Compiler Design
Chapter 3:RTE

GATE CS Lectures
by Monalisa

MonalisaCS

- **Section 7: Compiler Design**
- Lexical analysis, parsing, syntax-directed translation. Runtime environments. Intermediate code generation . Local optimization, Data flow analyses: constant propagation, liveness analysis, common subexpression elimination.

- Chapter 1: Introduction to Compiler [Language processing System ,Compiler ,Phases of Compiler , Lexical Analysis]
- Chapter 2: Parsing [Syntax Analysis , CFG, Ambiguous Grammar , Recursive Grammar ,Left Factoring ,Top down parser : LL(1),FIRST & FOLLOW , Bottom up parser : shift-reduce parsing ,LR(0),SLR(1),CLR(1), LALR(1), Operator Precedence grammar]
- Chapter 3:SDT,Code generation & optimization [syntax-directed translation. Runtime environments. Intermediate code generation . Local optimization, Data flow analyses: constant propagation, liveness analysis, common subexpression elimination.]

MonalisaCS

Syntax Directed Translation:

- A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules or actions.
- Attributes are associated with grammar symbols and rules are associated with productions.
- If X is a symbol and a is one of its attributes , then we write X.a to denote the value of a at a particular parse-tree node labeled X.
- Ex : S→AB{print (“TOC”)} , A →a{print (“CD”)}, B →b{print (“Alg”)}
- **Application:**

1) Constructing annotated parse tree.
2) Store type information into symbol table.
3) To evaluate algebraic expression.
4) To verify variable declaration. type checking ,type conversion ,type equivalence etc.
5) To convert postfix or prefix notation.
6) To generate intermediate code & target code.
7) To verify proper use of operator.
8) To construct DAG.

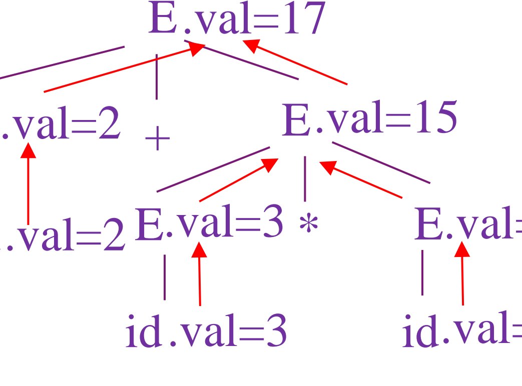
MonalisaCS

https://monalisacs.com/

Input → SDT → Output

Parse Tree → SDT → Annotated Parse Tree

- Annotation=a note of explanation or comment added to a text or diagram.
- Attaching attribute to the identifier.
- **Annotated / decorated parse tree:**
- A parse tree, showing the value(s) of its attribute(s) is called an annotated parse tree.
- Before we can evaluate an attribute at a node of a parse tree, we must evaluate all the attributes upon which its value depends.
- Ex:
- **Production** **Semantic rules**
- E→E+E {$E.val=E_1.val+E_2.val$}
- E→ E*E {$E.val=E_1.val*E_2.val$}
- E→ id {E.val=id.val}
- Input=2+3*5
- Output=17



MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

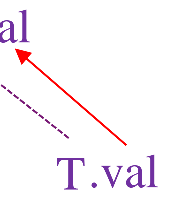
- **Classification of Attributes:**
- Based on process of evaluation attribute are classified into two type.
- Synthesized and Inherited Attributes
- 1.The attribute whose value is evaluated from the attribute values of children is called as **synthesized attribute**.
- A synthesized attribute for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. A must be head.
- A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.
- 2. The attribute whose value is evaluated from the attribute values of parent or sibling is called as **inherited attribute**.
- An inherited attribute for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N.
- Note that the production must have B as a symbol in its body.
- An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a postorder traversal of the parse tree.
- With Inherited attributes, we can evaluate attributes in any top-down order, such as that of a preorder traversal of the parse tree.
- **Evaluation Orders for SDD's:**
- "Dependency graphs" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree.
- While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.
- A dependency graph depicts the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first is needed to compute the second.
- Edges express constraints implied by the semantic rules
- Ex: **Production** **Semantic Rule**
- $E \rightarrow E_1 + T$ $E.val = E_1.val + T.val$
- E.val is synthesized from E_1.val and T.val
- parse tree edges as dotted lines, while the
- edges of the dependency graph are solid.



MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- If there is any cycle in the graph, then there are no topological sorts; that is , there is no way to evaluate the SDD on this parse tree.
- If there are no cycles, however, then there is always at least one topological sort.
- **S-attributed definition**
- An SDD is S-attributed if every attribute is synthesized.
- It is simple to evaluate the attributes by performing a post order traversal of the parse tree and evaluating the attributes at a node N when the traversal leaves N for the last time.
- S-attributed definitions can be implemented during bottom-up parsing, since a bottom-up parse corresponds to a post order traversal.
- Post order corresponds exactly to the order in which an LR parser reduces a production body to its head.
- Semantic action will be placed at right end of production .
- **L-attributed definitions**
- In dependency-graph edges between the attributes associated with a production body can go from left to right, but not from right to left(hence “L-attributed").
- Each attribute must be either
- 1. Synthesized, or
- 2. Inherited, but with the rules limited as follows.

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- Suppose there is a production $A \rightarrow X_1X_2 \ldots. Xn$, and an inherited attribute $X_i.a$ computed by a rule associated with this production.
- (a) Inherited attributes associated with the head A.
- (b) Either inherited or synthesized attributes associated with the occurrences of symbols $X_1, X_2, \ldots., X_{i-1}$ located to the left of X_i.
- (c) Inherited or synthesized attributes associated with this occurrence of X_i itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this X_i.
- Semantic action can be placed anywhere on body of production.
- Ex: S→A {print 1}B
- A → a{print 2}
- B → {print 3}b
- Every S-attributed SDD is L-attributed SDD.
- Every L-attributed SDD can be converted into S-attributed SDD.
- L-attributed SDD can be parsed top-down.

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- The rules for turning an L-attributed SDD into an SDT are as follows:
- 1. Embed the action that computes the inherited attributes for a nonterminal A immediately before that occurrence of A in the body of the production.
- If several inherited attributes for A depend on one another in an acyclic fashion, order the evaluation of attributes so that those needed first are computed first.
- 2. Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production.
- The following methods do translation by traversing a parse tree:
- 1. *Build the parse tree and annotate*.
- 2. *Build the parse tree, add actions, and execute the actions in pre order*.

