JECRC Foundation

JAIPUR ENGINEERING COLLEGE AND RESEARCH CENTRE

JAIPUR ENGINEERING COLLEGE AND RESEARCH CENTRE

Year & Sem – 3rd Year & 5th Sem
Subject – COMPILER DESIGN
Unit – 5
Presented by – (Abhishek Dixit, Assistant Prof., Dept of CSE)

VISION AND MISSION OF INSTITUTE

To become a renowned center of outcome based learning and work towards academic, professional, cultural and social enrichment of the lives of individuals and communities

M1: Focus on evaluation of learning outcomes and motivate students to inculcate research aptitude by project based learning.

M2: Identify, based on informed perception of Indian, regional and global needs, the areas of focus and provide platform to gain knowledge and solutions.

M3: Offer opportunities for interaction between academia and industry.

M4: Develop human potential to its fullest extent so that intellectually capable and imaginatively gifted leaders can emerge in a range of professions.

VISION OF THE DEPARTMENT

To become renowned Centre of excellence in computer science and engineering and make competent engineers & professionals with high ethical values prepared for lifelong learning.

MISION OF THE DEPARTMENT

M1: To impart outcome based education for emerging technologies in the field of computer science and engineering.
M2: To provide opportunities for interaction between academia and industry.
M3: To provide platform for lifelong learning by accepting the change in technologies
M4: To develop aptitude of fulfilling social responsibilities

PROGRAM OUTCOMES

Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
Problem analysis: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM EDUCATIONAL OBJECTIVES

1. To provide students with the fundamentals of Engineering Sciences with more emphasis in Computer Science &Engineering by way of analyzing and exploiting engineering challenges.
2. To train students with good scientific and engineering knowledge so as to comprehend, analyze, design, and create novel products and solutions for the real life problems.
3. To inculcate professional and ethical attitude, effective communication skills, teamwork skills, multidisciplinary approach, entrepreneurial thinking and an ability to relate engineering issues with social issues.
4. To provide students with an academic environment aware of excellence, leadership, written ethical codes and guidelines, and the self motivated life-long learning needed for a successful professional career.
5. To prepare students to excel in Industry and Higher education by Educating Students along with High moral values and Knowledge

PROGRAM SPECIFIC OBJECTIVES

1. PSO1. Ability to interpret and analyze network specific and cyber security issues, automation in real word environment.

2. PSO2. Ability to Design and Develop Mobile and Web-based applications under realistic constraints.

COURSE OUTCOME

CO1: Compare different phases of compiler and design lexical analyzer.
CO2: Examine syntax and semantic analyzer by understanding grammars.
CO3: Illustrate storage allocation and its organization & analyze symbol table organization.
CO4: Analyze code optimization, code generation & compare various compilers.

CO-PO MAPPING

Semester	Subject	Code	L/T/P	CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
V	COMPILER DESIGN	5CS4-02	L	1. Compare different phases of compiler and design lexical analyzer..	3	3	3	3	1	1	1	1	1	2	1	3
			L	2. Examine syntax and semantic analyzer by understanding grammars.	3	3	3	2	1	1	1	0	1	2	1	3
			L	3. Illustrate storage allocation and its organization & analyze symbol table organization.	3	3	2	2	1	1	1	1	1	2	1	3
			L	4.Analyze code optimization, code generation & compare various compilers.	3	3	3	3	2	1	1	1	1	2	1	3

SYLLABUS

RAJASTHAN TECHNICAL UNIVERSITY, KOTA

Syllabus

III Year-V Semester: B.Tech. Computer Science and Engineering

5CS4-02: Compiler Design

Credit: 3 **Max. Marks: 150(IA:30, ETE:120)**

3L+0T+0P **End Term Exam: 3 Hours**

SN	Contents	Hours
1	**Introduction:**Objective, scope and outcome of the course.	01
2	**Introduction:** Objective, scope and outcome of the course. Compiler, Translator, Interpreter definition, Phase of compiler, Bootstrapping, Review of Finite automata lexical analyzer, Input, Recognition of tokens, Idea about LEX: A lexical analyzer generator, Error handling.	06
3	**Review of CFG Ambiguity of grammars:** Introduction to parsing. Top down parsing, LL grammars & passers error handling of LL parser, Recursive descent parsing predictive parsers, Bottom up parsing, Shift reduce parsing, LR parsers, Construction of SLR, Conical LR & LALR parsing tables, parsing with ambiguous grammar. Operator precedence parsing, Introduction of automatic parser generator: YACC error handling in LR parsers.	10
4	**Syntax directed definitions;** Construction of syntax trees, S-Attributed Definition, L-attributed definitions, Top down translation. Intermediate code forms using postfix notation, DAG, Three address code, TAC for various control structures, Representing TAC using triples and quadruples, Boolean expression and control structures.	10
5	**Storage organization;** Storage allocation, Strategies, Activation records, Accessing local and non-local names in a block structured language, Parameters passing, Symbol table organization, Data structures used in symbol tables.	08
6	**Definition of basic block control flow graphs;** DAG representation of basic block, Advantages of DAG, Sources of optimization, Loop optimization, Idea about global data flow analysis, Loop invariant computation, Peephole optimization, Issues in design of code generator, A simple code generator, Code generation from DAG.	07

