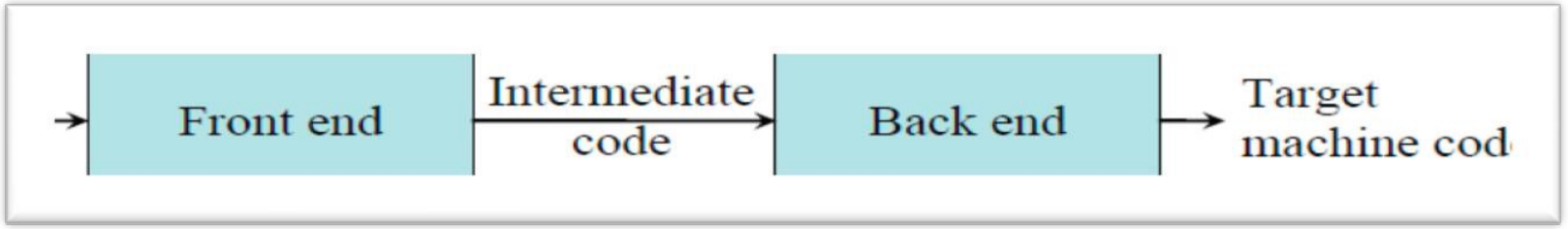
Unit -4

4.1 lintermediate Code Generator

Intermediate Code Generation

- The front end translates the source program into an intermediate representation from which the backend generates target code. Intermediate codes are machine independent codes, but they are close to machine instructions.



Advantage of Using Intermediate Code Generation

- It is Machine Independent. It can be executed on different platforms.
- It creates the function of code optimization easy. A machine-independent code optimizer can be used to intermediate code to optimize code generation.
- It can perform efficient code generation.
- From the existing front end, a new compiler for a given back end can be generated.

Level of Abstraction:

- The amount of details exposed in the Intermediate Representation influence the feasibility and profitability of different optimizations.
- Generally, it can be abstracted into three levels.

  (i) High Level: It is almost similar to the source language. It is good for memory disambiguation. Here, one Intermediate Representation implies one target machine operation. A single Intermediate Representation may include array access or a procedure call. It includes procedure calls and structured objects like structures and arrays.

  (ii) Medium Level: Here, it does not involve any structured objects but still it is independent of the target language. It can be source or target-oriented.

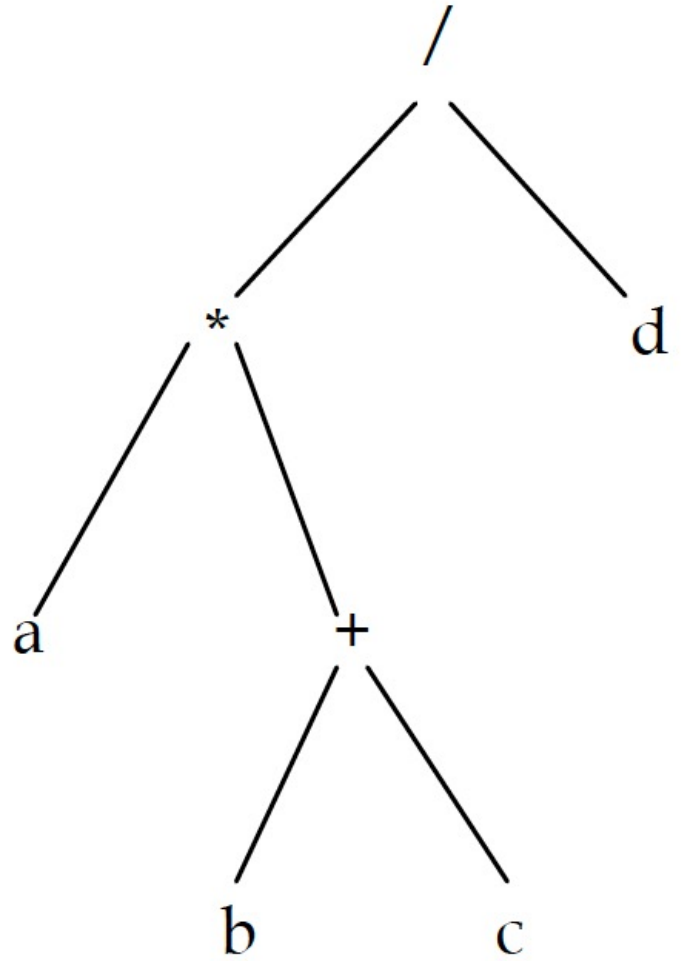
  (iii) Low Level: It is extremely close to the target language. A single Intermediate Representation may include an operation in the procedure. Here several Intermediate Representations implement one target machine optimization.

Intermediate Representations

- There are three kinds of intermediate representations:
  1. *Graphical representations* (e.g. Syntax tree or DAG)
  2. *Postfix notation*: operations on values stored on operand stack (similar to JVM byte code)
  3. *Three-address code*: (e.g. triples and quads) Sequence of statement of the form $x = y$ op z

Syntax tree:

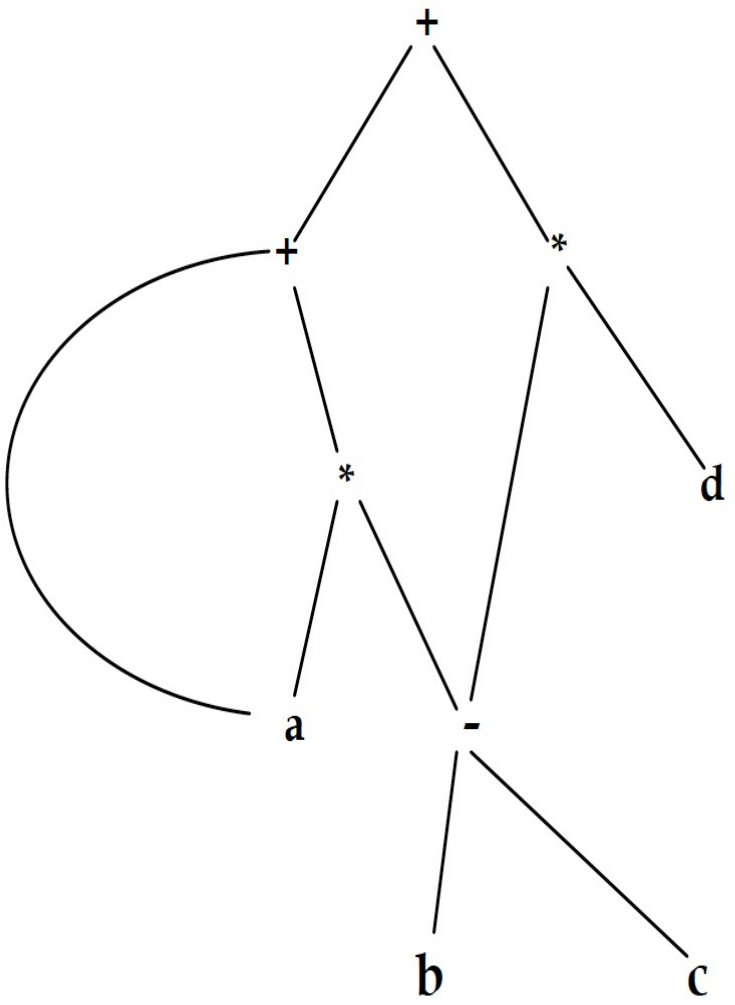
- Syntax tree is a graphic representation of given source program, and it is also called variant of parse tree. A tree in which each leaf represents an operand, and each interior node represents an operator is called syntax tree.
- Example: Syntax tree for the expression a*(b + c)/d



Directed acyclic graph (DAG)

A DAG for an expression identifies the common sub expressions in the expression. It is similar to syntax tree, only difference is that a node in a DAG representing a common sub expression has more than one parent, but in syntax tree the common sub expression would be represented as a duplicate sub tree.