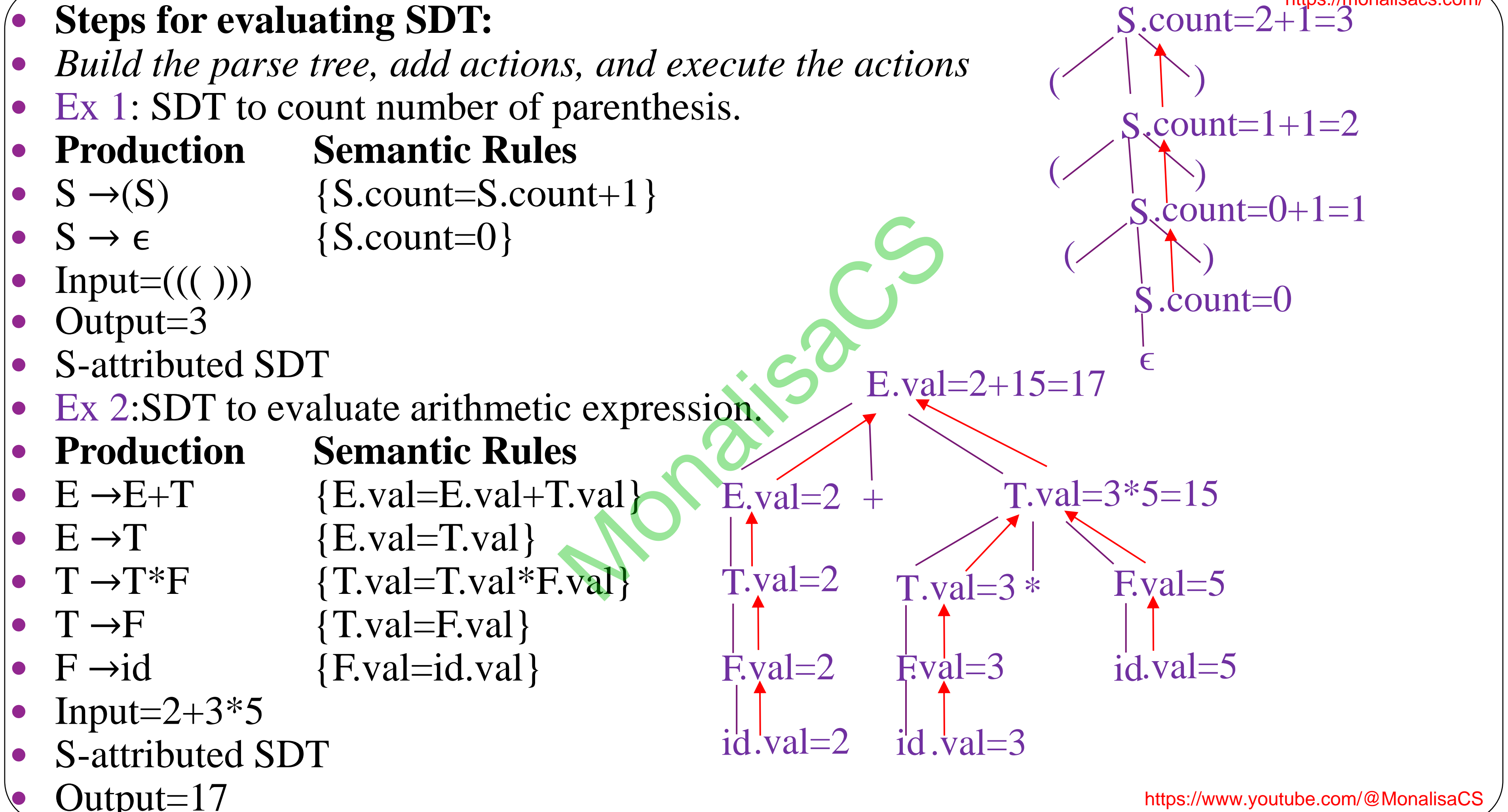
S-Attributed SDT	L-Attributed SDT
➢ Based on Synthesized attributes.	➢ Based on both synthesized & Inherited attributes with restriction to inherit from parent or left sibling only.
➢ Semantic rules always placed at rightmost position of RHS.	➢ Semantic rules can be placed anywhere on RHS.
➢ Attributes are evaluated bottom up , post order traversal.	➢ Attributes are evaluated top down ,pre order traversal or depth first traversal

