System Programming and Compiler Construction

CSC 602

Subject Incharge

Varsha Shrivastava

Assistant Professor

email: varshashrivastava@sfit.ac.in

Room No: 407

CSC 602 System Programming and Compiler Construction

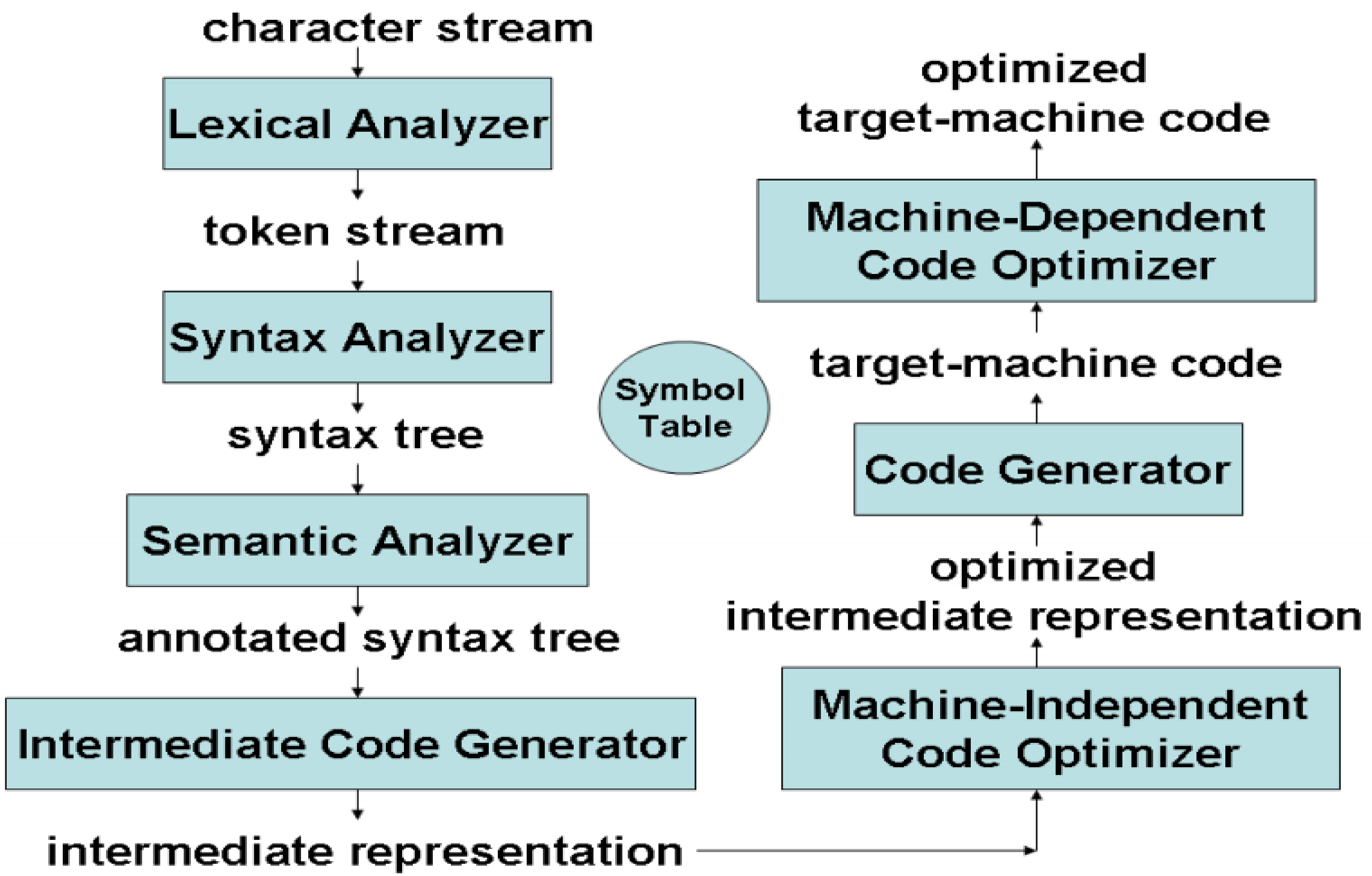
Module 6

Compilers : Synthesis phase

Contents as per syllabus

- Intermediate Code Generation:
  - Types of Intermediate codes: Syntax tree, Postfix notation, Three address codes: Triples and Quadruples.
- Code Optimization:
  - Need and sources of optimization
  - Code optimization techniques: Machine Dependent and Machine Independent.
- Code Generation:
  - Issues in the design of code generator
  - Code generation algorithm, Basic block and flow graph.