Example: DAG for the expression a + a * (b - c) + (b - c) * d



Postfix Notation

- The representation of an expression in operators followed by operands is called postfix notation of that expression. In general if x and y be any two postfix expressions and OP is a binary operator then the result of applying OP to the x and y in postfix notation by “x y OP”.
  - Examples:
    1. (a+ b) * c in postfix notation is: a b + c *
    2. a * (b + c) in postfix notation is: a b c + *

Three Address Code

- The address code that uses three addresses, two for operands and one for result is called three code. Each instruction in three address code can be described as a 4-tuple:

  (operator, operand1, operand2, result)
- A quadruple (Three address code) is of the form:
- ***x = y op z***
- Where *x*, *y* and *z* are names, constants or compiler-generated temporaries and ***op*** is any operator.
- We use the term —three-address cod because each statement usually contains three addresses (two for operands, one for the result). Thus the source language like *x + y * z* might be translated into a sequence
- *t1 = y * z*
- *t2 = x + t1* where *t1* and *t2* are the compiler generated temporary name.

Assignment statements: *x* **=** *y op z, op is binary*

Assignment statements: *x* **=** *op y, op is unary*

Indexed assignments: *x* **=** *y*[*i*], *x*[*i*] **=** *y*

Pointer assignments: *x* **=** **&***y*, *x* **=** ***y**, ***x** **=** *y*

Copy statements: *x* **=** *y*

Unconditional jumps: **goto** *label*

Conditional jumps: **if** *x relop y* **goto** *label*

Function calls: **param** *x…* **call** *p, n* **return** *y*

- **Example: Three address code for expression: (B+A)*(Y-(B+A))**

  t1 = B + A
  t2 = Y - t1
  t3 = t1 * t2

- **Example 2: Three address code for expression:**

  i = 2 * n + k
  While i do
  i = i - k

- **Solution:**

  t1 = 2
  t2 = t1 * n
  t3 = t2 + k
  i = t3

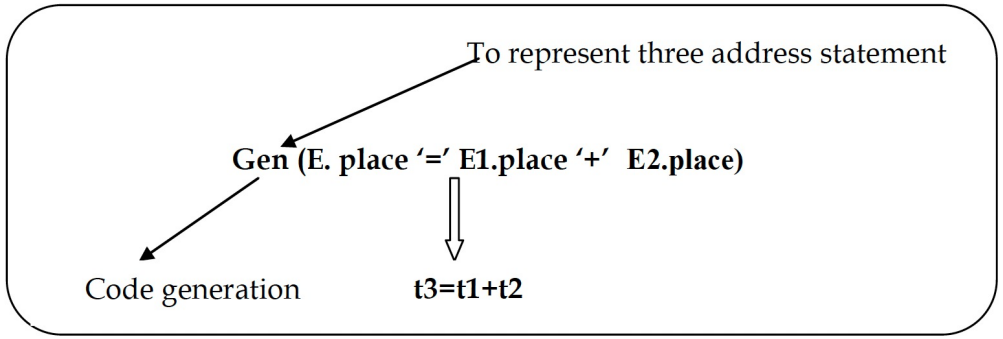
- L1: if i = 0 goto L2

  t4 = i - k
  i = t4
  goto L1

- L2: ……..

Naming conventions for three address code

- *S*.code →three-address code for evaluating *S*
- *S*.begin → label to start of *S* or nil
- *S*.after →label to end of *S* or nil
- *E*.code → three-address code for evaluating *E*
- *E*.place → a name that holds the value of *E*



Implementation of Three address code

1.Quadruple

It is a structure which consists of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

- **Advantage** –
- Easy to rearrange code for global optimization.
- One can quickly access value of temporary variables using symbol table.
- **Disadvantage** –
- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.

Example – Consider expression a = b * – c + b * – c

t1 = - c

t2 = b * t1

t3 = - c

t4 = b * t3

t5 = t2 + t4

a = t5

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

- **2. Triples** – This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.
- **Disadvantage** –
- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code.

Example – Consider expression a = b * – c + b * – c

t1 = - c

t2 = b * t1

t3 = - c

t4 = b * t3

t5 = t2 + t4

a = t5

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

3. Indirect Triples – This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Example – Consider expression a = b * – c + b * – c

t1 = - c

t2 = b * t1

t3 = - c

t4 = b * t3

t5 = t2 + t4

a = t5

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

List of pointers to table

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Indirect Triples representation

Syntax-Directed Translation into Three-Address Code

1. Assignment

Note: Follow Class Notes

2. **Boolean Expressions**

- Boolean expressions are used to compute logical values. They are logically used as conditional expressions in statements that alter the flow of control, such as if—then, if—then—else or while---do statements.

Note: Follow Class Notes

3.Flow of control statements

- Control statements are ‗if—then', ‗if—then—else', and ‗while---do'. Control statements are generated by the following grammars:
- S → If exp then S1
- S → If exp then S1 else S2
- S → while exp do S1

