JECRC Foundation

JAIPUR ENGINEERING COLLEGE AND RESEARCH CENTRE

Year & Sem – 3rd Year & 5th Sem

Subject – COMPILER DESIGN

Unit – 3

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

Syntax Directed Translation in Compiler Design

- Parser uses a CFG(Context-free-Grammer) to validate the input string and produce output for next phase of the compiler. Output could be either a parse tree or abstract syntax tree. Now to interleave semantic analysis with syntax analysis phase of the compiler, we use Syntax Directed Translation.

- The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree.

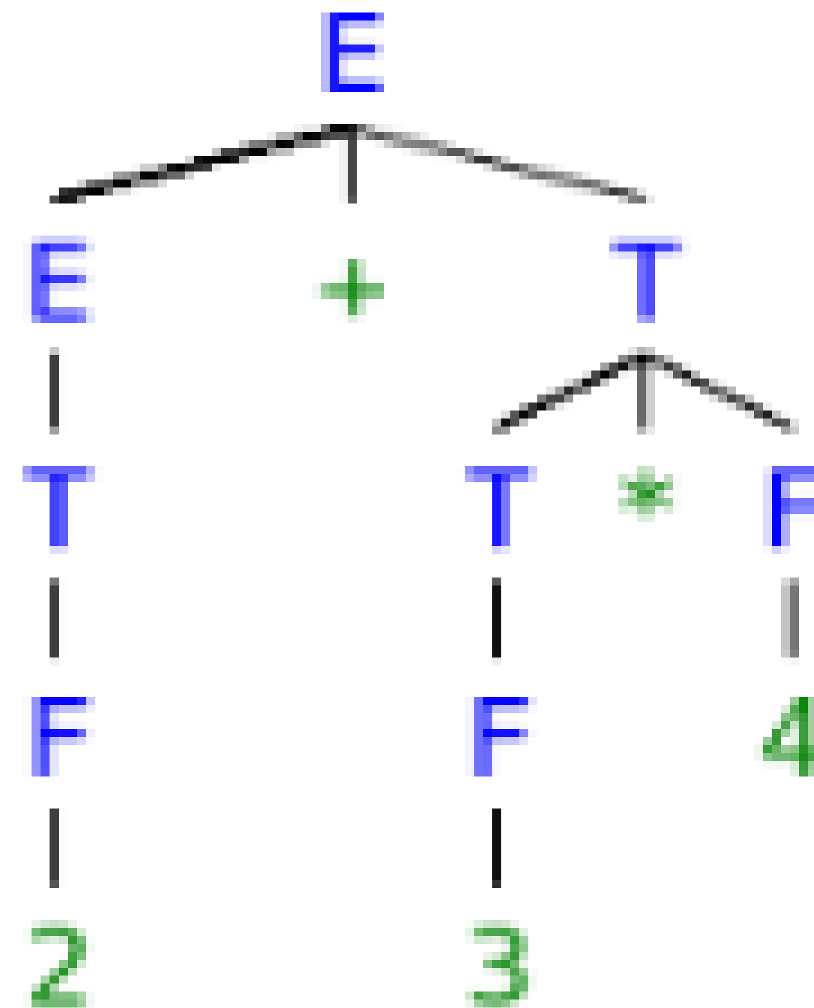
Example
E -> E+T | T
T -> T*F | F
F -> INTLIT

This is a grammar to syntactically validate an expression having additions and multiplications in it. Now, to carry out semantic analysis we will augment SDT rules to this grammar, in order to pass some information up the parse tree and check for semantic errors, if any. In this example we will focus on evaluation of the given expression, as we don’t have any semantic assertions to check in this very basic example.

E -> E+T { E.val = E.val + T.val } PR#1
E -> T { E.val = T.val } PR#2
T -> T*F { T.val = T.val * F.val } PR#3
T -> F { T.val = F.val } PR#4
F -> INTLIT { F.val = INTLIT.lexval } PR#5

For understanding translation rules further, we take the first SDT augmented to [E -> E+T] production rule. The translation rule in consideration has val as attribute for both the non-terminals – E & T. Right hand side of the translation rule corresponds to attribute values of right side nodes of the production rule and vice-versa. Generalizing, SDT are augmented rules to a CFG that associate 1) set of attributes to every node of the grammar and 2) set of translation rules to every production rule using attributes, constants and lexical values.

Let’s take a string to see how semantic analysis happens – S = 2+3*4. Parse tree corresponding to S would be



S – attributed and L – attributed SDTs in Syntax directed translation

Types of attributes –

Attributes may be of two types – Synthesized or Inherited.