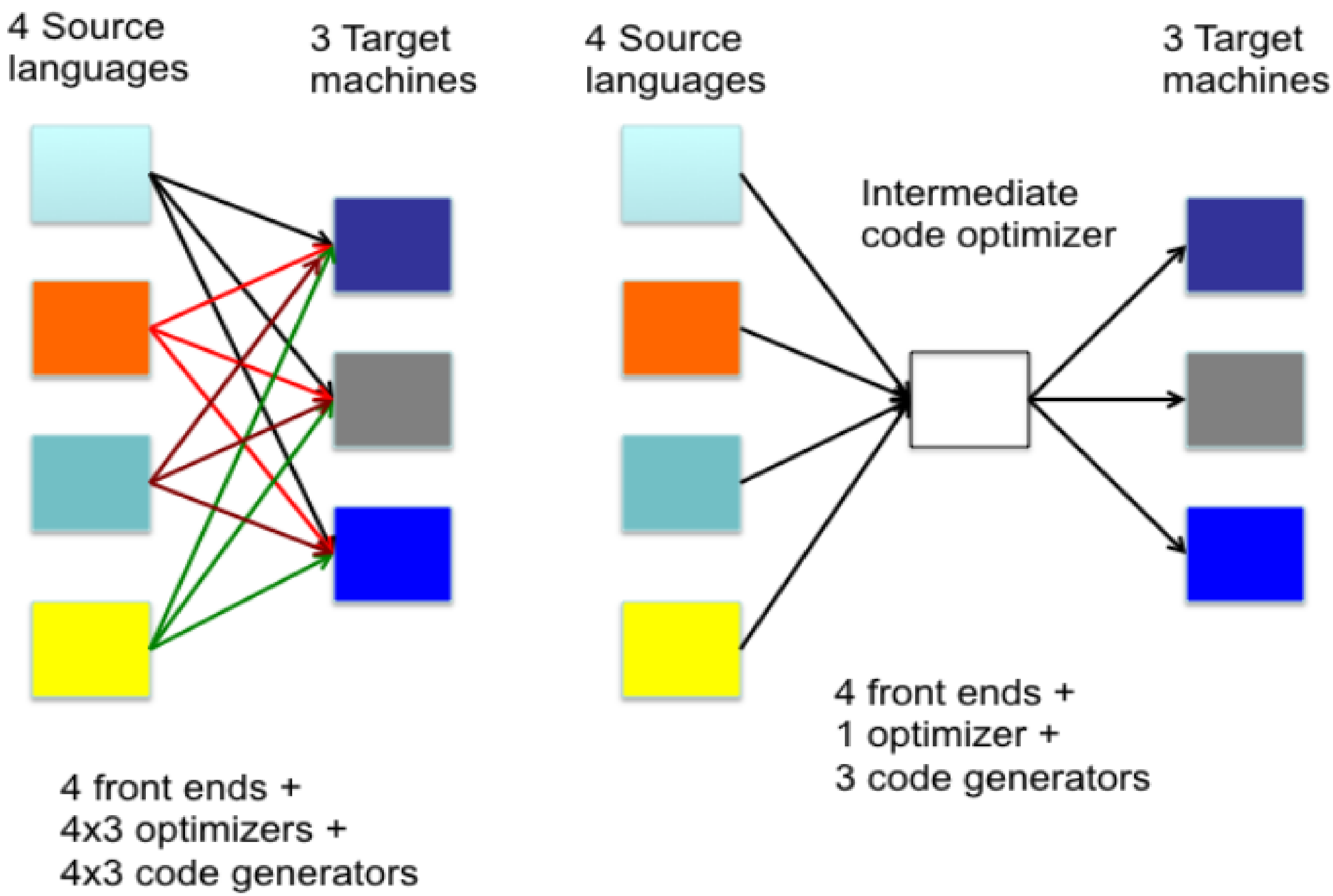
Compiler Overview



Compilers and Interpreters

- Compilers generate machine code, whereas interpreters interpret intermediate code
- Interpreters are easier to write and can provide better error messages (symbol table is still available)
- Interpreters are at least 5 times slower than machine code generated by compilers
- Interpreters also require much more memory than machine code generated by compilers
- Examples: Perl, Python, Unix Shell, Java, BASIC, LISP

Why Intermediate Code?



Why Intermediate Code?

- While generating machine code directly from source code is possible, it entails two problems
  - With m languages and n target machines, we need to write m front ends, m × n optimizers, and m × n code generators
  - The code optimizer which is one of the largest and very-difficult-to-write components of a compiler, cannot be reused.

❖ **By converting source code to an intermediate code, a machine-independent code optimizer may be written**

❖ **This means just m front ends, n code generators and 1 optimizer**

Intermediate Code

- Intermediate code must be easy to produce and easy to translate to machine code
  - A sort of universal assembly language.
  - Should not contain any machine-specific parameters (registers, addresses, etc.)
- The type of intermediate code deployed is based on the application
- Quadruples, triples, indirect triples, abstract syntax trees are the classical forms used for machine-independent optimizations and machine code generation
- Static Single Assignment form (SSA) is a recent form and enables more effective optimizations

Different Types of Intermediate Code

Syntax Tree Variants

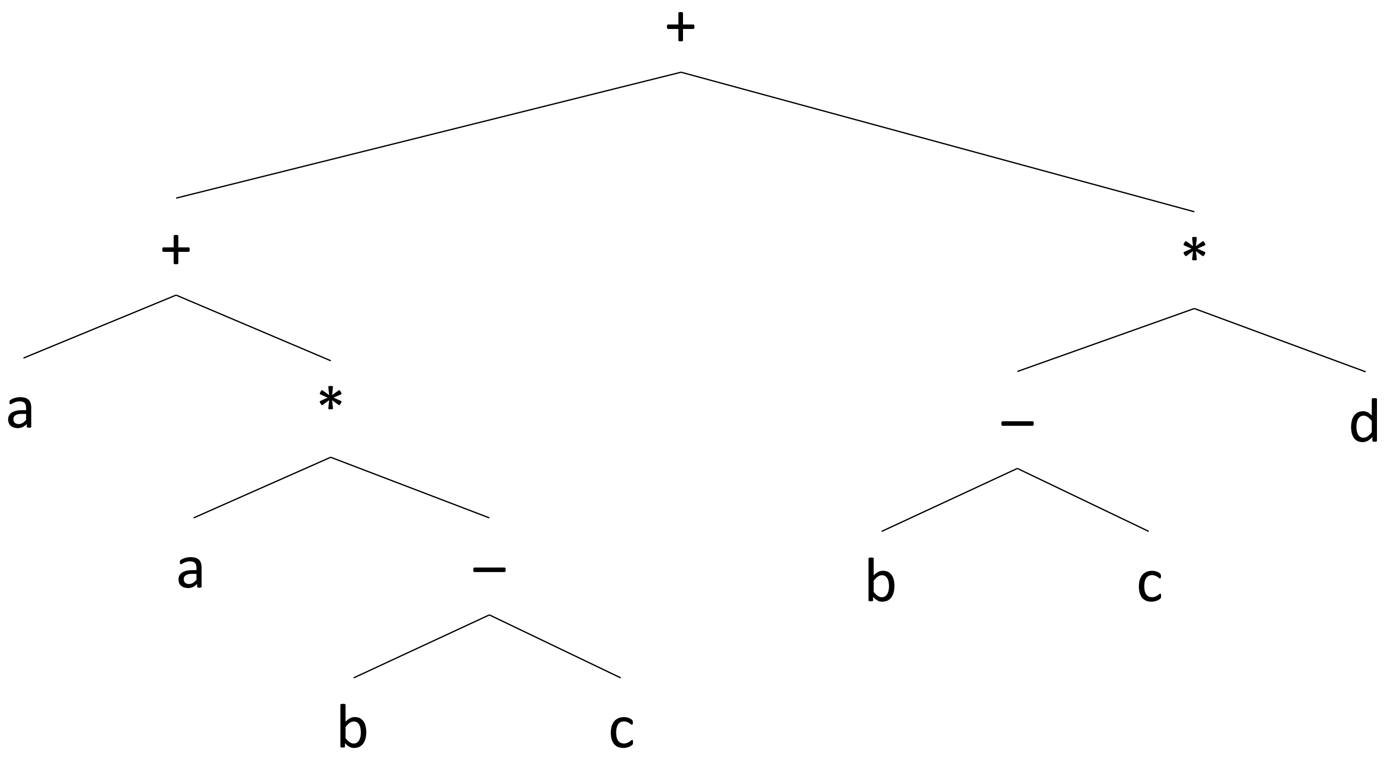
- **Directed Acyclic Graph (DAG)**
  - Similar to Syntax Tree
  - Uses compact representation