MonalisaCS

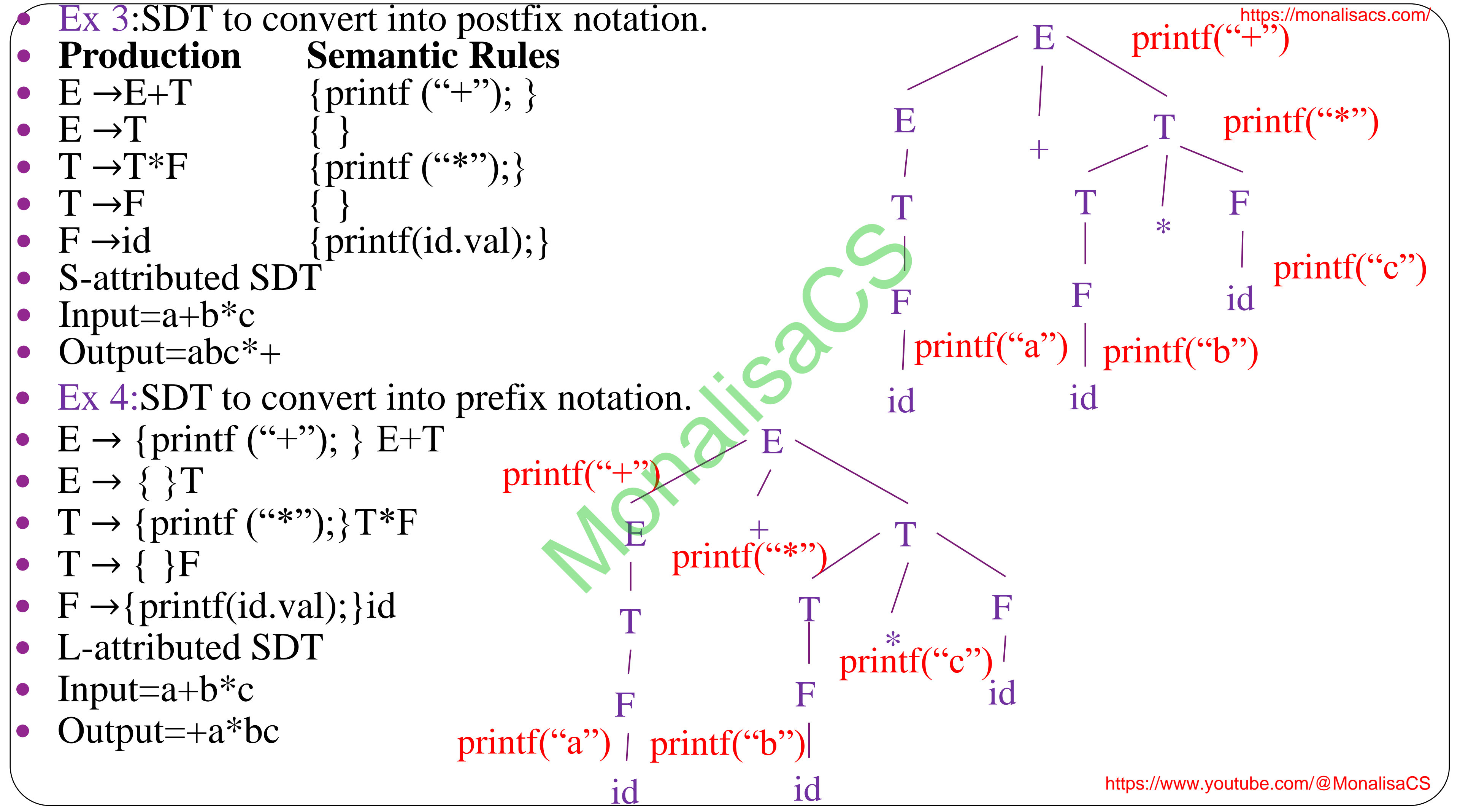
https://www.youtube.com/@MonalisaCS

- Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth-first order; that is, during a preorder traversal.
- 1. The underlying grammar is LR-parsable, and the SDD is S-attributed.
- 2. The underlying grammar is LL-parsable, and the SDD is L-attributed.
- SDT's with all actions at the right ends of the production bodies are called postfix SDT's.
- Postfix SDT's can be implemented during LR parsing by executing the actions when reductions occur.
- An action may be placed at any position within the body of a production.
- It is performed immediately after all symbols to its left are processed.
- If we have a production $B \rightarrow X\{a\}Y$, the action a is done after we have recognized X (if X is a terminal) or all the terminals derived from X (if X is a nonterminal).