Synthesized attributes –

A Synthesized attribute is an attribute of the non-terminal on the left-hand side of a production. Synthesized attributes represent information that is being passed up the parse tree. ***The attribute can take value only from its children (Variables in the RHS of the production).*** For eg. let's say A -> BC is a production of a grammar, and A's attribute is dependent on B's attributes or C's attributes then it will be synthesized attribute.

Inherited attributes –

An attribute of a nonterminal on the right-hand side of a production is called an inherited attribute. The attribute can take value either from its parent or from its siblings (variables in the LHS or RHS of the production).For example, let's say A -> BC is a production of a grammar and B's attribute is dependent on A's attributes or C's attributes then it will be inherited attribute.

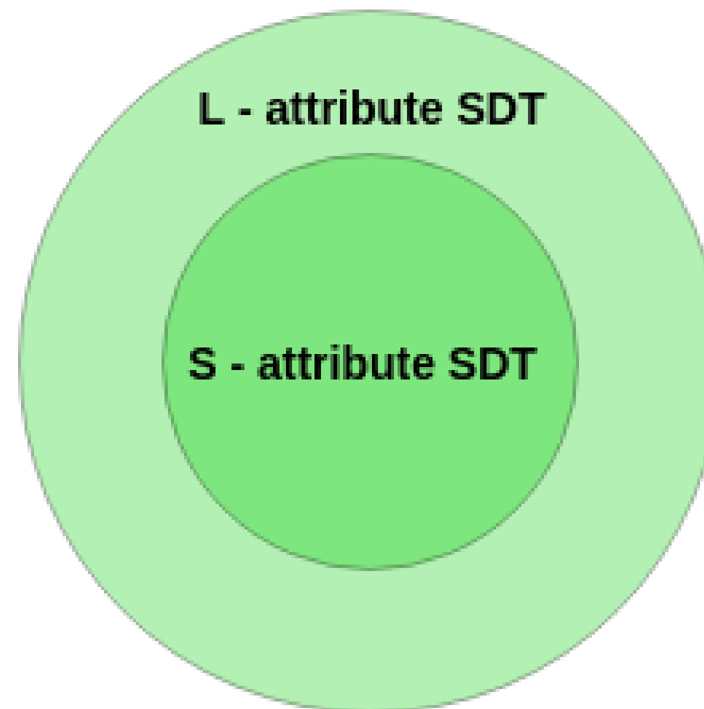
Now, let’s discuss about S-attributed and L-attributed SDT.

S-attributed SDT :
If an SDT uses only synthesized attributes, it is called as S-attributed SDT.
S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.
Semantic actions are placed in rightmost place of RHS.
L-attributed SDT:
If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed SDT.
Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.
Semantic actions are placed anywhere in RHS.

For example,
A -> XYZ {Y.S = A.S, Y.S = X.S, Y.S = Z.S}

is not an L-attributed grammar since Y.S = A.S and Y.S = X.S are allowed but Y.S = Z.S violates the L-attributed SDT definition as attributed is inheriting the value from its right sibling.

Note – If a definition is S-attributed, then it is also L-attributed but **NOT** vice-versa.



Example – Consider the given below SDT.

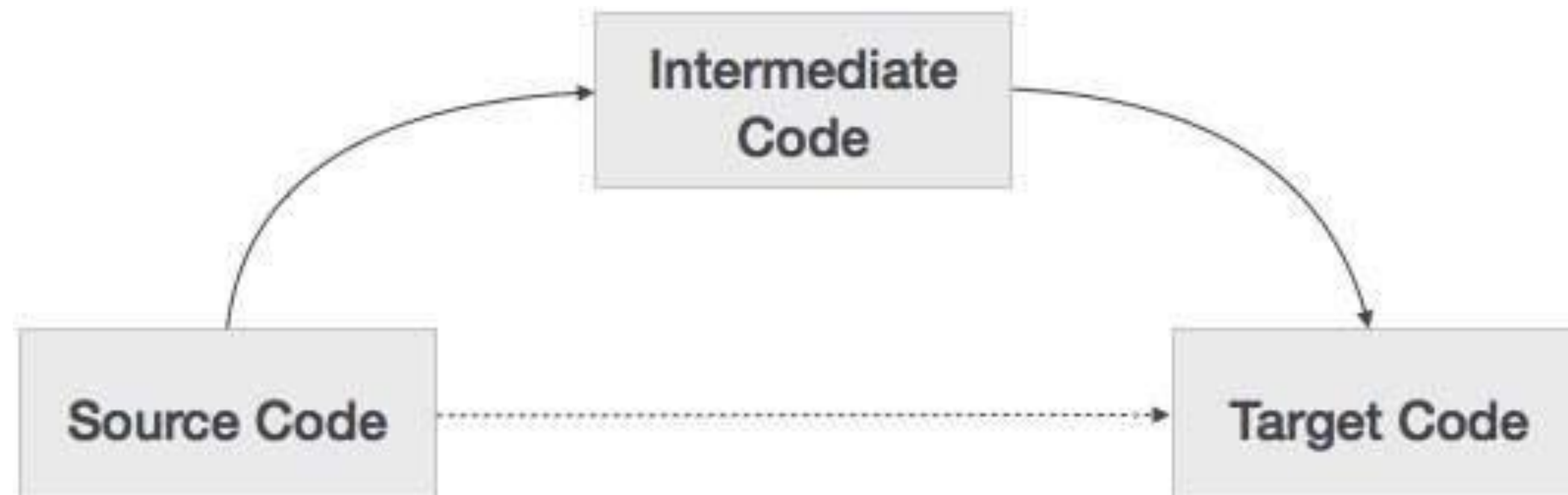
P1: S -> MN {S.val= M.val + N.val}
P2: M -> PQ {M.val = P.val * Q.val and P.val =Q.val}