- **Value number method for creating DAGs**
  - Nodes of DAG are stored in an array of records
  - Each row of array corresponds to one record

Different Types of Intermediate Code

Example (Syntax Tree)

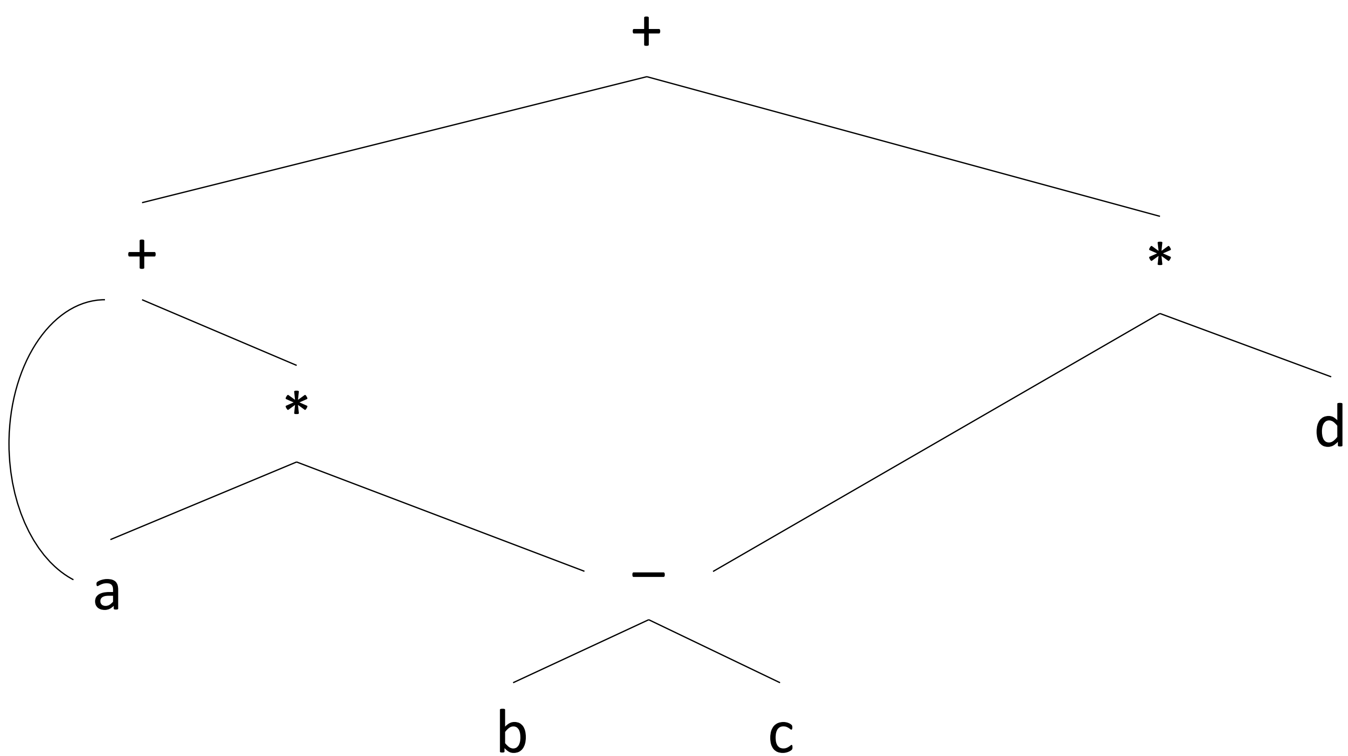
a + a * (b - c) + (b - c) * d



Different Types of Intermediate Code

Example (DAG)

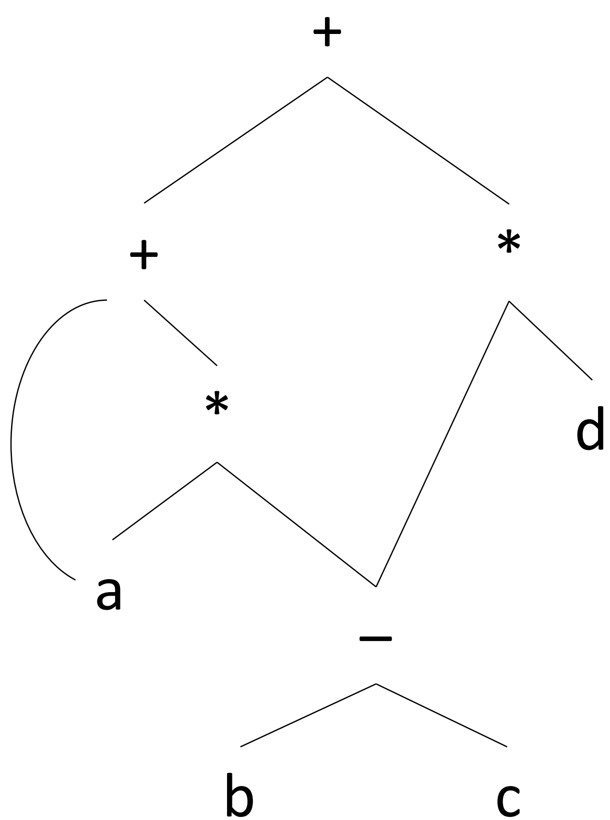
a + a * (b - c) + (b - c) * d



Different Types of Intermediate Code

Example (Value no. Method)

a + a * (b - c) + (b - c) * d



0	id	b	
1	id	c	
2	–	0	1
3	id	a	
4	*	3	2
5	+	3	4
6	id	d	
7	*	2	6
8	+	5	7

Different Types of Intermediate Code

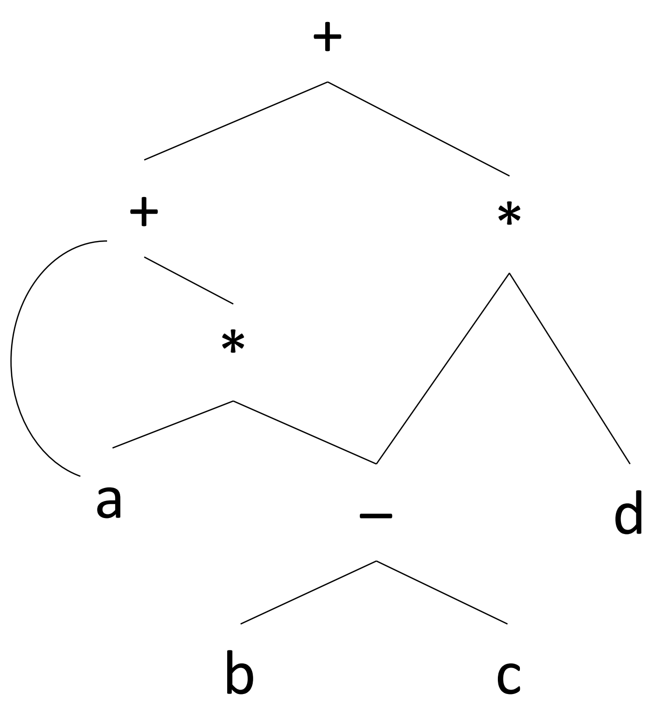
Three Address Code

- There is at most 1 operator on the right side of an instruction
- No built up arithmetic (multi-operator) expressions are permitted
- It is a linearized representation of a syntax tree or DAG in which explicit names correspond to interior nodes of the graph