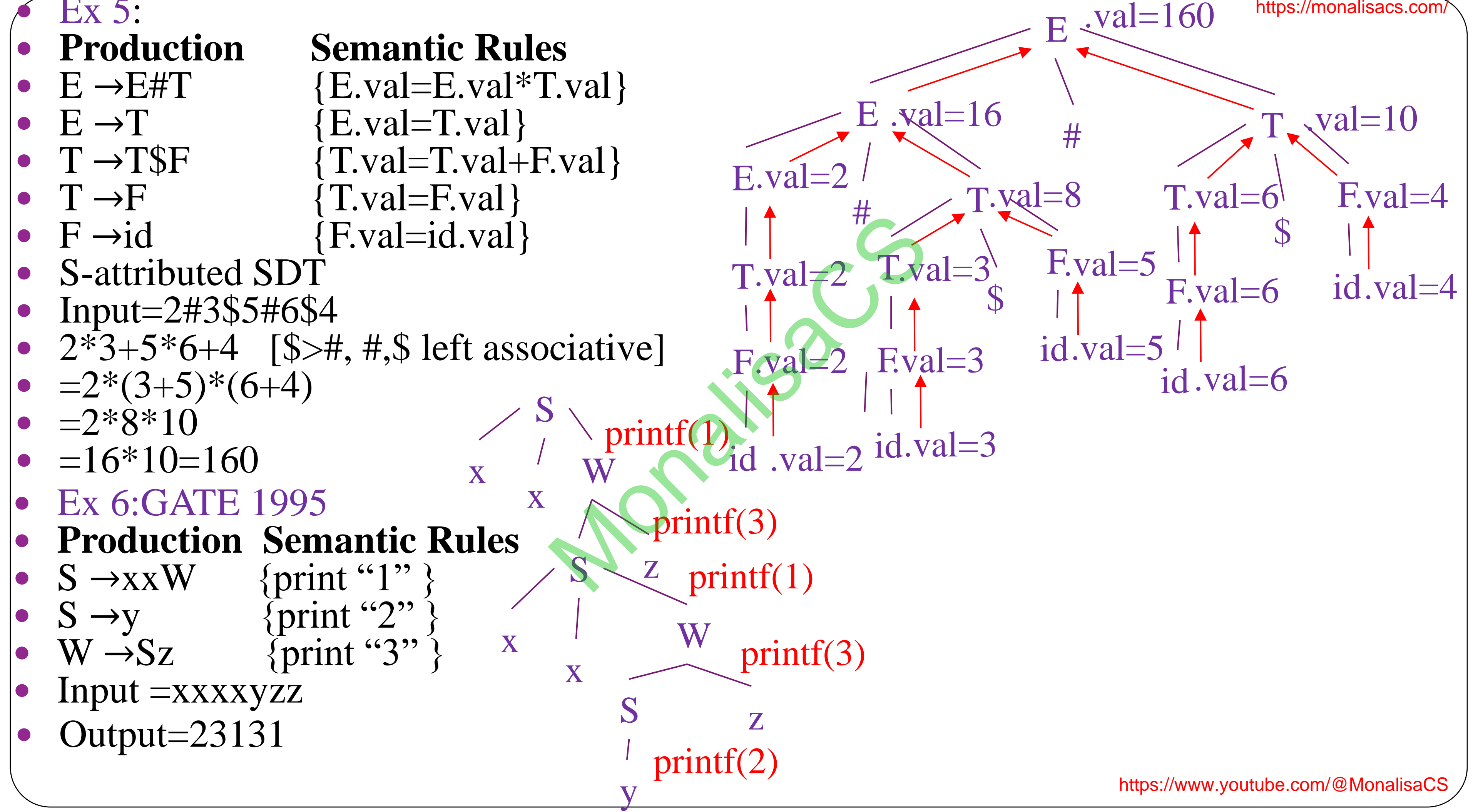
MonalisaCS



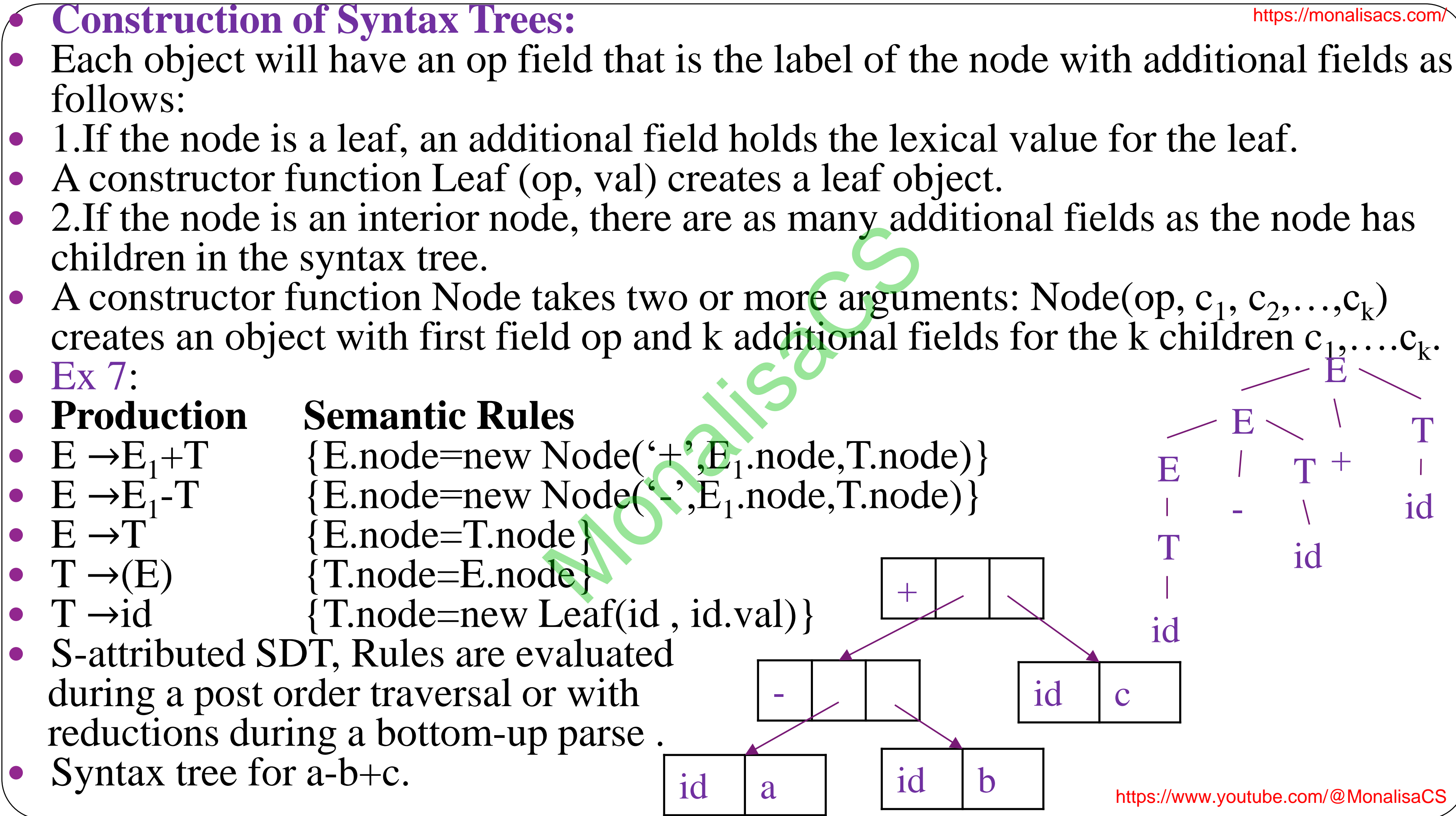
https://monalisacs.com/
Steps for evaluating SDT:
Build the parse tree, add actions, and execute the actions
Ex 1: SDT to count number of parenthesis.
Production
Semantic Rules
S →(S)
{S.count=S.count+1}
S → ϵ
{S.count=0}
Input=((( )))
Output=3
S-attributed SDT
Ex 2:SDT to evaluate arithmetic expression.
Production
Semantic Rules
E →E+T
{E.val=E.val+T.val}
E →T
{E.val=T.val}
T →T*F
{T.val=T.val*F.val}
T →F
{T.val=F.val}
F →id
{F.val=id.val}
Input=2+3*5
S-attributed SDT
Output=17
S.count=2+1=3
(
)
S.count=1+1=2
(
)
S.count=0+1=1
(
)
S.count=0
ϵ
E.val=2+15=17
E.val=2
+
T.val=3*5=15
T.val=2
T.val=3
*
F.val=5
F.val=2
F.val=3
id.val=5
id.val=2
id.val=3
MonalisaCS
https://www.youtube.com/@MonalisaCS