Select the correct option.

A. Both P1 and P2 are S attributed.
B. P1 is S attributed and P2 is L-attributed.
C. P1 is L attributed but P2 is not L-attributed.
D. None of the above

Intermediate Code Generation:

A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code? Let us see the reasons why we need an intermediate code.



- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.

- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.

- The second part of compiler, synthesis, is changed according to the target machine.

- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

1. **Postfix Notation –**

    The ordinary (infix) way of writing the sum of a and b is with operator in the middle : a + b
    The postfix notation for the same expression places the operator at the right end as ab +. In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +. No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression. In postfix notation the operator follows the operand.

Example – The postfix representation of the expression (a – b) * (c + d) + (a – b) is :   ab – cd + *ab -+.

2. Three-Address Code –

A statement involving no more than three references(two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form x = y op z , here x, y, z will have address (memory location). Sometimes a statement might contain less than three references but it is still called three address statement.

Example – The three address code for the expression a + b * c + d :

T 1 = b * c

T 2 = a + T 1

T 3 = T 2 + d

T 1 , T 2 , T 3 are temporary variables.

For example:

a = b + c * d;

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

r1 = c * d;

r2 = b + r1;

a = r2

A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms : quadruples and triples

Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format:

Op	arg_1	arg_2	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

Triples

Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

Triples face the problem of code immovability while optimization, as the results are positional and changing the order or position of an expression may cause problems.

Op	arg_1	arg_2
*	c	d
+	b	0
+	1	0
=	2	

3. DAG

DAG stands for Directed Acyclic Graph.

A DAG is an Abstract Syntax Tree(AST) with a unique node for each value.

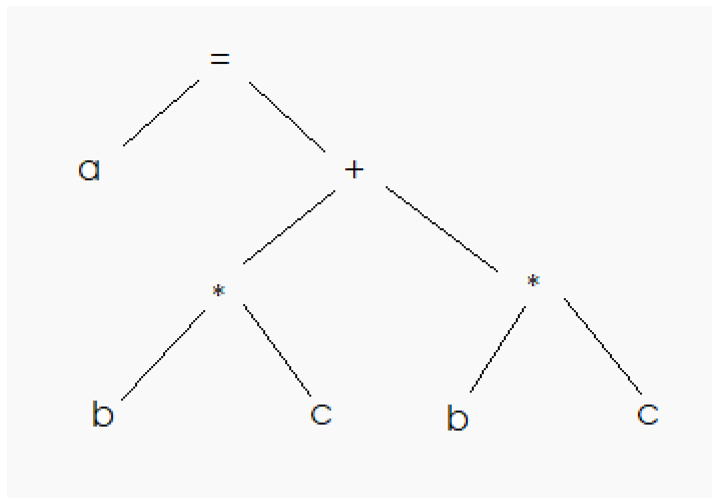
It is a tool that depicts the structure of basic blocks.