Different Types of Intermediate Code

Three Address Code

Example : a + a * (b - c) + (b - c) * d



$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$

Different Types of Intermediate Code

Three Address Code

- Three address code is built from 2 concepts – addresses and instructions

- Address can be one of the following –
  - A name
  - A constant
  - A compiler generated temporary

Different Types of Intermediate Code

Three Address Code

- Instructions have the following common forms –
  - Assignment operations
    - x = y op z (where op is a binary operator)
    - x = op y (where op is a unary operator)
  - Copy instruction (x = y)
  - Unconditional Jump - goto L (L is a label)
  - Conditional Jump
    - if x goto L and if False x goto L
    - If x relop y goto L
  - Procedure calls
  - Indexed copy : x = y[i] OR x[i] = y
  - Address and pointer assignments – x=&y, x=*y and *x=y

Different Types of Intermediate Code

Three Address Code - Quadruples

- Has 4 fields – op, arg_1, arg_2 and result
- (result=arg_1 op arg_2)
- Example – a = b*(-c)+b*(-c)

t_1 = minus c
t_2 = b * t_1
t_3 = minus c
t_4 = b * t_3
t_5 = t_2 + t_4
a = t_5

	op	arg_1	arg_2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a

Different Types of Intermediate Code

Three Address Code - Triples

- Has only 3 fields – op, arg_1 and arg_2 (arg_1 op arg_2)
- Result field is used for temporary names
- Example – a = b*(-c)+b*(-c)

t_1 = minus c
t_2 = b * t_1
t_3 = minus c
t_4 = b * t_3
t_5 = t_2 + t_4
a = t_5

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Different Types of Intermediate Code

Three Address Code - Indirect Triples

- Does not list triples
- Consist of listing of pointers to triples
- An optimizing compiler can move an instruction by reordering the instruction list, without affecting the triples themselves

	instruction
30	(0)
31	(1)
32	(2)
33	(3)
34	(4)
35	(5)

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Different Types of Intermediate Code

Three Address Code - Static Single-Assignment (SSA)

- Intermediate representation that facilitates certain code optimizations
- There are 2 differences between SSA and 3AC
  1. All assignments in SSA are to variables with distinct names

$$
\begin{array}{l}
p = a + b \\
q = p - c \\
p = q * d \\
p = e - p \\
q = p + q
\end{array}
\qquad\qquad
\begin{array}{l}
p_1 = a + b \\
q_1 = p_1 - c \\
p_2 = q_1 * d \\
p_3 = e - p_2 \\
q_2 = p_3 + q_1
\end{array}
$$

(a) Three-address code. (b) Static single-assignment form.

What is a DAG?

- A directed acyclic graph (DAG) is an abstract syntax tree (AST) that has no cycles, and every node has a unique value.

What is the use of DAG?

- DAG is a useful data structure for implementing transformations on basic blocks.
- Transformations such as “common subexpression elimination” and “dead code elimination” can be applied to perform local optimization.
- A basic block of code can be optimized by construction of a DAG
- A DAG is constructed from the three address codes

Properties of a DAG

- The reachability relation in a DAG forms a partial order and any finite partial order may be represented by a DAG using reachability.
- The transitive reduction and transitive closure are both uniquely defined for DAGs
- Every DAG has a topological ordering

Applications of DAG

- Determine common subexpressions
- Determine which names are used in the block and computed outside the block
- Determine which statements of the block could have their computed value outside the block
- Simplify the quadruples by eliminating common subexpressions, and avoiding simple assignment statements (x:=y) as much as possible.

Rules for construction of a DAG

Rule 1: In a DAG

- leaf nodes will be identifiers or constants
- interior nodes will be operators

Rule 2: Before constructing a new node, check if there is an existing node with the same children.

Rule 3: The assignment of the form "x:= y" should not be performed, unless it is a must

Practice Question-

1. Construct a DAG for (a+b) * (a+b+c)

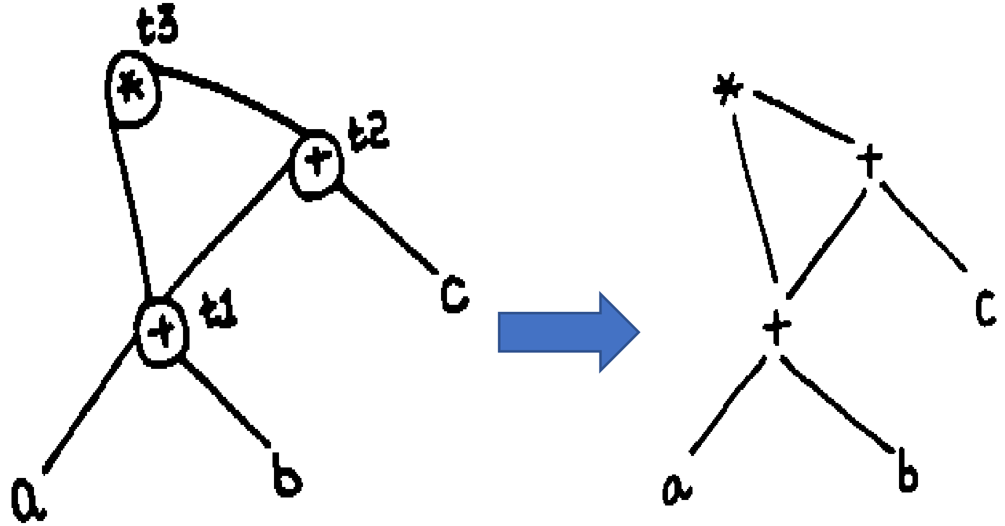
- First generate the 3AC (Three Address Code)