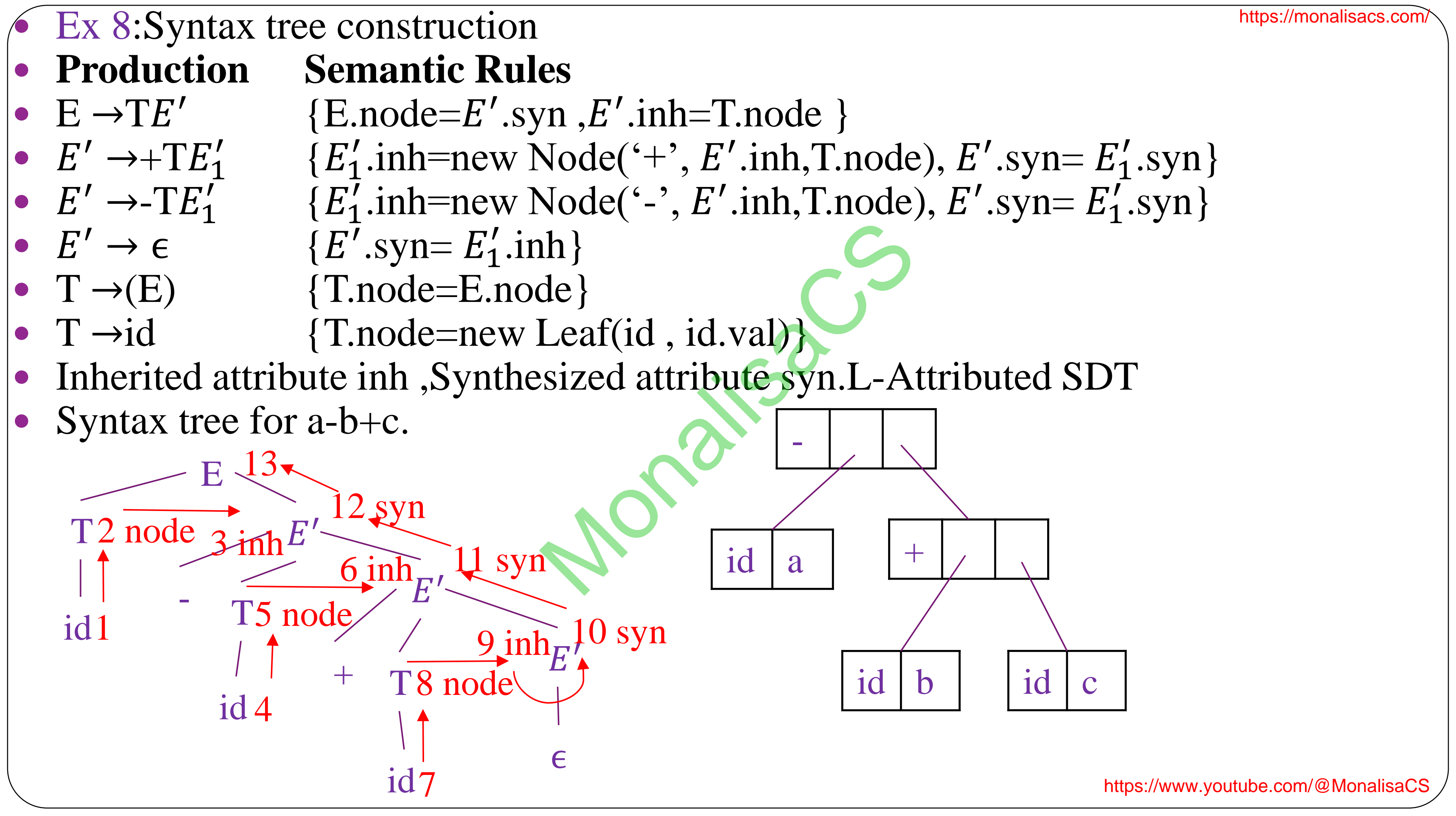
https://monalisacs.com/
Ex 3:SDT to convert into postfix notation.
Production Semantic Rules
E →E+T {printf ("+"); }
E →T { }
T →T*F {printf ("*");}
T →F { }
F →id {printf(id.val);}
S-attributed SDT
Input=a+b*c
Output=abc*+
Ex 4:SDT to convert into prefix notation.
E → {printf ("+"); } E+T
E → { }T
T → {printf ("*");}T*F
T → { }F
F →{printf(id.val);}id
L-attributed SDT
Input=a+b*c
Output=+a*bc
printf("+")
printf("*")
printf("c")
printf("a")
printf("b")
E
T
F
id
+
*
MonalisaCS
https://www.youtube.com/@MonalisaCS



Ex 5:
Production Semantic Rules
E →E#T {E.val=E.val*T.val}
E →T {E.val=T.val}
T →T$F {T.val=T.val+F.val}
T →F {T.val=F.val}
F →id {F.val=id.val}
S-attributed SDT
Input=2#3$5#6$4
2*3+5*6+4 [$>#, #,$ left associative]
=2*(3+5)*(6+4)
=2*8*10
=16*10=160
Ex 6:GATE 1995
Production Semantic Rules
S →xxW {print "1" }
S →y {print "2" }
W →Sz {print "3" }
Input =xxxxyzz
Output=23131
E .val=160
E .val=16
#
T .val=10
E.val=2
#
T.val=8
T.val=6
$
F.val=4
T.val=2
T.val=3
$
F.val=5
F.val=6
id.val=4
F.val=2
F.val=3
id.val=5
id.val=6
id .val=2
id.val=3
S
x
x
W
printf(1)
printf(3)
S
z
printf(1)
x
x
W
printf(3)
S
z
y
printf(2)
MonalisaCS
https://monalisacs.com/
https://www.youtube.com/@MonalisaCS



https://monalisacs.com/
Construction of Syntax Trees:
Each object will have an op field that is the label of the node with additional fields as follows:
1.If the node is a leaf, an additional field holds the lexical value for the leaf.
A constructor function Leaf (op, val) creates a leaf object.
2.If the node is an interior node, there are as many additional fields as the node has children in the syntax tree.
A constructor function Node takes two or more arguments: Node(op, c_1, c_2,…,c_k) creates an object with first field op and k additional fields for the k children c_1,….c_k.
Ex 7:
Production
Semantic Rules
E →E_1+T
{E.node=new Node('+',E_1.node,T.node)}
E →E_1-T
{E.node=new Node('-',E_1.node,T.node)}
E →T
{E.node=T.node}
T →(E)
{T.node=E.node}
T →id
{T.node=new Leaf(id , id.val)}
S-attributed SDT, Rules are evaluated during a post order traversal or with reductions during a bottom-up parse .
Syntax tree for a-b+c.
E
E
+
T
E
-
T
id
T
id
id
+
-
id
c
id
a
id
b
MonalisaCS
https://www.youtube.com/@MonalisaCS



https://monalisacs.com/
Ex 8:Syntax tree construction
Production Semantic Rules
E →TE' {E.node=E'.syn ,E'.inh=T.node }
E' →+TE'_1 {E'_1.inh=new Node('+', E'.inh,T.node), E'.syn= E'_1.syn}
E' →-TE'_1 {E'_1.inh=new Node('-', E'.inh,T.node), E'.syn= E'_1.syn}
E' → ϵ {E'.syn= E'_1.inh}
T →(E) {T.node=E.node}
T →id {T.node=new Leaf(id , id.val)}
Inherited attribute inh ,Synthesized attribute syn.L-Attributed SDT
Syntax tree for a-b+c.
E
13
12 syn
T
2 node
3 inh
E'
-
11 syn
6 inh
E'
id
1
T
5 node
+
9 inh
E'
10 syn
id
4
T
8 node
ϵ
id
7
-
id
a
+
id
b
id
c
MonalisaCS
https://www.youtube.com/@MonalisaCS

- Ex 9:Array declaration
- In C, int [2][3] =array(2, array(3, integer))
- can be read as,"array of 2 arrays of 3 integers."