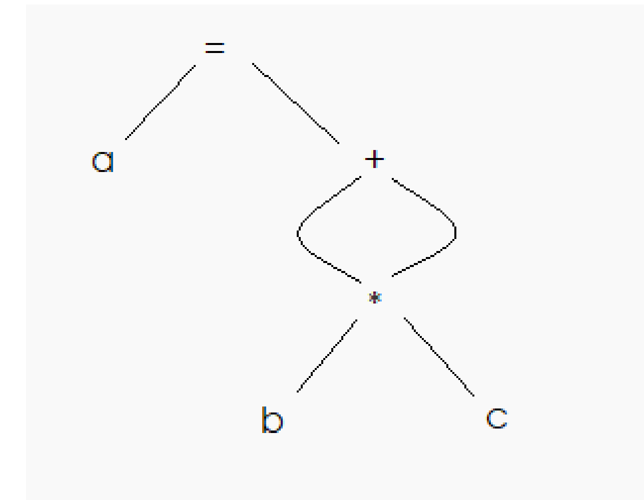
Like a syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators.

Syntax tree and DAG for a = (b * c) + (b * c)

Syntax Tree

DAG





Syntax Tree and DAG both are graphical representations.
Syntax tree does not find the common sub-expressions whereas DAG
can. Thus, DAG not only represents expressions briefly,
it gives the compiler important clues regarding the generation of
efficient code to evaluate the expressions.

Construction of DAGs

Rule-01:

In a DAG,
Interior nodes always represent the operators.
Exterior nodes (leaf nodes) always represent the names, identifiers or constants.
Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Rule-02:

While constructing a DAG,
A check is made to find if there exists any node with the same value.
A new node is created only when there does not exist any node with the same value.
This action helps in detecting the common sub-expressions and avoiding the re-computation of the same.

Rule-03:

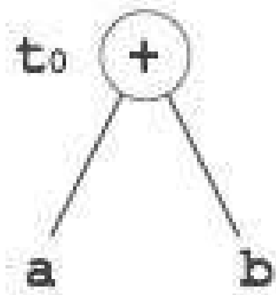
The assignment instructions of the form x:=y are not performed unless they are necessary.

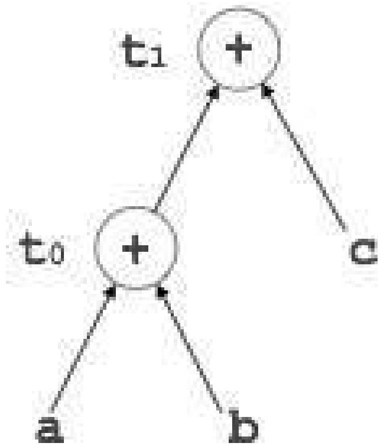
Q : Construct a DAG for the expression : (a+b)*(a+b+c)

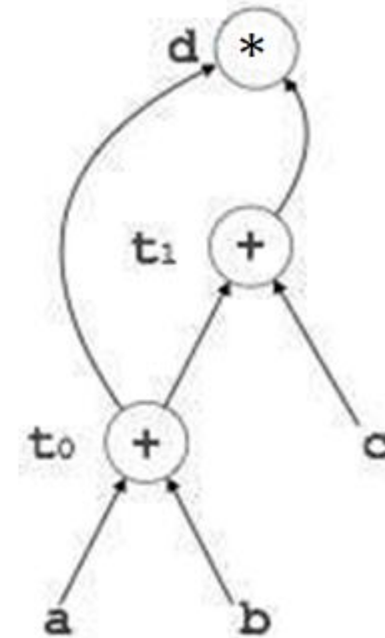
Sol : The Three Address Code for the given expression is -

$t_0 = a + b$ **$t_1 = t_0 + c$** **$d = t_0 * t_1$**

Now, Directed Acyclic Graph is-



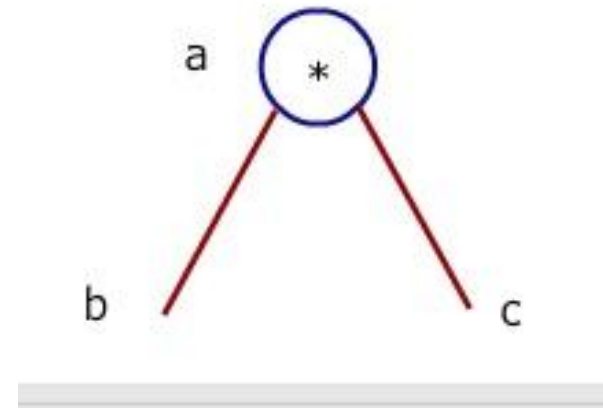




From the constructed DAG, we observe-

- The common sub-expression (a+b) has been expressed into a single node in the DAG.
- The computation is carried out only once and stored in the identifier t_0 and reused later.

Examples



Consider the following three address code statements.