DAG (Directed Acyclic Graph):

• Directed Acyclic Graph (DAG) is a form of syntax tree in which there is a single node for common sub expressions.

• It is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks.

• Leaf nodes represent identifiers, names or constants.

• Interior nodes represent operators.

• Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Applications-

DAGs are used for the following purposes-

To determine the expressions which have been computed more than once (called common sub-expressions).

To determine the names whose computation has been done outside the block but used inside the block.

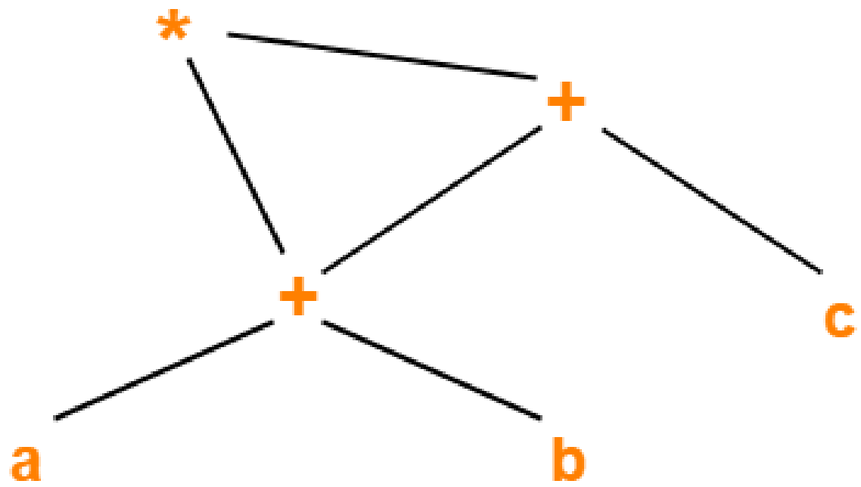
To determine the statements of the block whose computed value can be made available outside the block.

To simplify the list of Quadruples by not executing the assignment instructions x:=y unless they are necessary and eliminating the common sub-expressions.

Problem-01:

Consider the following expression and construct a DAG for it-

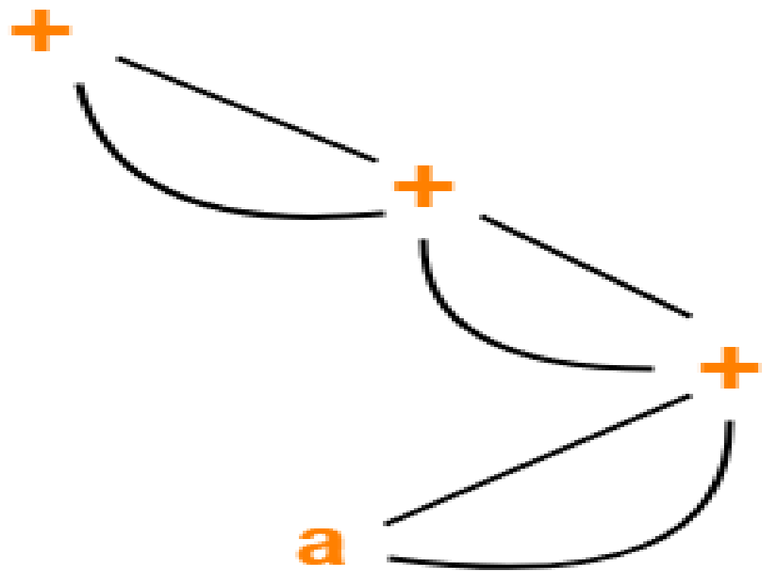
(a + b) x (a + b + c)



Directed Acyclic Graph

Problem-02:

Consider the following expression and construct a DAG for it-
(((a + a) + (a + a)) + ((a + a) + (a + a)))



Directed Acyclic Graph

BASIC BLOCKS:

Basic Block is a straight line code sequence which has no branches except to the entry and at the end respectively. Basic Block is a set of statements which always executes one after other, in a sequence.