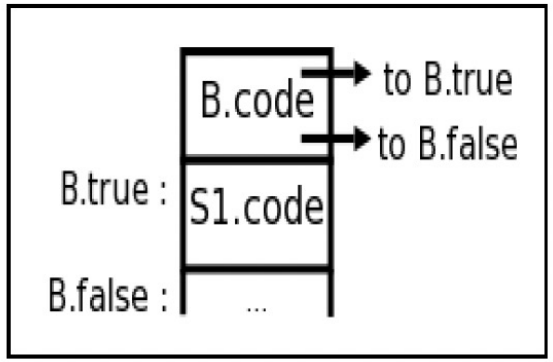


Fig: If--then

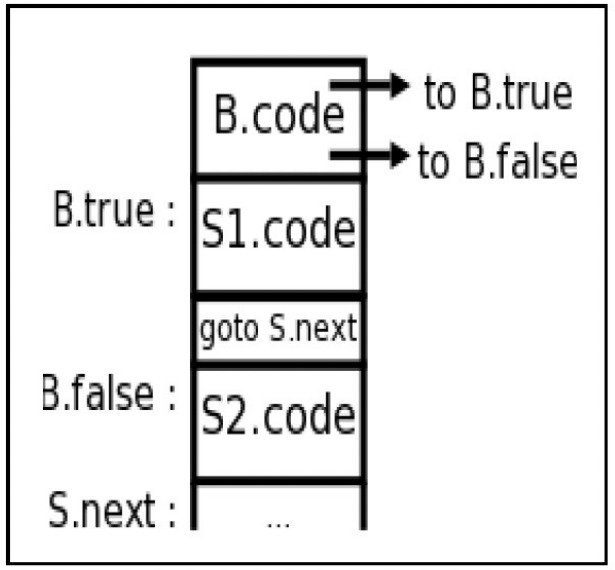


Fig: If—then—else

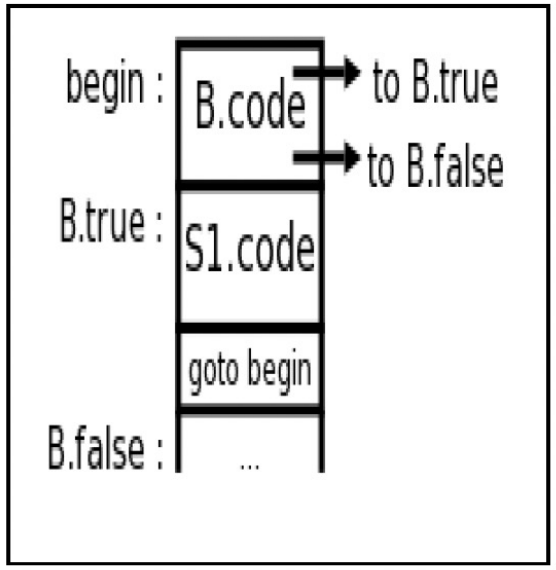
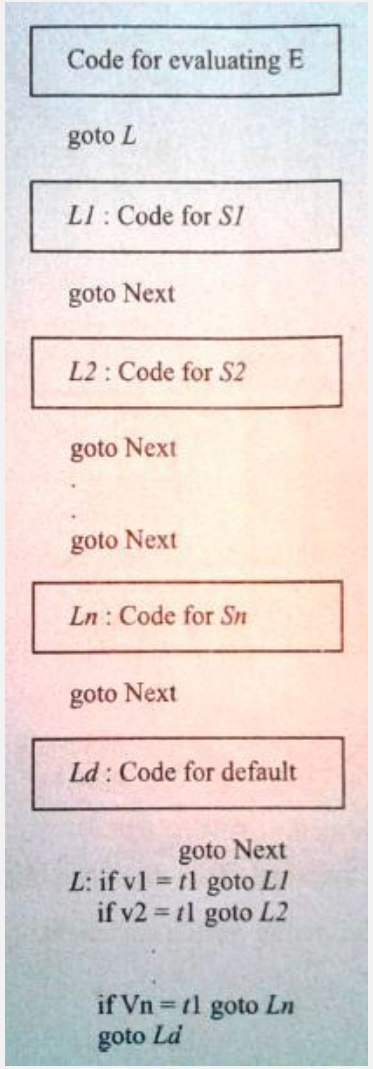


Fig: while---do

4. **Switch/ case statements**

- A switch statement is composed of two components: an expression E, which is used to select a particular case from the list of cases; and a case list, which is a list of n number of cases, each of which corresponds to one of the possible values of the expression E, perhaps including a default value.

```
Switch(E){
Case V1: S1
Case V2: S2
……
……
Case Vn:Sn}
```




```
Switch (i + j)
{
Case 1: x=y + z
Case 2: u=v + w
Case 3: p=q * w
Default: s=u / v
}
```

Addressing array elements:

- Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations. If the width of each array element is w, then the ith element of array 'A‘ begins in location,

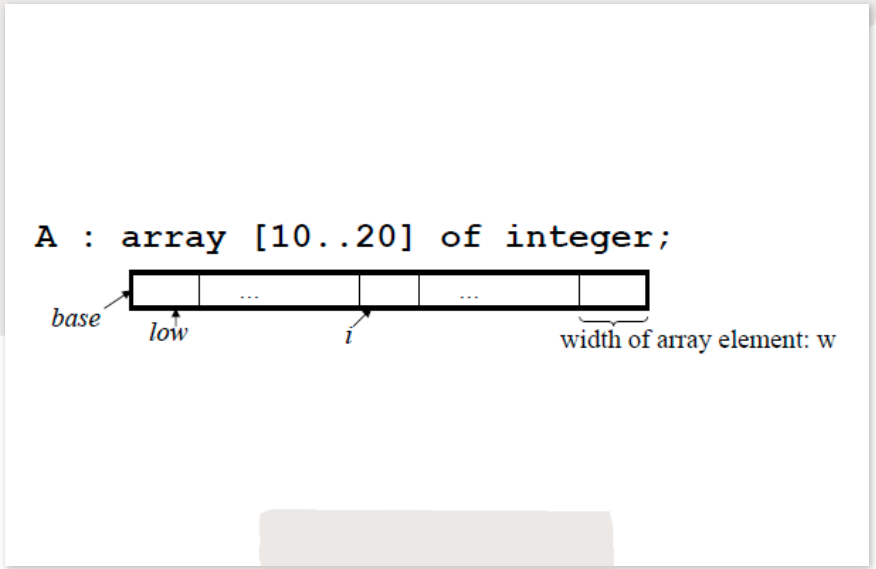
  base + (i – low)* w

- Where low is the lower bound on the subscript and base is the relative address of the storage allocated for the array. That is base is the relative address of A[low].
- The given expression can be partially evaluated at compile time if it is rewritten as,

  i * w + (base - low* w)

  i * w + C

- Where C=base – low*w can be evaluated when the declaration of the array is seen.



- We assume that C is saved in the symbol table entry for A, so the relative address of A[i] is obtained by simply adding i * w to C.

  i.e A[i]= i* w + C

  A[i] = base **A** + (i - low) * w

  i * w + c

- **Example:** address of 15th element of array is calculated as below,
- Suppose base address of array is 100 and type of array is integer of size 4 bytes and lower bound of array is 10 then,
- A[15]=15 * 4 + (100 – 10 * 4)
- = 60 + 60
- = 120

- Similarly for two dimensional array, we assume that array implements by using row major form, the relative address of A[i1, i2] can be calculated by the formula,
- A[i1,i2]=base**A** + ((i1 - low1) * n2 + i2 - low2) * w
- Where low1, low2 are the lower bounds on the values i1 and i2, n2 is the number of values that i2 can take. Also given expression can be rewrite as,

  = ((i1 * n2) + i2) * w + base**A** - ((low1 * n2) + low2) * w

  = ((i1 * n2) + i2) * w + C **where** C= base**A** - ((low1 * n2) + low2) * w

- **Example**: Let A be a 10 X 20 array, there are 4 bytes per word, assume low1=low2=1.
- **Solution:** Let X=A[Y, Z]
- Now using formula for two dimensional array as,

((i1 * n2) + i2) * w + base**A** - ((low1 * n2) + low2) * w

= ((Y * 20) + Z) * 4 + base**A** - ((1 * 20) + 1) * 4

= ((Y * 20) + Z) * 4 + base**A** – 84

- We can convert the above expression in three address codes as below:

- T1= Y * 20
- T1= T1+Z
- T2=T1*4
- T3=baseA -84
- T4=T2+ T3
- X= T4

Procedure Call

Param x call p return y

Here p is a function which takes x as a procedure and return y.

Procedure Calls have the form

P(A1,A2,……..An)
Param x1
Param x2
…….
Param xn
Call p, n

Return Statement: They have the form return y, representing a returned value is optional.

Example

```
Void main()
{
Intx,y;
Swap(&x,&y);
}
Void swap(int*a,int*b)
{
Int I;
i=*b
*b=*a
*a=i
}
```

Back patching

- If we decide to generate the three address code for given syntax directed definition using single pass only, then the main problem that occurs is the decision of addresses of the labels. _goto‘ statements refer these label statements and in one pass it becomes difficult to know the location of these label statements. The idea to back-patching is to leave the label unspecified and fill it later, when we know what it will be.
- If we use two passes instead of one pass then in one pass we can leave these addresses unspecified and in second pass this incomplete information can be filled up.