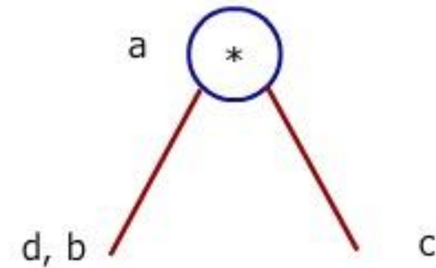
a = b * c
d = b
e = d * c
b = e
f = b + c
g = f + d

Step 1

Consider the first statement, i.e., a = b * c. Create a leaf node with label b and c as left and right child respectively and parent of it will be *. Append resultant variable a to the node *.

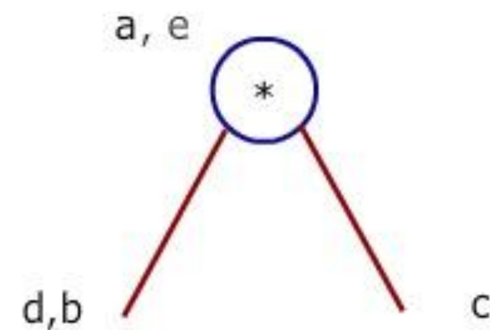
Step 2

For second statement, i.e., d = b, node b is already created. So, append d to this node.

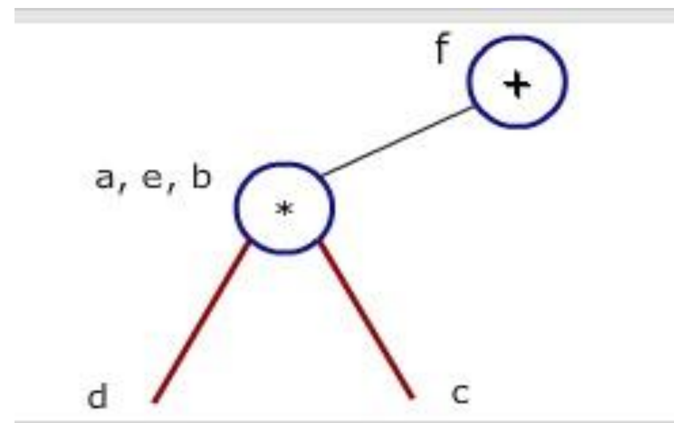


Step 3

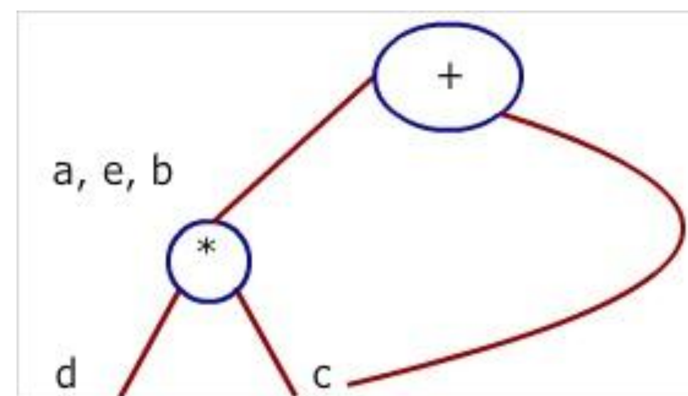
For third statement e = d * c, the nodes for d, c and * are already create. Node e is not created, so append node e to node *.



Step 4
For fourth statement b = e, append b to node e.