  t1 = a + b

  t2 = t1 + c

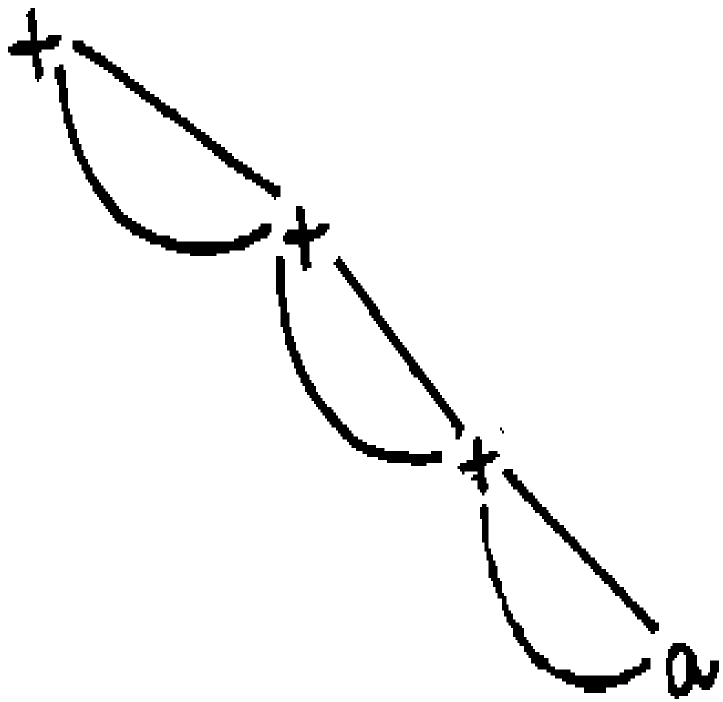
  t3 = t1 * t2

- Then construct DAG



Practice Question-

2. DAG for (((a+a) + (a+a)) + ((a+a) + (a+a)))



Practice Question-

3. Construct a DAG for the following block

a = b* c

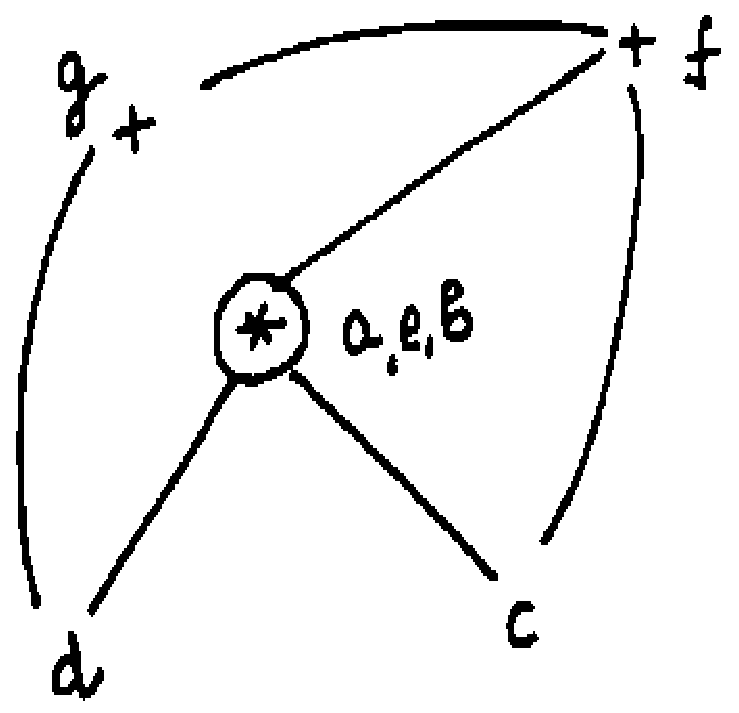
d = b

e = d * c

b = e

f = b + c

g = f + d



Practice Question-

Optimize the block

1. Construct DAG for given block
2. Optimized code can be generated by traversing the DAG
   1. The common subexpression e = d*c which is actually b*c and b = d, is eliminated
   2. The dead code b = e is eliminated
3. The optimized code is:

   a = d*c

   f = a + c

   g = f + d

Code Optimization

What is Code Optimization??

Optimization is the process of transforming a piece of code to make more efficient(either in terms of time or space) without changing its output or side-effects.

"Code optimization refers to the techniques a compiler can employ in an attempt to produce a better object language program"

Code Optimization

Types of Code Optimization

Machine Independent –

- The compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations.

Machine Dependent –

- Machine-dependent optimization is done after the target code has been generated
- When the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references

Code Optimization

When attempting any optimizing transformation, the following criteria should be applied:

- The optimization should capture most of the potential improvements without an <u>unreasonable amount of effort.</u>
- The optimization should be such that the <u>meaning of the source program is preserved.</u>
- The optimization should, on average, <u>reduce the time and space</u> expended by the object code

Types of Optimization

- Basic Blocks
  - Invariant Code Motion
  - Common sub-expression elimination
  - Dead code elimination
  - Loop unrolling

- Peephole
  - Elimination of redundant load and store operations
  - Elimination of unreachable code
  - Algebraic simplifications and strength reduction
  - Use of machine idioms
  - Elimination of multiple jumps

Code Optimization

Peephole Optimization

Peephole Optimization is a kind of optimization performed over a very small set of instructions in a segment of generated code. The set is called a "peephole" or a "window". It works by recognizing sets of instructions that can be replaced by shorter or faster sets of instructions.

Goals:

- improve performance
- reduce memory footprint
- reduce code size

Code Optimization Techniques

- Dead code elimination
- Constant folding and propagation
- Variable propagation
- Common sub-expression elimination
- Code movement
- Algebraic simplification and Strength reduction
- Loop Optimization

Code Optimization

Dead Code Elimination

- Portion of the program that is never visited and executed under any situation is called Dead Code
- It can be removed from the program without affecting behavior and performance
- A variable that is defined and not used is called a dead variable

Code Optimization

Dead Code Elimination

- Example

```
int main(){
…
flag = false;
if (flag)
        a = b + c;
…}
```

Code Optimization

Constant Folding

- It refers to placement of expressions that can be evaluated at compile time by their computed values

- It corresponds to determination of an expression with constant operands and replacing it with a single value

- Example : a = 5 + 3 + b + c can be written as

  a = 8 + b + c

Code Optimization

Constant Propagation

- Name variables are replaced with constants that have been assigned to them
- It is an **indirect optimization**
- Example : PI = 3.1415

  …

  R = PI/180.0 → R=3.1415/180 → R=0.0175

Code Optimization

Variable (Copy) Propagation

- It is the method of optimization that replaces one variable with another holding an identical value
- It may increase probability of generating common sub-expressions which can be easily removed
- Example:

BEFORE

```
a=b;
…
y=a*4;
z=b*4+10;
…
```

AFTER

```
a=b;
…
y=b*4;
z=b*4+10;
…
```

Common Sub-expression Elimination

- Some expressions may contain a sub-expression which may compute the same value
- This makes compiler execute same expression repeatedly
- The common sub-expression can only be eliminated if it computes the same value for 2 or more distinct expressions

Code Optimization

Example

```
...
...
Y = a + b;
...
Z = a + b + 10;
...
```

BEFORE

```
...
temp = a + b;
Y = temp;
...
Z = temp + 10;
...
```

AFTER

Code Optimization

Code Movement

- In this method, code from one part of the program can be moved to another part of the program
- The modified program maintains the semantic equivalence after modification
- Code movement is done to –
  - Reduce code space of the program
  - Reduce execution frequency

Code Optimization

Example

```
...
If p>q then
  x = a * 10;
  ...
Else
  y = a * 10 + 20;
...
```

BEFORE

```
...
temp = a * 10;
 If p>q then
x = temp;
        ...
 Else
y = temp+20;
...
```

AFTER

Code Optimization

Algebraic Simplification and Strength Reduction

Algebraic simplification – replacing simple algebraic expressions by a simplified expression

Example : $a = x * 0 \rightarrow a = 0$

$a = x * 1 \rightarrow a = x$

Strength Reduction – replacing expensive machine instructions by cheaper ones

Example : $x^2 \rightarrow x * x$

$x*2 \rightarrow x<<1$

Code Optimization

Loop Optimization

- Code motion
- Loop unrolling
- Loop interchange
- Loop splitting
- Loop pealing
- Loop fission

Code Optimization

Loop Optimization- Code motion

- Sometimes loops may contain expressions that retain its value (invariant expressions)
- Such expressions can be moved before the loop
- This reduces frequency of execution of the expression

Code Optimization

Example

```
…
while ( i < n-2 )
{
   …
}
…
```

BEFORE

```
…
t = n-2;
 while ( i < t )
{
         …
}
…
```

AFTER

Code Optimization

Loop Optimization- Loop Unrolling

- Body of the loop is reproduced a number of times
- This method improves efficiency by reducing loop overhead – test and loop variable
- Loop structure may disappear
- In partial unrolling increases source statements but reduces loop execution

Code Optimization

Example

```
…
for (int i=0; i<100; i++)
{
  printf("%d\n",i);
}
…
```

BEFORE

```
…
for (int i=0; i<100; i=i+5)
{
        printf("%d\n",i);
        printf("%d\n",i+1);
        printf("%d\n",i+2);
        printf("%d\n",i+3);
        printf("%d\n",i+4);
} …
```

AFTER

Loop Optimization- Loop Interchange

- Inner and outer loops are interchanged to improve locality of reference
- Example:

```
...
for (int J=0; J<50; J++)
        for (int i=0; i<50; i++)
                A[i][J] = ...;
...
```

BEFORE

```
...
for (int i=0; i<50; i++)
for (int J=0; J<50; J++)
                A[i][J] = ...;
...
```

AFTER

Code Optimization

Loop Optimization- Loop Splitting

- Breaking of a single loop into multiple loops
- Example:

```
for i=1 to 100 do
   if i<50 then b1
   if 50<= i <60 then
         b2
   else
         b3
end loop
```

BEFORE

```
for i=1 to 49 do
         b1
end loop
for i=50 to 59 do
         b2
end loop
for i=60 to 100 do
         b3
end loop
```

AFTER

Code Optimization

Loop Optimization- Loop Fission

- Breaking a loop over multiple loops with same index range
- Example :

```
…
for i= 1 to 50
   x[i] = 0;
   y[i] = 0;
End loop
…
```

BEFORE

```
…
for i= 1 to 50
        x[i] = 0;
end loop
for i= 1 to 50
        y[i] = 0;
end loop
…
```

AFTER

Code Optimization

Loop Optimization- Loop Pealing

- If there is a condition on the first value of the loop variable, it can be performed separately
- Example:

```
…
…
for i=1 to 50 do
  if i=1 then b1;
  else b2;
End loop
…
```

→

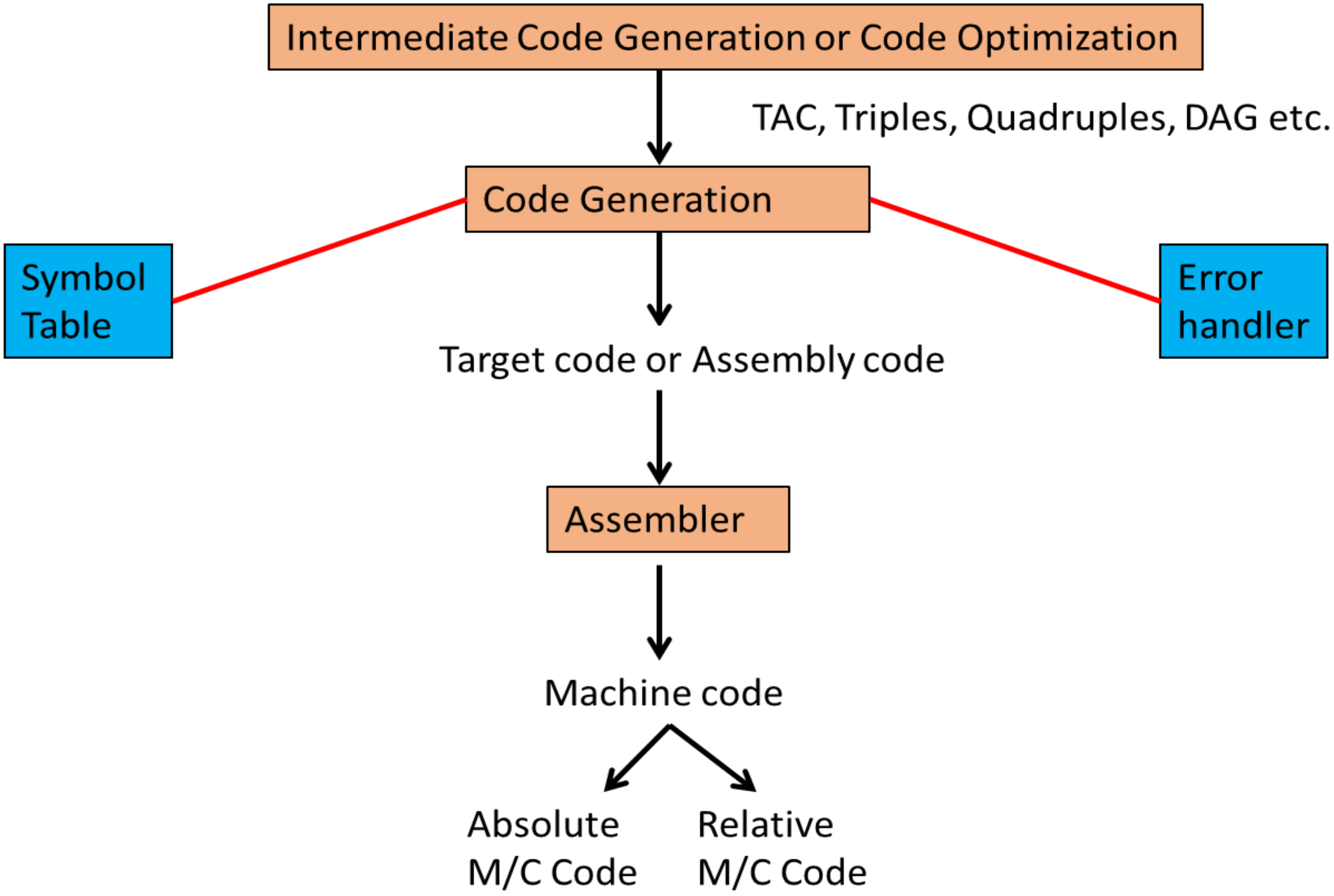
```
i=1
        b1;
for i= 2 to 50 do
        b2;
end loop
…
```