4.2 Code Generation

Code Generation and optimization

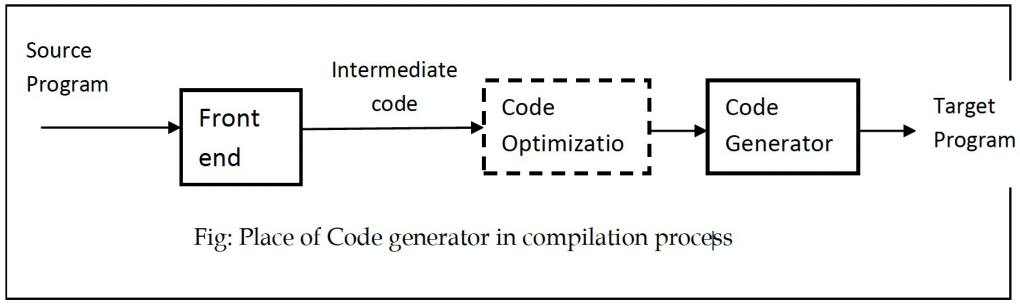


Fig: Place of Code generator in compilation process

- The process of transform intermediate code + tables into final machine (or assembly) code is known as code generation.
- The process of eliminating unnecessary and inefficient code such as dead code, code duplication etc from the intermediate representation of source code is known as code optimization.
- Code generation + Optimization are the back end of the compiler

Code generator design Issues

- The code generator mainly concern with:

1. Input to the code generator
2. Target program
3. Target machine
4. Instruction selection
5. Register allocation (Storage allocation)
6. Choice of evaluation order

1. Input to the Code Generator

- The input to the code generator is intermediate representation together with the information in the symbol table.

2. The Target Program

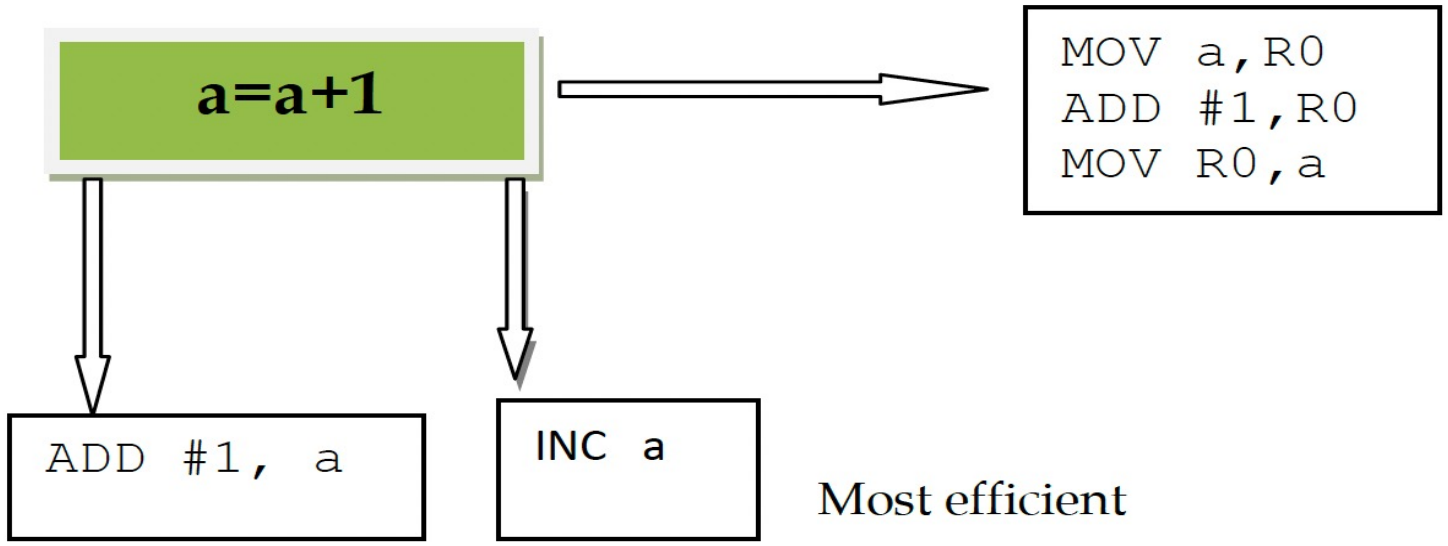
- The output of the code generator is target code. Typically, the target code comes in three forms such as: absolute machine language, relocatable machine language and assembly language.
- The advantage of producing target code in absolute machine form is that it can be placed directly at the fixed memory location and then can be executed immediately. The benefit of such target code is that small programs can be quickly compiled.

3. The Target Machine

- Implementing code generation requires thorough understanding of the target machine architecture and its instruction set.

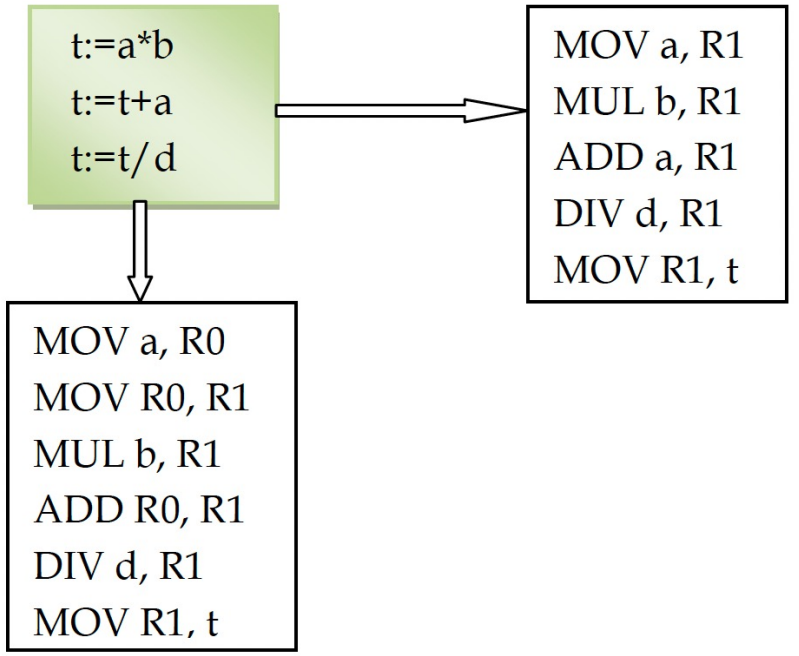
4. Instruction Selection

- Instruction selection is important to obtain efficient code. Suppose we translate three-address code,