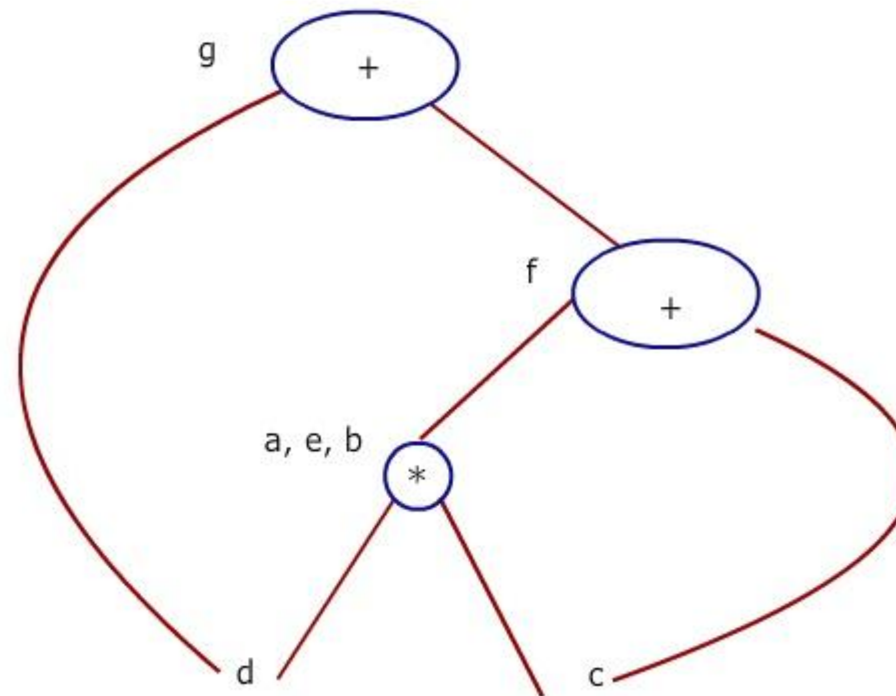


Step 5
For fifth statement f = b + c, create a node for operator + whose left child b and right child c and append f to newly created node +.



Step 6

For last statement g = f + d, create a node for operator + whose left child d and right child f and append g to newly created node +.

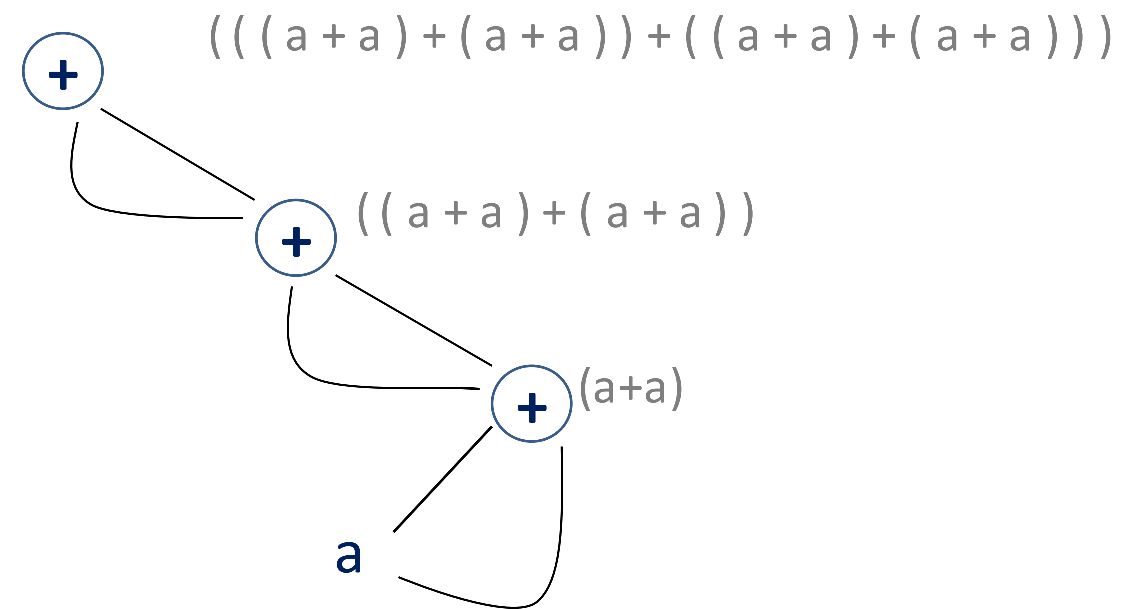


Thus we have the desired Directed Acyclic Graph

Q: Consider the following expression and construct a DAG for it-

(((a + a) + (a + a)) + ((a + a) + (a + a)))

Sol:



Directed Acyclic Graph

APPLICATIONS

Determines the common sub-expressions (expressions computed more than once).

It is a useful data structure for implementing transformation on basic blocks.

A basic block can be optimized by the construction of DAG.

A DAG is usually constructed using Three Address Code (TAC).

Transformations such as dead code elimination and common sub expression elimination are then applied.

It gives a picture representation of how the value computed by the statement is used in subsequent statements.

To simplify the list of Quadruples by not executing the assignment instructions x=y unless they are necessary.

BOOLEAN EXPRESSIONS

A Boolean Expression is an expression that results in a Boolean value, that is, in a value of either true or false. A Boolean Expression may be composed of a combination of Boolean constants true or false.

A Boolean Expression can compare data of any type as long as both parts of the expression have the same basic data types.

Boolean expression are used for statements changing the flow of control.

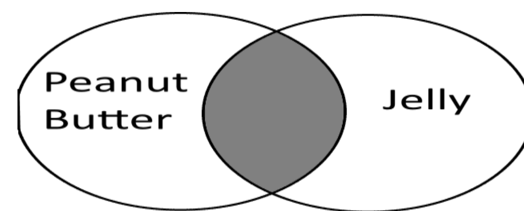
Evaluation of Boolean expression can be optimized if it is sufficient to evaluate a part of the expression that determines its value.

