG, A MULT AREG, B Build descriptor # 3
5.	$T^3 \rightarrow E^3$	–
6.	$<id>_c \rightarrow F^4$	Build descriptor # 4
7.	$F^4 \rightarrow T^4$	–
8.	$<id>_d \rightarrow F^5$	Build descriptor # 5

9.	$T^4 * F^5 \rightarrow T^6$	Generate MOVEM AREG, TEMP_1
		MOVER AREG, C
		MULT AREG, D
		Build descriptor # 6

10.	$E^3 + T^6 \rightarrow E^7$	Generate ADD AREG, TEMP_1

Fig (a) shows parse tree and descriptors after step 8 and Fig (b) shows details after step 9



Error Handling

- On the basis of execution, errors can be classified into 2 types:
  - **1. Compile Time Errors**
  - **2. Runtime Errors**

Compile Time Errors

- 1. **Lexical Errors**
  - Character streams that don't match token patterns
  - Refer Module 1 for lexical errors and its handling
- 2. **Syntactic Errors**
  - Token streams that don't match the grammar

- Refer Module 2 for syntax errors and its handling

- 3. **Semantic Errors**
  - Syntactically legal statement that don't make sense

Run time Errors

- These are really program's logical errors.
- But compilers can generate code to check for some of such errors as:
  - Divide by zero
  - Indexes going beyond the range of an array
  - Pointers going beyond the program's memory
  - Infinite loops or recursive calls
- Some programming languages such as java, ADA have
  - default or user defined exceptions that **trap** these errors
  - and jump control to **exception handler function**, instead of simply crashing the program

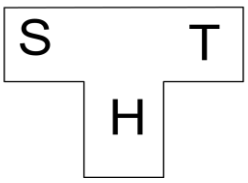
Incremental Compilers

- An incremental compiler is one that can recompile only those portions of a program that have been modified.
- Ordinary compilers must process entire modules or programs.
- The incremental compiler operates on an incremental basis, line-by-line, so if only one line is changed in an edit session, then only that line need be recompiled if no other code is affected.
- The compiler is able to skip lines or blocks of lines of source code which haven't been changed

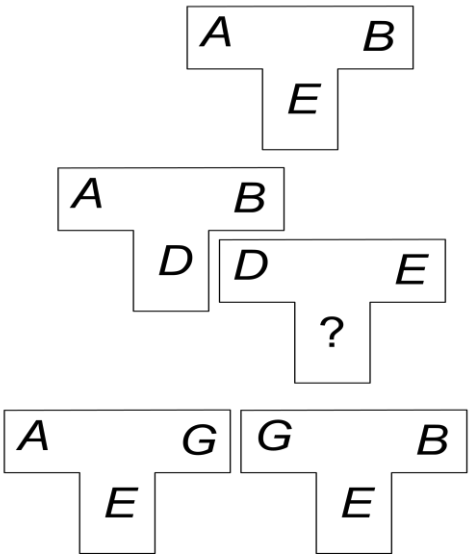
- For this purpose, each line of source text in the editor has a **change-tag** to indicate whether this line has been changed
- From this change-tag information a clean-lines table is built having a clean-lines indication for each line of source code, indicating how many clean lines follow the present line.

Cross Compiler

- A **cross compiler** is a compiler capable of creating executable code for a platform other than the one on which the compiler is running.
- Cross compiler tools are used to generate executables for embedded system or multiple platforms.
- It is used to compile for a platform upon which it is not feasible to do the compiling, like microcontrollers that don't support an operating system.
- A host language is a language in which the compiler is written.
    - T-diagram



- Cross compilers are used very often in practice.
- If we want a compiler from language *A* to language *B* on a machine with language *E*,
    - write one with *E*
    - write one with *D* if you have a compiler from *D* to *E* on some machine
    - It is better than the former approach if *D* is a high-level language but *E* is a machine language
    - write one from *G* to *B* with *E* if we have a compiler from *A* to *G* written in *E*


A
B
E
A
B
D
D
E
?
A
G
E
G
B
E