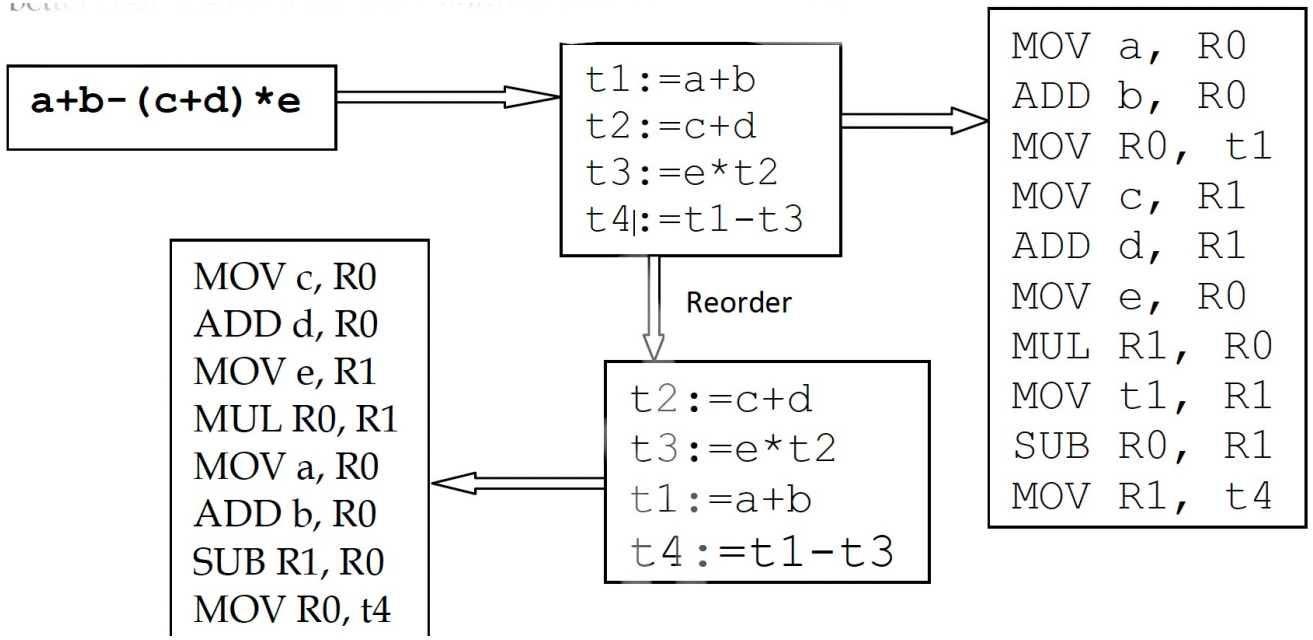
5. **Register Allocation**

- Since registers are the fastest memory in the computer, the ideal solution is to store all values in registers. However, there are normally not nearly enough registers for this to be possible. So we must choose which values are in the registers at any given time.
- Actually this problem has two parts.
    1. Which values should be stored in registers?
    2. Which register should each selected value be stored in
- The reason for the second problem is that often there are register requirements, e.g., floating-point values in floating-point registers and certain requirements for even-odd register pairs for multiplication/division.



6. Evaluation Order

- When instructions are independent, their evaluation order can be changed. Sometimes better code results if the three address codes are reordered.

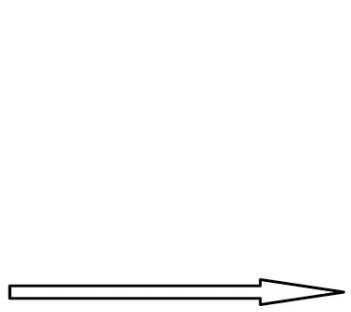


Basic Blocks

- A *basic block* is a sequence of consecutive instructions in which flow of control enters by one entry point and exit to another point without halt or branching except at the end.

Example:

```
        MOV 1, R0
        ADD n, R0
        MOV 2, R1
        MUL R0, R1
        JMP L2
L1:     MUL 2, R0
        SUB 1, R1
L2:     MUL 3, R1
        JMPNZ R1, L1
```



```
MOV 1, R0
ADD n, R0
MOV 2, R1
MUL R0, R1
JMP L2
```

```
L1: MUL 2, R0
    SUB 1, R1
```

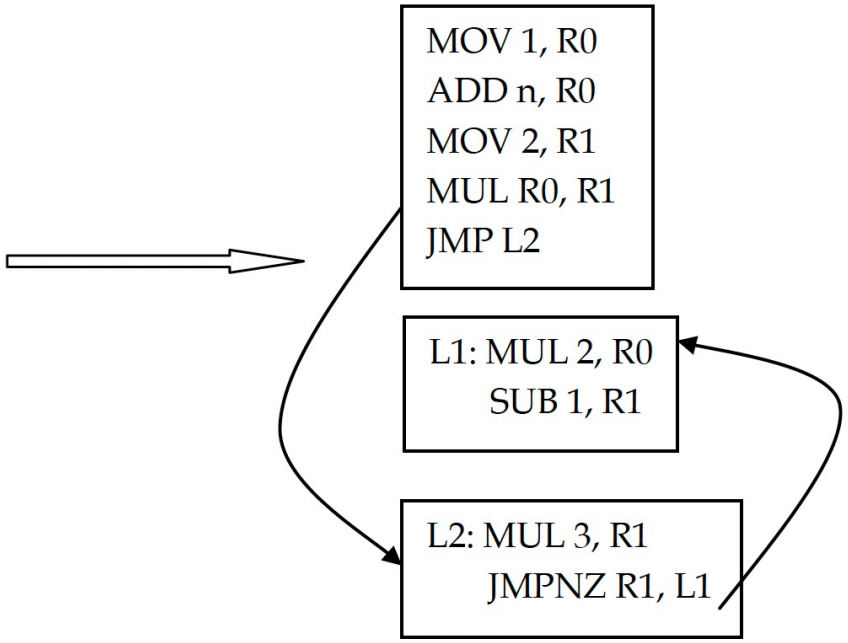
```
L2: MUL 3, R1
    JMPNZ R1, L1
```

Flow Graphs

- A *flow graph* is a graphical depiction of a sequence of instructions with control flow edges. A flow graph can be defined at the intermediate code level or target code level. The nodes of flow graphs are the basic blocks and flow-of-control to immediately follow node connected by directed arrow.
- Simply, if flow of control occurs in basic blocks of given sequence of instructions then such group of blocks is known as flow graphs.

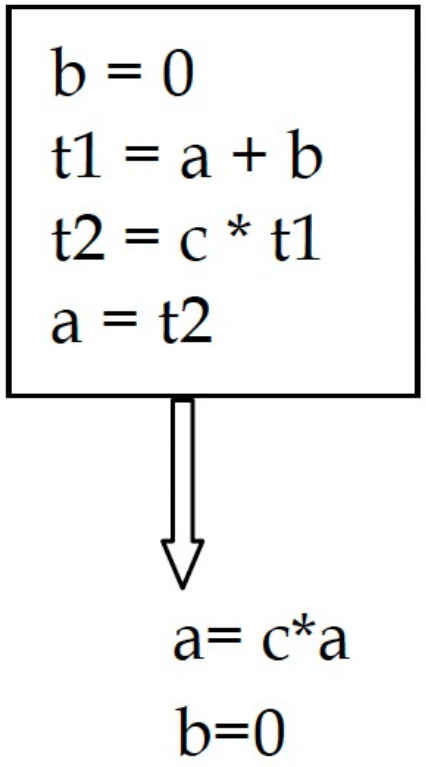
Example:

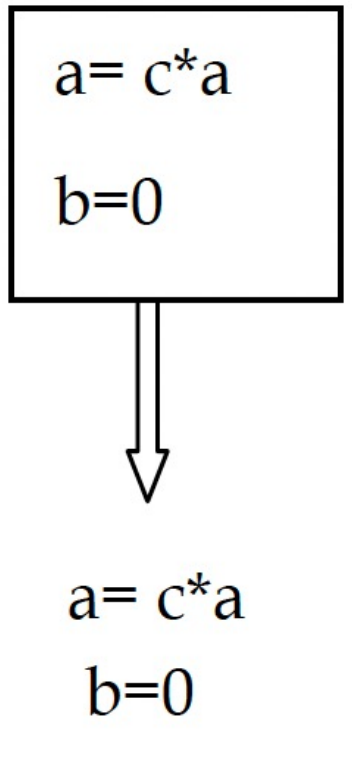
```
        MOV 1, R0
        ADD n, R0
        MOV 2, R1
        MUL R0, R1
        JMP L2
L1:     MUL 2, R0
        SUB 1, R1
L2:     MUL 3, R1
        JMPNZ R1, L1
```



Equivalence of basic blocks

- Two basic blocks are (semantically) *equivalent* if they compute the same set of expressions.