When translating Boolean expression into TAC we use two methods:
Numerical Method and Jump Method.

IN Numerical method we assign numerical values to true and false and evaluate the expression analogously to an arithmetic expression.

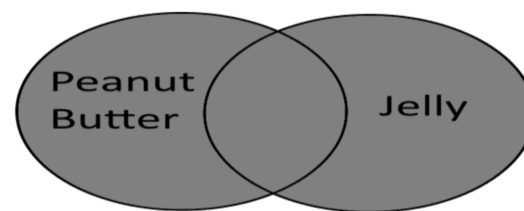
In Jump method we evaluate a Boolean expression E as a sequence of conditional
And unconditional jump to location E.true(if E is true) or to E.false.

There are three Boolean operators: “AND”, “OR” and “NOT”.



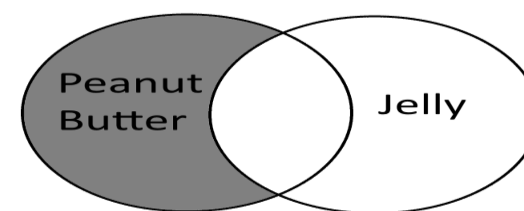
AND

Using AND, this search would only retrieve results with Peanut Butter and Jelly.



OR

Using OR, this search would retrieve results with peanut butter, with jelly, and with both.



NOT

Using NOT, this search would retrieve results with peanut butter, and exclude those with jelly or PB with jelly.

In Compiler Design Boolean Expression has two primary purposes :

1. It is used for computing logical values.
2. It is used as conditional expressions using if then-else or while-do.

Grammer used in Boolean Expression

- E → E or E
- E → E and E
- E → Not E
- E → (E)
- E → id relop id
- E → True
- E → False

The relop is denoted by <,>.

Here, relop is Relational Operator.

Production rule	Semantic actions
E → E1 OR E2	{ E.place = newtemp(); Emit (E.place ':=' E1.place 'OR' E2.place) }
E → E1 + E2	{ E.place = newtemp(); Emit (E.place ':=' E1.place 'AND' E2.place)}
E → NOT E1	{ E.place = newtemp(); Emit (E.place ':=' 'NOT' E1.place)}
E → (E1)	{ E.place = E1.place}

```
E → id relop id2          { E.place = newtemp();
                            Emit ('if' id1.place relop.op id2.place 'goto' nextstar + 3);
                            EMIT (E.place ':=' '0')
                            EMIT ('goto' nextstat + 2)
                            EMIT (E.place ':=' '1')}

E → TRUE                  { E.place := newtemp();
                            Emit (E.place ':=' '1')}

E → FALSE                 { E.place := newtemp();
                            Emit (E.place ':=' '0') }
```

The Emit function is used to generate the three address code and the newtemp() function is used to generate the temporary variables.
The E → id relop id2 contains the next_state and it gives the index of next three address statements in the output sequence.

Here is the example which generates the three address code using the above translation scheme :

p>q AND r<s OR u>r

```
100: if p>q goto 103
101: t1:=0
102: goto 104
103: t1:=1
104: if r>s goto 107
105: t2:=0
106: goto 108
107: t2:=1
108: if u>v goto 111
109: t3:=0
110: goto 112
111: t3:= 1
112: t4:= t1 AND t2
113: t5:= t4 OR t3
```

CONTROL STRUCTURE

Control Structure is like a block of programming that analysis variables and chooses
a direction in which to go based on given parameters.

The term control flow details the direction the program takes, hence it is the decision
Making process in computing.

Control Flow is the order in which individual statements, instructions, or function calls
of an imperative program are executed or evaluated.

Statements that alter the flow of control:

The goto statement alters the flow of control. If we implement goto statements then we need to define a LABEL for a
statement. A production can be added for this purpose:

S → LABEL : S
LABEL → id

In this production system, semantic action is attached to record the LABEL and its value in the symbol table.

Basic Terminologies

Those initial conditions and parameters are called preconditions. Preconditions are the state of variables before entering a control structure. Based on those preconditions, the computer runs an algorithm (the control structure) to determine what to do. The result is called a post condition. Post conditions are the state of variables after the algorithm is run.

An Example

Let us analyze flow control by using traffic flow as a model. A vehicle is arriving at an intersection. Thus, the precondition is the vehicle is in motion. Suppose the traffic light at the intersection is red. The control structure must determine the proper course of action to assign to the vehicle.