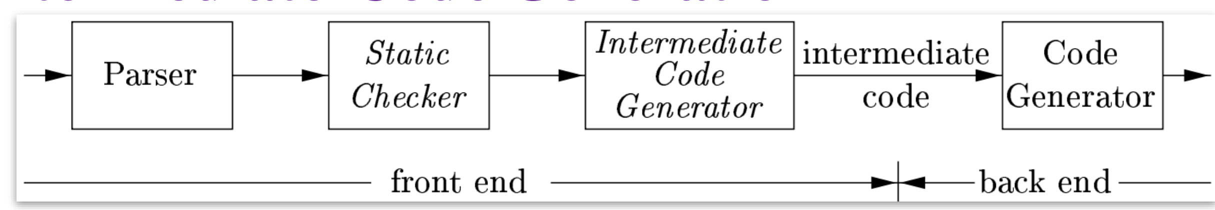
Production	Semantic Rules
T →BC	{T.t=C.t , C.b=B.t}
B →int	{B.t=integer}
B →float	{B.t=float}
C →[num]C_1	{C.t=array(num.val,C_1.t), C_1.b=C.b}
C → ϵ	{C.t= C.b}

- T generates either a basic type or an array type.
- T generates a basic type when T derives B C and C derives ϵ
- B generates one of the basic types int and float.
- C generates array components consisting of a sequence of integers, surrounded by brackets.
- B and T have a synthesized attribute 't' representing a type.
- C has two attributes: an inherited attribute 'b' and a synthesized attribute 't'.
- L-Attributed SDT ,The inherited b attributes pass a basic type down the tree, and the synthesized t attributes accumulate the result.

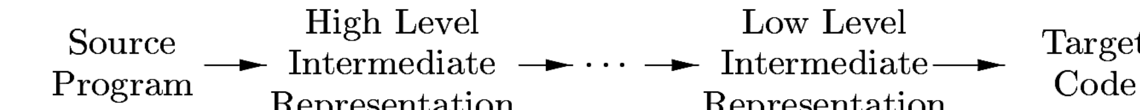
MonalisaCS

https://monalisacs.com/

Intermediate-Code Generation



- The front end translates a source program into an intermediate representation from which the back end generates target code.
- Static checking includes type checking, which ensures that operators are applied to compatible operands. It also includes any syntactic checks that remain after parsing.
- Benefits of using a machine-independent intermediate form are:
- 1. Retargeting is facilitated. That is, a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
- 2. A machine-independent code optimizer can be applied to the intermediate representation.
- In the process of translating a program in a given source language into code for a given target machine, a compiler may construct a sequence of intermediate representations



MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

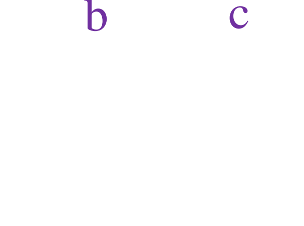
- High-level representations are close to the source language and low-level representations are close to the target machine.
- Syntax trees are high level; well suited to tasks like static type checking.
- A low-level representation is suitable for machine-dependent tasks like register allocation and instruction selection.
- Three-address code can range from high- to low-level, depending on the choice of operators.
- The term "three-address code' comes from instructions of the general form z = y op x with three addresses: two for the operands y and x and one for the result z.
- The choice or design of an intermediate representation varies from compiler to compiler.
- Intermediate code can be represented in following representation
- Non linear: Syntax Tree ,DAG ,Control Flow Graph
- Linear :Postfix code, Three-address code , SSA code
- **DAG:**
- Variants of Syntax Trees
- A directed acyclic graph (DAG) for an expression identifies the common subexpressions (subexpressions that occur more than once) of the expression.
- DAG's can be constructed by using the same techniques that construct syntax trees.