Transformations on Basic Blocks

- The process of code optimization to improve speed or reduce code size of given sequence of codes and convert to basic blocks by shafting and preserving the meaning of the code is known as transformation on basic blocks.
- There are mainly two types of code transformations:
  1. Global transformations and
  2. Local transformations

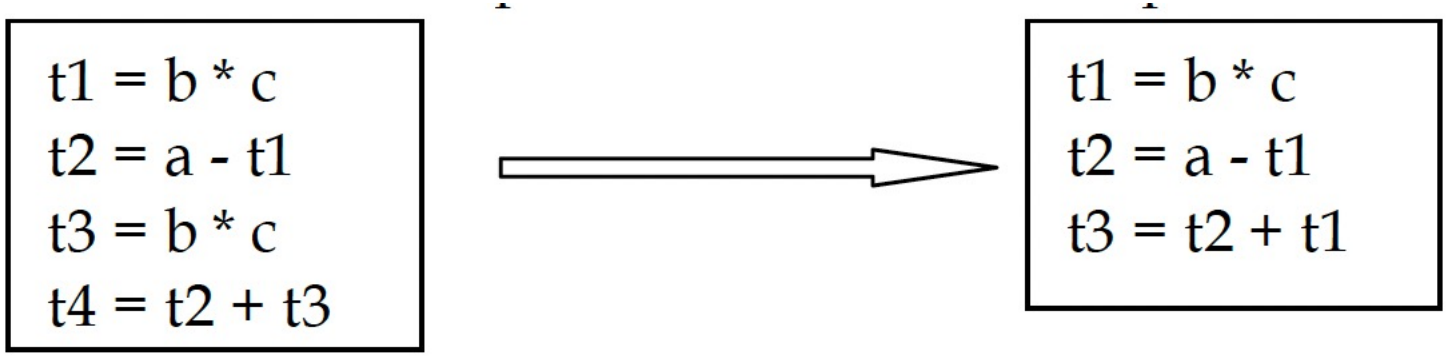
Global transformations are performed across basic blocks whereas *Local transformations* are only performed on single basic blocks.

- There are two classes of local transformations which are:

1. Structure preserving transformations and
   A. Common sub-expression elimination
   B. Dead code elimination
   C. Renaming of temporary variables
   D. Interchange of two independent statements
2. Algebraic transformations

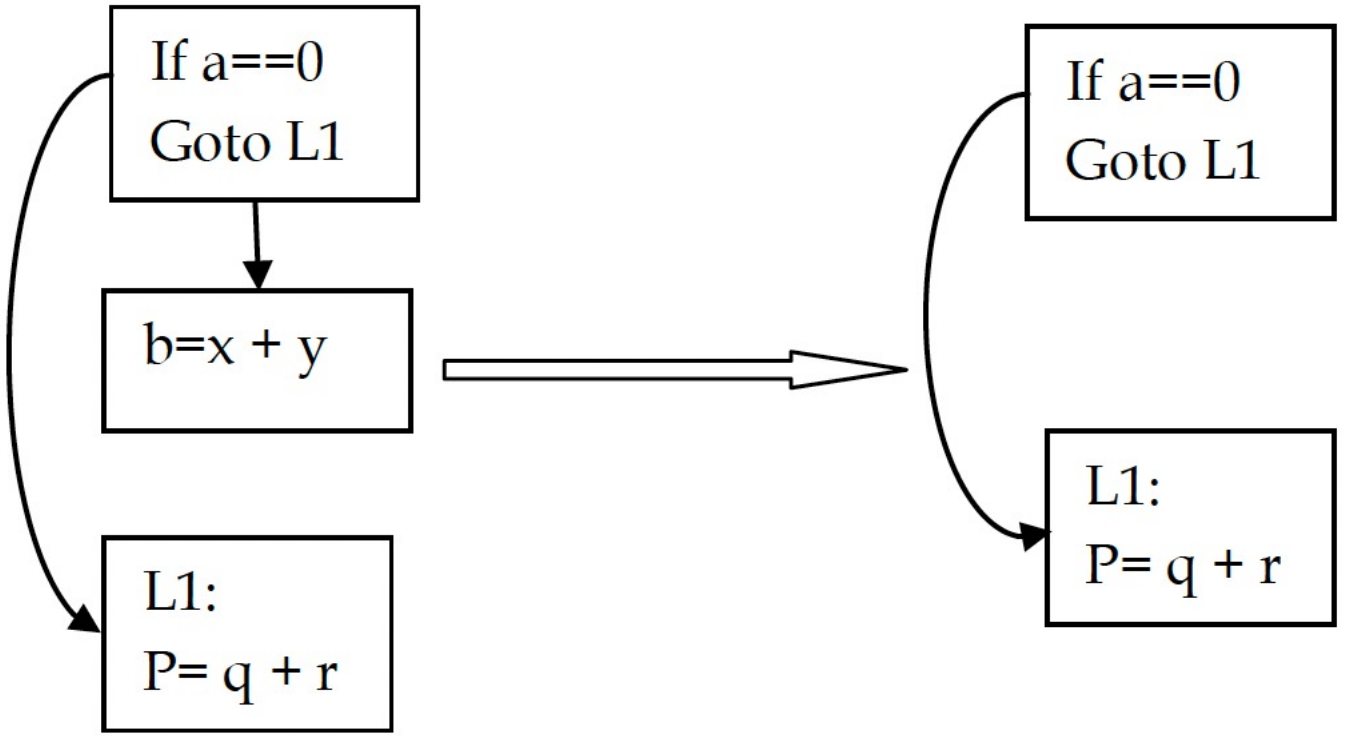
A. Common sub expression elimination

- Here we eliminate sub expressions that do not impact on our resultant basic block.



B. Dead code elimination

- Here we remove unused expressions

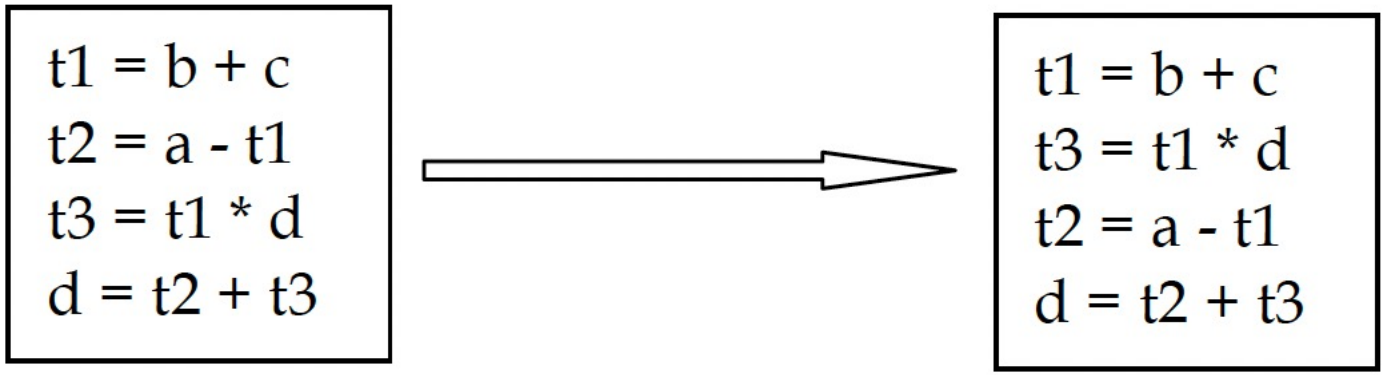


C. Renaming Temporary variables

- Temporary variables that are dead at the end of a block can be safely renamed.
- Let we have a statement t1=a + b where t1 is a temporary variable.
- If we change this statement to t2= a + b where t2 is a new temporary variable. Then the value of basic block is not changed and such new equivalent statement of their original statement is called ***normal-form block***.