Precondition

The vehicle is in motion.
Treatments of Information through Control Structures
Is the traffic light green? If so, then the vehicle may stay in motion.
Is the traffic light red? If so, then the vehicle must stop.
End of Treatment

Post condition

The vehicle comes to a stop.
Thus, upon exiting the control structure, the vehicle is stopped.

can also be form as a structure

Following grammar used to incorporate structure flow of control statement:

S → if E then S
S → if E then S else S
S → while E do S
S → begin L end
S → A
L → L ; S
L → S

Here, S is a statement, L is a statement-list, A is an assignment statement and E is a Boolean-valued expression.

IF-THEN-ELSE Statement

IF-THEN statements test for only one action. With an IF-THEN-ELSE statement, the control can "look both ways" so to speak, and take a secondary course of action. If the condition is true, then an action occurs. If the condition is false, take an alternate action. To illustrate:

IF variable is true
THEN take this course of action
ELSE call another routine
In this case, if the variable is true, it takes a certain course of action and completely skips the ELSE clause. If the variable is false, the control structure calls a routine and completely skips the THEN clause.
Note that you can combine ELSE's with other IF's, allowing several tests to be made. In an IF-THEN-ELSEIF-THEN-ELSEIF-THEN-ELSEIF-THEN structure, tests will stop as soon as a condition is true. That's why you'd probably want to put the most "likely" test first, for efficiency (Remembering that ELSE's are skipped if the first condition is true, meaning that the remaining portions of the IF-THEN-ELSEIF... would not be processed). eg:
In case your computer doesn't start
IF a floppy disk is in the drive
THEN remove it and restart
ELSE IF you don't have any OS installed
THEN install an OS
ELSE call the hotline
You can have as many ELSE IF's as you like.

DO-WHILE Loops

A DO-WHILE loop is nearly the exact opposite to a WHILE loop. A WHILE loop initially checks to see if the parameters have been satisfied before executing an instruction. A DO-WHILE loop executes the instruction before checking the parameters. To illustrate:

DO Add 1 to X
WHILE X is not equal 9

As you can see, the example differs from the first illustration, where the DO action is taken before the WHILE. The WHILE is inclusive in the DO. As such, if the WHILE results in a false (X is equal to 9), the control structure will break and will not perform another DO. Note that if X is equal to or greater than 9 prior to entering the DO-WHILE loop, then the loop will never terminate.

FLOW OF CONTROL STATEMENTS WITH THE JUMP METHOD

We will consider the following rules:

s → if E then S1
s → if E then S1 else S2
s → while E repeat S1

In each of these productions, E is the boolean expression to be translated. The Boolean expression E is associated with two labels (that are inherited attributes in the following semantic rules)

1. E.true the label to which control flows if E is true,
2. E.false the label to which control flows if E is false.

In each of these productions, S is a flow of control statement associated with two attributes

1. S.next which is a label that is attached to the first 3-address statement to be executed after the code for S , S.next is an inherited attribute,
2. S.code is the translation code for S, as usual it is a synthesized attribute.

Production	Semantic Rule
$S \longmapsto$ **if** E **then** S_1	$E.true := newlabel$
	$E.false := S.next$
	$S_1.next := S.next$
	$S.code := E.code \mid\mid generate(E.true\ ':') \mid\mid S_1.code$
$S \longmapsto$ **if** E **then** S_1 **else** S_2	$E.true := newlabel$
	$E.false := newlabel$
	$S_1.next := S.next$
	$S_2.next := S.next$
	$code_1 := E.code \mid\mid generate(E.true\ ':') \mid\mid S_1.code$
	$code_2 := generate(\textbf{'goto'}\ S.next) \mid\mid$
	$code_3 := generate(E.false\ ':') \mid\mid S_2.code$
	$S.code := code_1 \mid\mid code_2 \mid\mid code_3$
$S \longmapsto$ **while** E **repeat** S_1	$S.begin := newlabel$
	$E.true := newlabel$
	$E.false := S.next$
	$S_1.next := S.begin$
	$code_1 := generate(S.begin\ ':') \mid\mid E.code$
	$code_2 := generate(E.true\ ':') \mid\mid S_1.code$
	$code_3 := generate(\textbf{'goto'}\ S.begin)$
	$S.code := code_1 \mid\mid code_2 \mid\mid code_3$

REFERENCES/BIBLOGRAPHY

1. The DAG Representation of Basic Blocks
www.facweb.iitkgp.ac.in

2. geeksforgeeks.com

3. Directed Acyclic Graphs | DAGs | Examples | Gate Vidyalay
www.gatevidyalay.com

JECRC Foundation

JAIPUR ENGINEERING COLLEGE AND RESEARCH CENTRE

Thank you!