MonalisaCS

https://www.youtube.com/@MonalisaCS

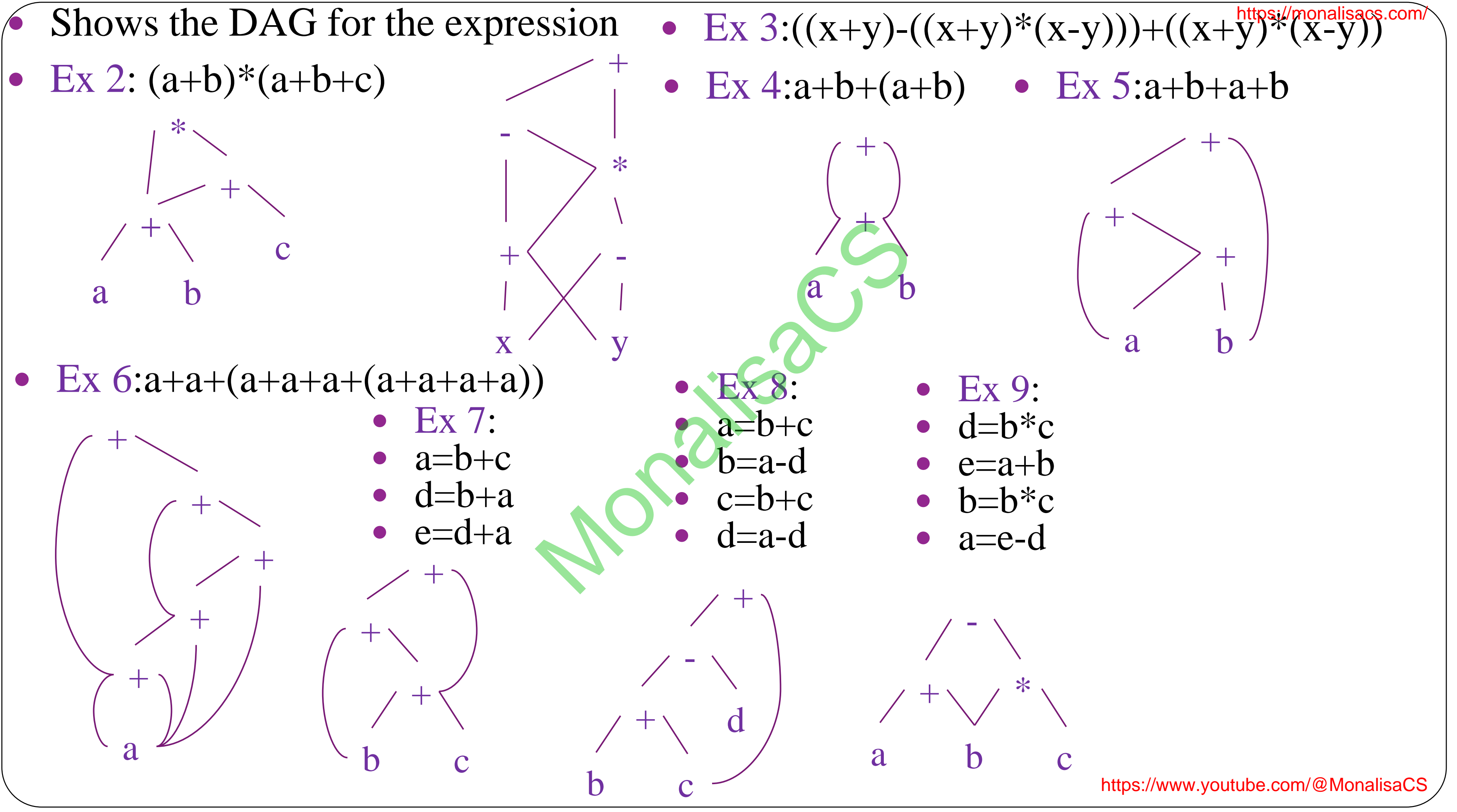
https://monalisacs.com/

- **Directed Acyclic Graphs for Expressions**
- A DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators.
- The difference is that a node N in a DAG has more than one parent if N represents a common subexpression.
- In a syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression.
- Thus, a DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.
- Ex 1: Shows the DAG for the expression a + a * (b - c) + (b - c) * d
- The leaf for a has two parents, because a appears twice in the expression.
- The two occurrences of the common subexpression b-c are represented by one node, That node has two parents , representing its two uses in the subexpressions a*(b-c) and (b-c)*d.
- The SDD can construct either syntax trees or DAG's.
- It will construct a DAG if , before creating a new node, these functions first check whether an identical node already exists.
- If a previously created identical node exists, the existing node is returned.



MonalisaCS

https://www.youtube.com/@MonalisaCS



Shows the DAG for the expression
Ex 2: (a+b)*(a+b+c)
Ex 3:((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))
Ex 4:a+b+(a+b)
Ex 5:a+b+a+b
Ex 6:a+a+(a+a+a+(a+a+a+a))
Ex 7:
a=b+c
d=b+a
e=d+a
Ex 8:
a=b+c
b=a-d
c=b+c
d=a-d
Ex 9:
d=b*c
e=a+b
b=b*c
a=e-d
MonalisaCS
https://monalisacs.com/
https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

The Value-Number Method for Constructing DAG's:

- The nodes of a syntax tree or DAG are stored in an array of records.
- Each row of the array represents one record, and therefore one node.
- In each record, the first field is an operation code, indicating the label of the node.
- leaves have one additional field, which holds the lexical value (either a symbol-table pointer or a constant), and interior nodes have two additional fields indicating the left and right children.
- Nodes of a DAG for i = i + 10 allocated in an array.
- In array, we refer to nodes by giving the integer index of the record for that node within the array.
- This integer called the value number for the node or for the expression represented by the node.
- The node labeled + has value number 3, and its left and right children have value numbers 1 and 2, respectively.
- value numbers help us construct expression DAG's efficiently

1	id	i	
2	num	10	
3	+	1	2
4	=	1	3

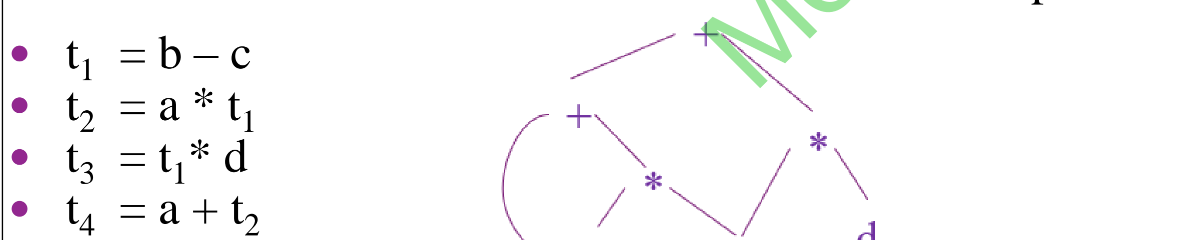
MonalisaCS

https://www.youtube.com/@MonalisaCS

Three-Address Code

- Three-address code is a sequence of statements of the form **x = y op z.**
- Where x, y and z are names, constants, or compiler-generated temporaries; op stands for any operator, such as arithmetic operator, or a logical operator on Boolean valued data.
- The expression x+ y*z might be translated into the sequence of three-address instructions
- $t_1 = y * z$
- $t_2 = x + t_1$
- **Advantages** : The complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.
- Three-address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph.
- Ex 1: A DAG and its three-address code for the expression a + a * (b - c) + (b - c) * d

- $t_1 = b - c$
- $t_2 = a * t_1$
- $t_3 = t_1 * d$
- $t_4 = a + t_2$
- $t_5 = t_4 + t_3$



MonalisaCS

- **Addresses and Instruction**
- Three-address code is built from two concepts: addresses and instructions.
- An address can be one of the following.
- **A name** . we allow source-program names to appear as addresses in three-address code.
- In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.
- **A constant**. a compiler must deal with many different types of constants and variables.
- **A compiler-generated temporary.** It is useful, in optimizing compilers, to create a distinct name each time a temporary is needed.
- **The common three-address statements are:**
- 1. Assignment statements of the form ***x = y op z***, where op is a binary arithmetic or logical operation.
- 2. Assignment instructions of the form ***x = op y***, where op is a unary operation.
- Essential unary operations include unary minus, logical negation,shift operators, and conversion operators
- 3. Copy instructions of the form ***x = y***,x is assigned the value of y.
- 4. The unconditional jump ***goto L***. The three-address statement with label L is the next to be executed.