CODE GENERATION

Code Generation

- Code generation is the final activity of compiler design
- Code generation is a process of creating machine language/ assembly language
- Properties of Code Generation:-
  - Correctness
  - High Quality
  - Efficient use of the resource of target machine
  - Quick Code generation

Code Generation



Code Generation

- Absolute Machine Code :
  - Fixed location in memory
  - Execution speed is fast.
  - Useful for small program
- Relative Machine Code:
  - Not a fixed location
  - Code can be placed wherever linker find the room in RAM
  - Useful for commercial Compilers

Issues in the design of code generator

- Input to the code generator
- Target program
- Memory management
- Instruction selection
- Register allocation
- Choice of evaluation order
- Approaches to code generation

Issues in the design of code generator

Input to the code generator

- The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation
- Intermediate representation can be :
  - Linear representation such as postfix notation
  - Three address representation such as quadruples
  - Virtual machine representation such as stack machine code
  - Graphical representations such as syntax trees and DAG's

Issues in the design of code generator

Input to the code generator

- Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking.
- Therefore, <u>input to code generation is assumed to be error-free.</u>

Issues in the design of code generator

Target program

- The output of the code generator is the target program. The output may be :
  - **Absolute machine language** - It can be placed in a fixed memory location and can be executed immediately.
  - **Relocatable machine language** - It allows subprograms to be compiled separately.
  - **Assembly language** - Code generation is made easier.

Issues in the design of code generator

Memory management

- The instructions of target machine should be complete and uniform.
- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
- The quality of the generated code is determined by its speed and size.

Issues in the design of code generator

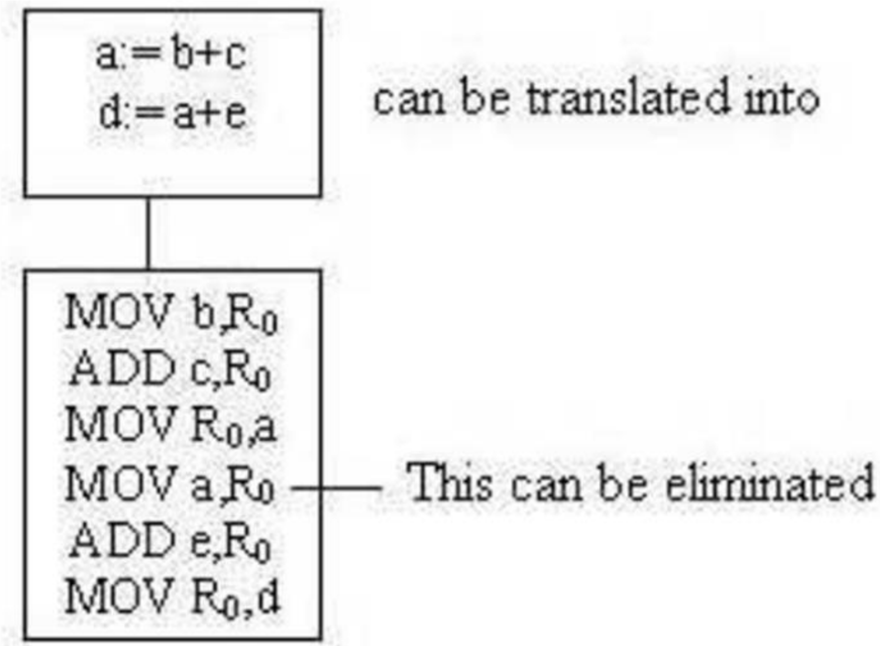
Instruction selection

- Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator. A name in the three address statements refers to the symbol table entry for name.
- Then from the symbol table entry, a relative address can be determined for the name.

Issues in the design of code generator

Instruction selection

- The former statement can be translated into the latter statement as shown below:



MOV b, R_0
ADD c, R_0
ADD e, R_0
MOV R_0, d

Issues in the design of code generator

Register allocation

- Instructions involving register operands are shorter and faster than those involving operands in memory.
- The use of registers is subdivided into two sub problems :
  - **Register allocation** – the set of variables that will reside in registers at a point in the program is selected.
  - **Register assignment** – the specific register that a variable will reside in is picked.

Issues in the design of code generator

Register allocation

- Certain machine requires even-odd register pairs for some operands and results.
- For example , consider the division instruction of the form : **D x, y**

    where,

    x – dividend even register in even/odd register pair

    y – divisor even register holds the remainder odd register holds the quotient

Issues in the design of code generator

Evaluation order

- The order in which the computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results than others.

Issues in the design of code generator

Approaches to code generation issues

Code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face.

Some of the design goals of code generator are :

- Correct
- Easily maintainable
- Testable
- Efficient

Designing of code generator should be done in such a way so that it can be easily implemented, tested and maintained.

Code Generation Algorithm

Code Generator generates target code for a sequence of instruction.

It uses a function getReg() to assign register to variables.

It uses 2 data structures:-

1. Register Descriptor :-
   - Used to keep track of which variable is stored in a register.
   - Initially all the registers are empty.
2. Address Descriptor :-
   - Used to keep track of location where variables is stored. Location may be Register, Memory Address, stack, etc.

Code Generation Algorithm

The algorithm takes as input a sequence of three-address statements constituting a basic block.

For each three-address statement of the form

x : = y op z, perform the following actions:

Step 1 : Invoke a function getreg() to determine the location L where the result of the computation y op z should be stored.