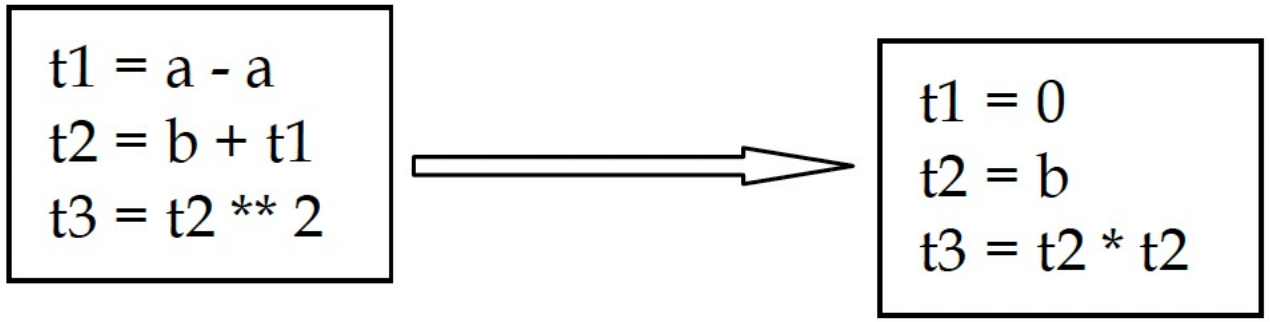
D. Interchange of statements

- Independent statements can be reordered without affecting the value of block to make its optimal use.



2. Algebraic Transformations

- Change arithmetic operations to transform blocks to algebraic equivalent forms. Here we replace expansive expressions by cheaper expressions.

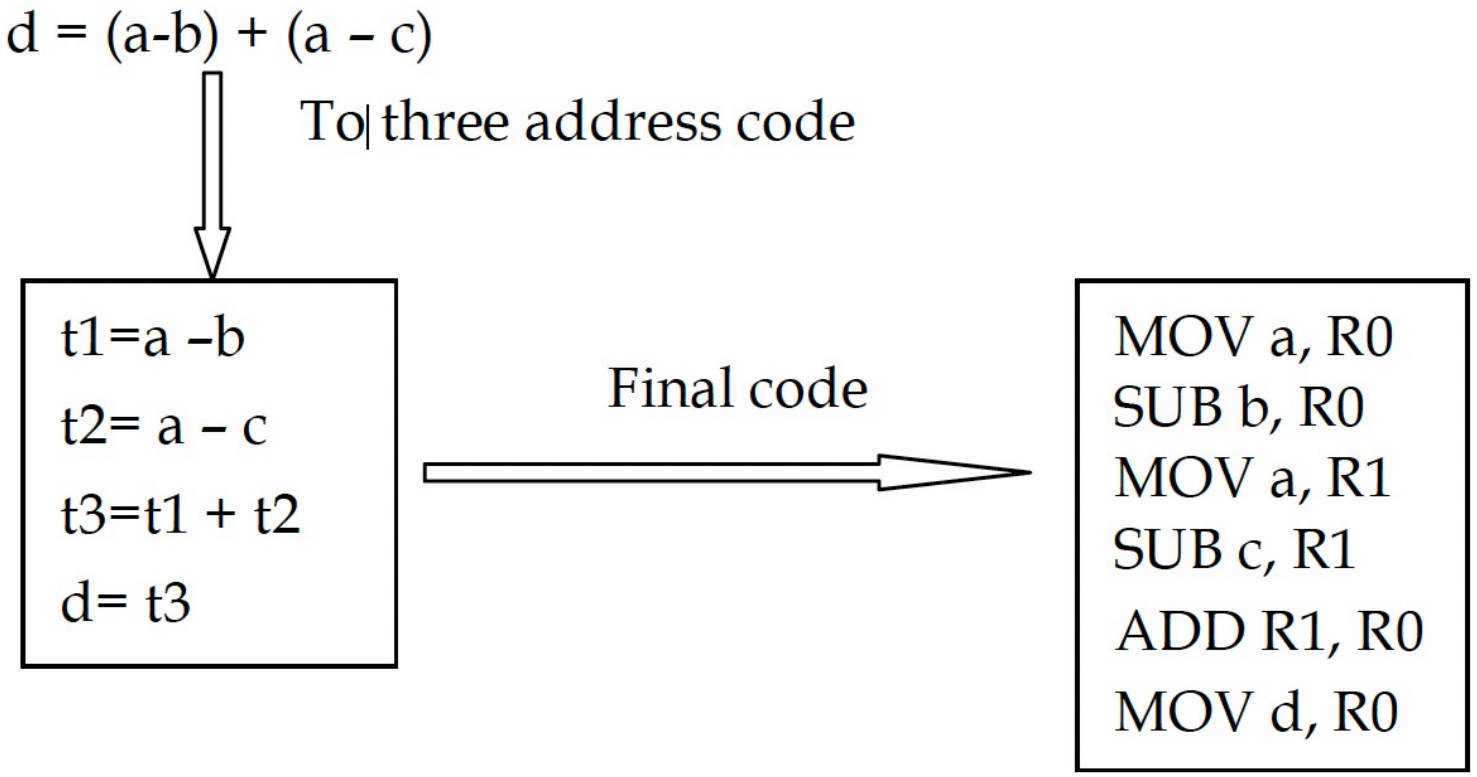


Next use information

- Next-use information is needed for dead-code elimination and register assignment (if the name in a register is no longer needed, then the register can be assigned to some other name).
- If *i: x = …* and *j: y = x + z* are two statements *i* & *j*, then *next-use* of *x* at *i* is *j*.
- Next-use is computed by a backward scan of a basic block and performing the following actions on statement
- ***i*: *x* = *y* op *z***
- Add liveness /next-use info on *x*, *y*, and *z* to statement *i* (whatever in the symbol table)
- Before going up to the previous statement (scan up):
  - Set *x* info to —not livell and —no next usell
  - Set *y* and *z* info to —livell and the next uses of *y* and *z* to *i*

Code generator

- The code generator converts the optimized intermediate representation of a code to final code which is normally machined dependent.



Register Descriptors

- A *register descriptor* keeps track of what is currently stored in a register at a particular point in the code, e.g. a local variable, argument, global variable, etc.
    - **MOV a, R0** “**R0** contains **a**”

Address Descriptors

- An *address descriptor* keeps track of the location where the current value of the name can be found at run time, e.g. a register; stack location, memory address, etc.
- **MOV a, R0**
- **MOV R0, R1** “**a** in **R0** and **R1**”

Statements	Code generated	Resister descriptor	Address Descriptor
t1=a –b	MOV a, R0 SUB b, R0	R0 contains t1	t1 in R0
t2= a – c	MOV a, R1 SUB c, R1	R0 contains t1 R1 contains t2	t1 in R0 t2 in R1
t3=t1 + t2	ADD R1, R0	R0 contains t3 R1 contains t2	t3 in R0 t2 in R1
d= t3+t2	ADD R1, R0 MOV d, R0	R0 contains d	d in R0 and memory

At first convert the following expression into three address code sequence then show the register as well as address descriptor contents.