The first task is to partition a sequence of three-address code into basic blocks. A new basic block is begun with the first instruction and instructions are added until a jump or a label is met. In the absence of jump control moves further consecutively from one instruction to another.

The following sequence of three address statements forms a basic block:

t1:= x * x
t2:= x * y
t3:= 2 * t2
t4:= t1 + t3
t5:= y * y
t6:= t4 + t5

Basic block construction:

Algorithm: Partition into basic blocks

Input: It contains the sequence of three address statements
Output: it contains a list of basic blocks with each three address statement in exactly one block
Method: First identify the leader in the code. The rules for finding leaders are as follows:
The first statement is a leader.
Statement L is a leader if there is an conditional or unconditional goto statement like: if....goto L or goto L
Instruction L is a leader if it immediately follows a goto or conditional goto statement like: if goto B or goto B

For each leader, its basic block consists of the leader and all statement up to. It doesn't include the next leader or end of the program.

https://www.youtube.com/watch?v=hzNFRDGZAE4

DAG Construction from basic block:

https://www.youtube.com/watch?v=fAjIlKmCuUY

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.

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.

- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.

- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Optimization can be categorized broadly into two types :

1. Machine independent
2. Machine dependent.

Machine-independent Optimization:

In this optimization, 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.

For example:

```
do
{
item = 10;
value = value + item;
}
while(value<100);
```

This code involves repeated assignment of the identifier item, which if we put this way:

```
Item = 10;
do
{
value = value + item;
}
while(value<100);
```

should not only save the CPU cycles, but can be used on any processor.

Machine-dependent Optimization:

Machine-dependent optimization is done after the target code has been generated and 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.

Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.

Loop Optimization:

Most programs run as a loop in the system. It becomes necessary to optimize the loops in order to save CPU cycles and memory. Loops can be optimized by the following techniques:

Invariant code : A fragment of code that resides in the loop and computes the same value at each iteration is called a loop-invariant code. This code can be moved out of the loop by saving it to be computed only once, rather than with each iteration.

```
While (i<limit-2)
a=limit-2
While(i<a)
```

Induction analysis : A variable is called an induction variable if its value is altered within the loop by a loop-invariant value.

Strength reduction : There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.

Global data flow analysis

- To efficiently optimize the code compiler collects all the information about the program and distribute this information to each block of the flow graph. This process is known as data-flow graph analysis.
- Certain optimization can only be achieved by examining the entire program. It can't be achieve by examining just a portion of the program.
- For this kind of optimization user defined chaining is one particular problem.
- Here using the value of the variable, we try to find out that which definition of a variable is applicable in a statement.

Based on the local information a compiler can perform some optimizations. For example, consider the following code:

```
x = a + b;
   x = 6 * 3
```

In this code, the first assignment of x is useless. The value computer for x is never used in the program.
At compile time the expression 6*3 will be computed, simplifying the second assignment statement to x = 18;

Some optimization needs more global information. For example, consider the following code:

```
a = 1;
   b = 2;
   c = 3;
   if (....) x = a + 5;
   else x = b + 4;
   c = x + 1;
```

In this code, at line 3 the initial assignment is useless and x +1 expression can be simplified as 7.

But it is less obvious that how a compiler can discover these facts by looking only at one or two consecutive statements. A more global analysis is required so that the compiler knows the following things at each point in the program:

1. Which variables are guaranteed to have constant values
2. Which variables will be used before being redefined

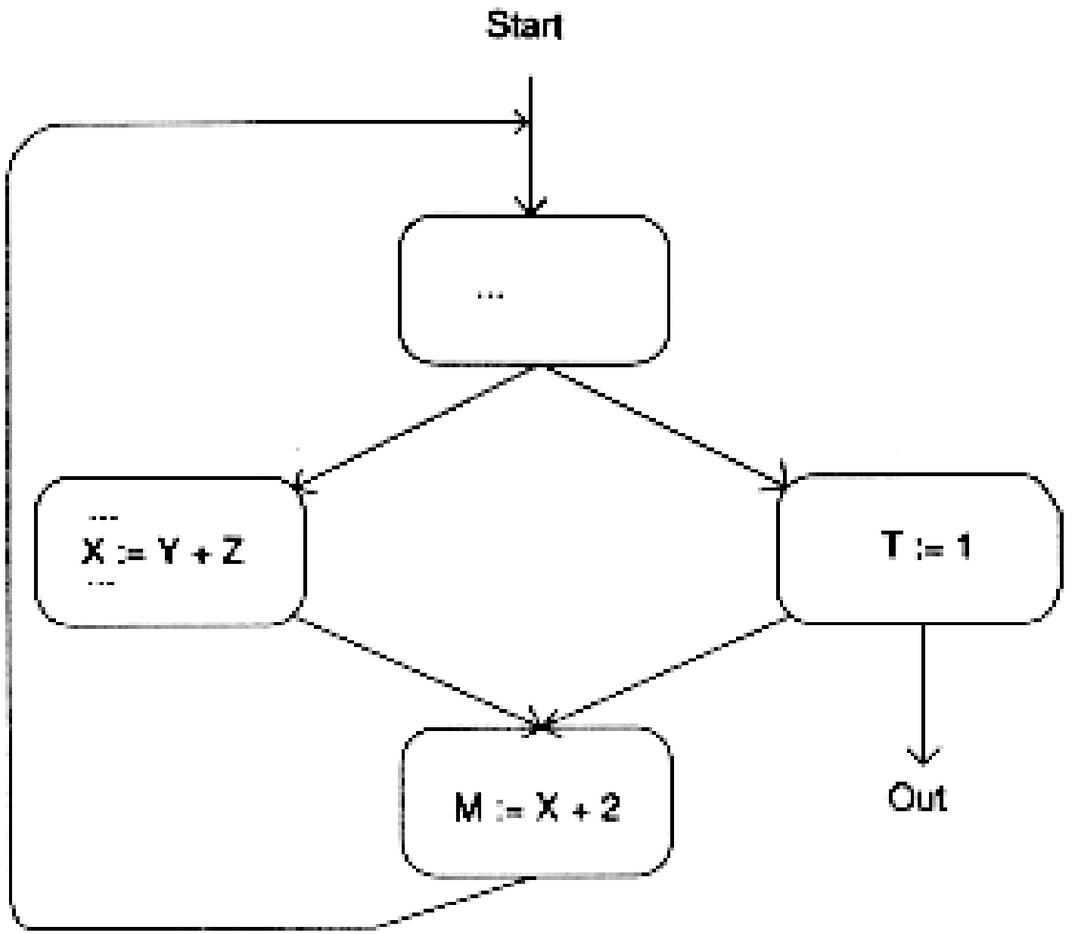
Data flow analysis is used to discover this kind of property. The data flow analysis can be performed on the program's control flow graph (CFG).

The control flow graph of a program is used to determine those parts of a program to which a particular value assigned to a variable might propagate.

Loop-Invariant Computations

Loop-invariant statements are those statements within a loop which produce the same value each time the loop is executed. We identify them and then move them outside the loop.

EXAMPLE 7 Loop invariants



In Example 7, *T := 1* is loop-invariant (presuming no assignment to *T* in the header) and may be moved outside the loop. The statement *X := Y + Z* is not loop invariant (why?).

Peephole Optimization

Peephole optimization is a type of Code Optimization performed on a small part of the code. It is performed on the very small set of instructions in a segment of code.

The small set of instructions or small part of code on which peephole optimization is performed is known as ***peephole*** *or* ***window****.*

It basically works on the theory of replacement in which a part of code is replaced by shorter and faster code without change in output.

The objective of peephole optimization is:

1. To improve performance

2. To reduce memory footprint

3. To reduce code size

Peephole Optimization Techniques:

1. **Redundant load and store elimination:**
   In this technique the redundancy is eliminated.

Initial code:

```
y = x + 5;
i = y;
z = i;
w = z * 3;
```

Optimized code:

```
y = x + 5;
 i = y;
w = y * 3;
```

2. Constant folding:

The code that can be simplified by user itself, is simplified.

Initial code:

```
x = 2 * 3;
```

Optimized code:

```
x = 6;
```

3. Strength Reduction:

The operators that consume higher execution time are replaced by the operators consuming less execution time.

Initial code:

y = x * 2;

Optimized code:

y = x + x; or y = x << 1;

Initial code:

y = x / 2;

Optimized code:

y = x >> 1;

4. Null sequences:
Useless operations are deleted.

5. Combine operations:
Several operations are replaced by a single equivalent operation.

Issues in the design of a code generator:

Code generator converts the intermediate representation of source code into a form that can be readily executed by the machine. A code generator is expected to generate the correct code. Designing of code generator should be done in such a way so that it can be easily implemented, tested and maintained.