Step 2 : Determine the present location of ‘y’ by consulting address descriptor of y. If y is not present in the location ‘L’ then generate the instruction MOV y’, L to copy value of y to L

Code Generation Algorithm

Step 3 :- The present Location of Z is determined using step 2 and the instruction is generated as OP z' , L

Step 4 : Now L contains the value of y op z i.e. assigned to x. So, if L is a register then update its descriptor that it contains value of x. Update address Descriptor of x to indicate that it is stored in 'L'.

Step 5 : if y, z have no future use then update the Descriptor to remove y and Z

Code Generation Algorithm

Example:- d : = (a-b) + (a-c) + (a-c)

Three Address Code→

t : = a - b
u : = a - c
v:= t+ u
d : = v + u
with d live at the end.
Code sequence for the example is:

Register Descriptor
Describes which register contains which variable.
Address Descriptor
Describes the location the variable is stored

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
t : = a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u : = a - c	MOV a , R1 SUB c , R1	R0 contains t R1 contains u	t in R0 u in R1
v:=t+ u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0
d : = v + u	ADD R1, R0	R0 contains d	d in R0 d in R0 and memory
	MOV R0, d		

Basic Blocks

- The basic block is a sequence of consecutive statements which are always executed in the same sequence without halt or a possibility of branching.
- The basic block will not have any jump statements

Example: a = b+c+d

1. t1 = b+c
2. t2 = t1+d
3. a = t2

if (a<b) then 1
else 0

1. if a<b goto 4
2. t1 = 0
3. goto 5
4. t1 = 1
5. ...

Basic blocks and Flow Graph

Rules for partitioning into blocks

- After an intermediate code is generated for the given code, we can use the following rules to partition into blocks

Rule 1: Determine the leaders

- The first statement is a leader
- Any target statement of a goto is a leader
- Any statement immediately following a goto is a leader

Rule 2: Basic block is formed starting from the leader, and ends before the next leader appears.

Basic blocks and Flow Graph

Identify the basic blocks of given 3AC

```
(1) PROD = 0
(2) I = 1
(3) T2 = addr(A) - 4
(4) T4 = addr(B) - 4
(5) T1 = 4 * I
(6) T3 = T2[T1]
(7) T5 = T4[T1]
(8) T6 = T3 * T5
(9) PROD = PROD + T6
(10) I = I + 1
(11) IF I <= 20 GOTO (5)
```

```
✓ (1) PROD = 0
  (2) I = 1
  (3) T2 = addr(A) - 4
  (4) T4 = addr(B) - 4
✓ (5) T1 = 4 * I
  (6) T3 = T2[T1]
  (7) T5 = T4[T1]
  (8) T6 = T3 * T5
  (9) PROD = PROD + T6
  (10) I = I + 1
  (11) IF I <= 20 GOTO (5)
```

Basic blocks and Flow Graph

B1

```
PROD = 0
I = 1
T2 = addr(A) - 4
T4 = addr(B) - 4
```

B2

```
T1 = 4 * I
T3 = T2[T1]
T5 = T4[T1]
T6 = T3 * T5
PROD = PROD + T6
I = I + 1
IF I <= 20 GOTO B2
```

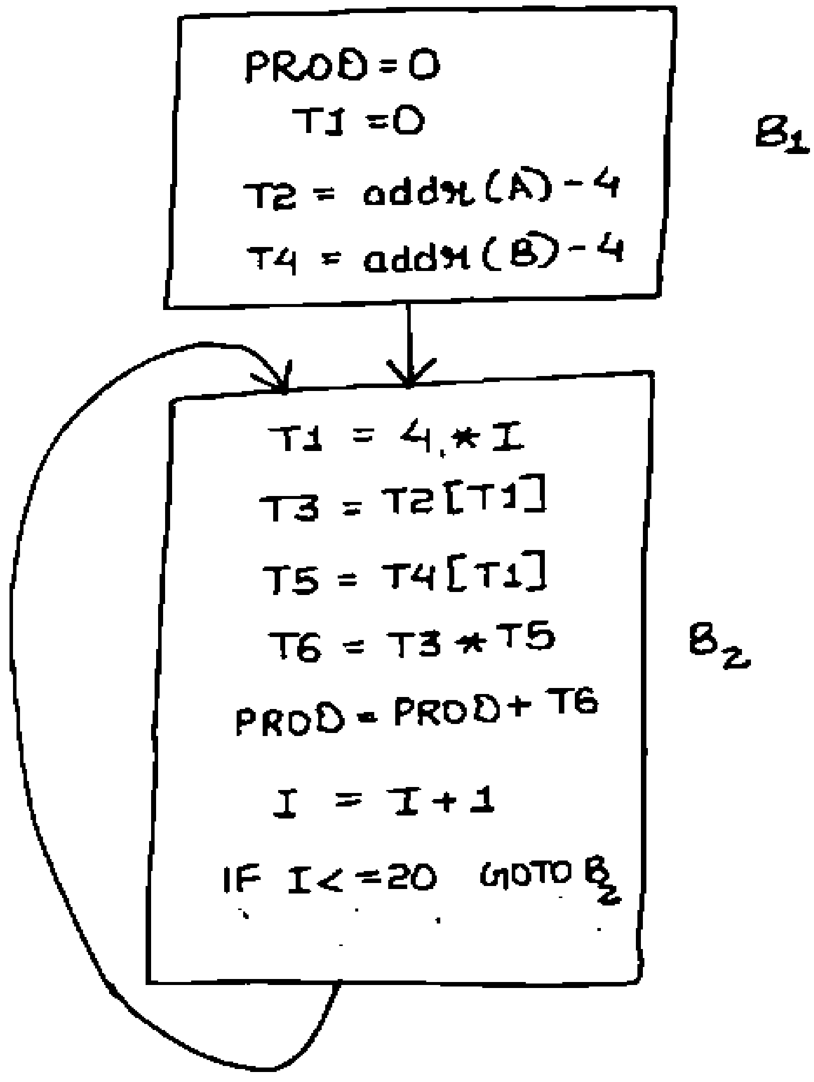
Flow Graph

- A directed graph in which the flow of control information is added to the basic blocks.

Rules:

  - Basic Blocks are the nodes of a flow graph
  - The block which has the initial statement of the code is the initial block.
  - An edge is drawn from B1 to B2, if B2 immediately follows B1

Flow Graph


PROD = 0
T1 = 0
T2 = addr(A) - 4
T4 = addr(B) - 4
B1
T1 = 4 * I
T3 = T2[T1]
T5 = T4[T1]
T6 = T3 * T5
PROD = PROD + T6
I = I + 1
IF I<=20 GOTO B2
B2

THE END!

Have a nice day!