- Statement: d = (a - b) + (a – c) + (a – c)
- Solution: The three-address code sequence of above statement is:

t1=a –b

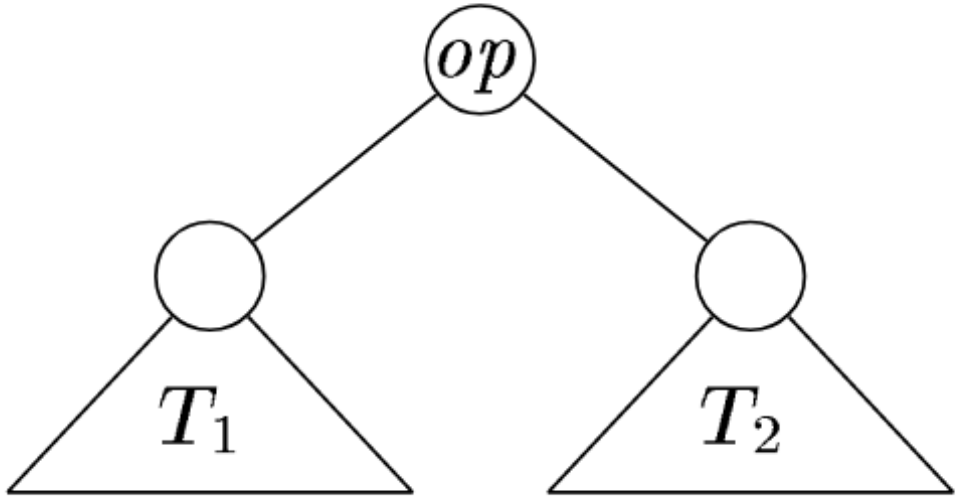
t2= a – c

t3=t1 + t2

d= t3+t2

Dynamic Programming Code Generation Algorithm

- A dynamic programming algorithm is used to extend the class of machines for which optimal code can be generated from expressions trees in linear time. This algorithm works for a broad class of register machines with complex instruction sets.
- Such an algorithm is used to generate code for machines with *r* interchangeable registers *R0, R1, ..., Rr-1* and load, store and operation instructions.



- For simplicity, we assume every instruction costs one unit, although the dynamic programming algorithm can easily be modified to work even if each instruction has its own cost.

1. Contiguous evaluation: Compute the evaluation of T1 and T2 then evaluate root.
2. Non-contiguous evaluation: first evaluate the part of T1 leaving the value in a register, next evaluate T2, then return to evaluate the rest of T1.

The dynamic programming algorithm uses contigious evaluation and proceeds in three phases:

1. Compute bottom-up for each node n of the expression tree T, an array c of costs, in which the ith component c[i] is the optimal cost of computing the subtree S rooted at n into a register assuming I registers are available for the computation for 1<=i<=r.
2. Traverse T, using the cost vectore to determinie which subtrees of T must be computed in memory.
3. Traverse each tree using the cost vector and associated instructions to generate the final target code. The code for the subtrees computed into memory locations is generated first.

4.3 Code Optimization

- Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.
- In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:
- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Need of Code Optimization

- Reduce the space consumed and increases the speed of compilation.
- Mannually analyzing dataset involves a lot of time. Hence, we make use of software like tableau for data analysis. Similarly, manually performing the optimization is also tendious and is better done using a code optimizer.
- An optimized code often promotes re-usability.

Basic Optimization Techniques

- Optimization techniques are classified into two categories:

1. Machine Independent optimization Techniques.
2. Machine Dependent Optimization Techniques.

1. Machine Independent Optimization Technique

- Machine independent optimization techniques are program transformation that improve the target code without taking into consideration any properties of the target machine.

1. Constant Folding

As it name suggest it involve folding the constant. The expressions that contain the operands having constant value at compile time are evaluated. Those expressions are then replaced with their respective results.

Example Circumference of circle=(22/7)*Diameter

Here,

1. This technique evaluates the expression 22/7 at compile time.
2. The expression is then replaced with its result 3.14.
3. This save time at run time.

2.Constant Propagation

- In this technique, if some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program during compilation.
- The condition is that the value of variable must not get alter in between.
- Example pi=3.14, radious=10, area of circle=pi*radious*radious

Here,

- This technique substiitutes the variable ’pi’ and ‘radious’ at compile time
- It then evaluates the expression 3.14*10*10
- The expression is then replaced with its result 314.
- This save the time at run time.

3. Redudant code elimination

- In compiler programming redundant code is source code or compiled code in a computer programming that is unnecessary such as code that is never executed (unrechable code).
- Example,
- x=y+z
- A=z+y+b+z
- That is reduced by,
  - A=z+x+b

4. Variable propagation

- Variable propagation means use of one variable instead of another.
- Example x=r

  A=pix^2

  This is reduced by A=pi*r^2

5. Strength reduction

- It involves reducing the strength of expressions. This technique replaces the expensive and costly operators with the simple and cheaper ones.
- Example B=A*2 B=A+A
- This is because the cost of multiplication operator is higher than that of addition operator

6. Loop optimization

- Loop optimization is the process of increasing execution speed and reducing the ooverhead associated with loops.

A. Code Motions/ Frequency Reduction:

It is a technique which move the code outside the loop.

Eg.,

```
For(i=0;i<100; i++)
{
x=i+a/b
y=y+2
}
```

B. Loop Jamming/ Loop Fusion

- In this technique combines the bodies of two loops whenever the same index variable and number of iterations are shared.
- Eg:, for(i=0; i<100;i++)

```
{
X[i]=1;
}
For (i=0; i<100; i++)
{
Y[i]=2;
}
```

C. Loop Unrolling

- Loop Transformation techniques that helps to optimize the execution time of a program.
- Replicate loop body multiple time, adjusting loop terminate code.
- Eg.,
- Sum=0
- For(i=0;i<100;i++)
- {
- Sum=sum+10;
- }

2. Machine Dependent Optimization Techniques:

- Machine dependent optimization techniques are based on register allocation and utilization of special machine instruction sequences.
- It is done after the target code has been generated and when the code is transformed according to the target machine architecture.

Peephole Optimization

- Peephole optimization is a simple and effective technique for locally improving target code.
- This technique is applied to improve performance of the target program by examinig the short sequence of target instruction called the peephole and replace these instructions by shorter or faster sequence whenever possible.
- Peephole optimization techniques are as follows:

a. Redudant load and store elimination

- In this technique the redudancy is eliminated which is shown in the following example.
- Example.,
  - Initial code
  - y=x+5;
  - i=y;
  - w=z*3;

Optimized code

y=x+5;

i=y;

w=y*3;

b. The flow of control optimization

i. Dead code Elimination
ii. Avoid jump on jump
iii. Algebric Simplification
iv. Machine Idioms
    - The target instructions have equivalent machine instructions for performing some operations. Hence we can replace these target instructions by equivalent machine instructions in order to improve the efficiency.
    - Example: some machine have auto-increment or auto decrement addressing mode.

V. Strength reduction:

- It involves reducing the strength of expressions.
- This technique replaces the expensive and costly operators with the simple and cheaper one.
- Example., B=A*2 B=A+A

Thank You