The following issue arises during the code generation phase:

1. **Input to code generator –**
   The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's, etc. The code generation phase just proceeds on an assumption that the input are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.

2. Target program –

The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language.

Absolute machine language as output has advantages that it can be placed in a fixed memory location and can be immediately executed.

Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader. But there is added expense of linking and loading.

Assembly language as output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code. And we need an additional assembly step after code generation.

3. **Memory Management –**
   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.

4. **Instruction selection –**
   Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also plays a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straight-forward

5. Register allocation issues –

Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers are subdivided into two subproblems:
During **Register allocation –** we select only those set of variables that will reside in the registers at each point in the program.
During a subsequent **Register assignment** phase, the specific register is picked to access the variable.
As the number of variables increases, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain machine requires register pairs consist of an even and next odd-numbered register.
For example
M a, b
These types of multiplicative instruction involve register pairs where the multiplicand is an even register and b, the multiplier is the odd register of the even/odd register pair.

6. Evaluation order –
The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results. However, picking the best order in the general case is a difficult NP-complete problem.

7. 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

Code Generator:

Code generator is used to produce the target code for three-address statements. It uses registers to store the operands of the three address statement.

Example:
Consider the three address statement x:= y + z. It can have the following sequence of codes:

MOV x, R_0

ADD y, R_0

Register and Address Descriptors:

A register descriptor contains the track of what is currently in each register. The register descriptors show that all the registers are initially empty.

An address descriptor is used to store the location where current value of the name can be found at run time.

A code-generation algorithm:

The algorithm takes a sequence of three-address statements as input. For each three address statement of the form a:= b op c perform the various actions. These are as follows:

1. Invoke a function getreg to find out the location L where the result of computation b op c should be stored.

2. Consult the address description for y to determine y'. If the value of y currently in memory and register both then prefer the register y' . If the value of y is not already in L then generate the instruction **MOV y' , L** to place a copy of y in L.

3. Generate the instruction **OP z' , L** where z' is used to show the current location of z. if z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all other descriptor.

4. If the current value of y or z have no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of x : = y op z those register will no longer contain y or z.

Generating Code for Assignment Statements:
The assignment statement d:= (a-b) + (a-c) + (a-c) can be translated into the following sequence of three address code:

t:= a-b
u:= a-c
v:= t +u
d:= v+u

Code sequence for the example is as follows:

Statement	Code Generated	Register descriptor Register empty	Address descriptor
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 R1
d:= v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

Code Generation from DAG:

Code generation can be considered as the final phase of compilation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language.

We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

It should carry the exact meaning of the source code.
It should be efficient in terms of CPU usage and memory management.
We will now see how the intermediate code is transformed into target object code (assembly code, in this case).

Directed Acyclic Graph:

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

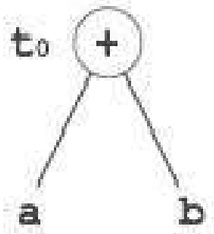
Leaf nodes represent identifiers, names or constants.

Interior nodes represent operators.

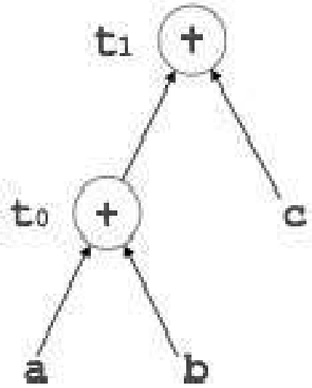
Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:

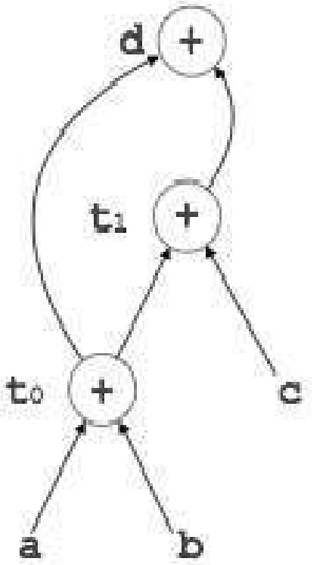
```
t0 = a + b
t1 = t0 + c
d = t0 + t1
```



[$t_0 = a + b$]



[$t_1 = t_0 + c$]



[$d = t_0 + t_1$]

REFERENCES/BIBLOGRAPHY

1. www.slideshare.com
2. Codingninjas.com
3. Javapoint.com
4. Gatevidyalaya.com

JECRC Foundation

JAIPUR ENGINEERING COLLEGE AND RESEARCH CENTRE

Thank you!