MonalisaCS

https://monalisacs.com/

- 5. Conditional jumps such as ***if x goto L*** and ***if False x goto L***.
- These instructions execute the instruction with label L next if x is true and false, respectively.
- Otherwise, the following three-address instruction in sequence is executed next, as usual.
- 6. Conditional jumps such as ***if x relop y goto L***, which apply a relational operator (<, ==, >=, etc.) to x and y, and execute the instruction with label L next if x stands in relation relop to y.
- If not, the three-address instruction following is executed next, in sequence.
- 7. Indexed copy instructions of the form ***x = y[i]*** and ***x[i] = y***.
- The instruction x = y[i] sets x to the value in the location i memory units beyond location y. The instruction x[i] = y sets the contents of the location i units beyond x to the value of y.
- 8. Address and pointer assignments of the form ***x = &y, x =*y***, and ****x= y.***
- 9. Procedure calls and returns are implemented using the following instructions: ***param x*** for parameters; ***call p, n*** and ***y = call p, n*** for procedure and function calls, respectively; and ***return y***, where y, representing a returned value, is optional.
- param x_1
- param x_2
- ....
- param x_n
- call p,n
- generated as part of a call of the procedure $p(x_1, x_2, \ldots, x_n)$.

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- Two possible translations: symbolic label , position numbers
- 1.Uses a **symbolic label** L, attached to the instruction.
- 2.The **position numbers** for the instructions, starting arbitrarily at position.
- Ex 1 : Consider the statement *do i = i+1;*
- *while (a[i] < v);*
- In both translations, the last instruction is a conditional jump to the first instruction.
- The multiplication i * 8 is for an array of elements that each take 8 units of space.
- Symbolic labels
- L: t_1=i+1
- i= t_1
- t_2=i*8
- t_3=a [t_2]
- If t_3<v goto L
- Position numbers
- 100: t_1=i+1
- 101: i= t_1
- 102: t_2=i*8
- 103: t_3=a [t_2]
- 104: If t_3<v goto 100

- Ex 2:x+y*z^d
- t_1=z^d
- t_2=y* t_1
- t_3=x+ t_2

- Ex 3:
- if (x<y)
- z=x;
- else
- z=y;
- z=z*z;

- Three address code
- if x≥y goto L_0
- z=x
- goto L_1
- L_0:z=y
- L_1:z=z*z

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- **Three-Address Code for Array**
- 1D Array : int a[i]
- Loc a[i]= L_0 +i×w
- L_0 =Starting location or base
- w=width or size
- Three address code
- t_1=i*4
- t_2=a [t_1]
- 2D Array :int a[i][j]
- Let a[r][c]
- Row major order
- Loc a[i][j]= L_0 +[i ×c +j] ×w
- Three address code
- t_1=i*c
- t_2= t_1+j
- t_3= t_2 *4
- t_4=a [t_3]
- 3D Array :int a[i][j][k]
- Let a[r][c][b]
- Row major order
- Loc a[i][j][k]= L_0 +[i ×c ×b +j ×b+k] ×w
- Three address code
- t_1=i*c
- t_2= t_1 *b
- t_3= j *b
- t_4= t_2+ t_3
- t_5= t_4+k
- t_6= t_5*4
- t_7=a [t_6]

- t0 = i ∗ 1024
- t1 = j ∗ 32
- t2 = k ∗ 4
- t3 = t1 + t0
- t4 = t3 + t2
- t5 = X[t4]
- What is value of c,b?
- X[t3 + t2]
- =X[i ∗1024 +j∗32+k∗4]
- =X[i ∗ 256+j ∗ 8+ k]*4
- =X[i ∗32*8+j∗8+k]*4
- int X[ ][32][8]

MonalisaCS

https://www.youtube.com/@MonalisaCS

- A three-address statement is an abstract form of intermediate code.
- In a compiler, these statements can be implemented as records with fields for the operator and the operands.
- Three such representations are called "quadruples", "triples", and "indirect triples".
- **Quadruples :**
- A quadruple (or just "quad') has four fields, which we call ***op***, $\boldsymbol{arg_1}$, $\boldsymbol{arg_2}$, **and *result*.**
- The ***op*** field contains an internal code for the operator.
- The three-address instruction x = y + *z* is represented by placing + in ***op***, y in $\boldsymbol{arg_1}$, z in $\boldsymbol{arg_2}$, and x in ***result***.
- The following are some exceptions to this rule:

1. Instructions with unary operators like x = minus y or x = y do not use arg2. Note that for a copy statement like x = y, op is =, while for most other operations, the assignment operator is implied.
2. Operators like param use neither arg2 nor result.
3. Conditional and unconditional jumps put the target label in result.

MonalisaCS

https://monalisacs.com/

- **Ex 1**: Three-address code and quadruples for the assignment **a = b * - *c* +b * - c ;**
- For readability, we use actual identifiers like a, b, and c in the fields arg1,arg2, and result, instead of pointers to their symbol-table entries.
- t_1=minus c
- t_2=b*t_1
- t_3= t_2+t_2
- a= t_3
- a.Three-address code

	op	*arg₁*	*arg₂*	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	+	t_2	t_2	t_3
3	=	t_3		a

b.Quadruples

- **Triple** :
- In Quad the result field is used primarily for temporary names.
- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.
- If we do so, three-address statements can be represented by records with only three fields: op, arg1 and arg2.
- Using triples, we refer to the result of an operation x op y by its position, rather than by an explicit temporary name.
- The fields arg1 and arg2, for the arguments of op, are either pointers to the symbol table or pointers into the triple structure.

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

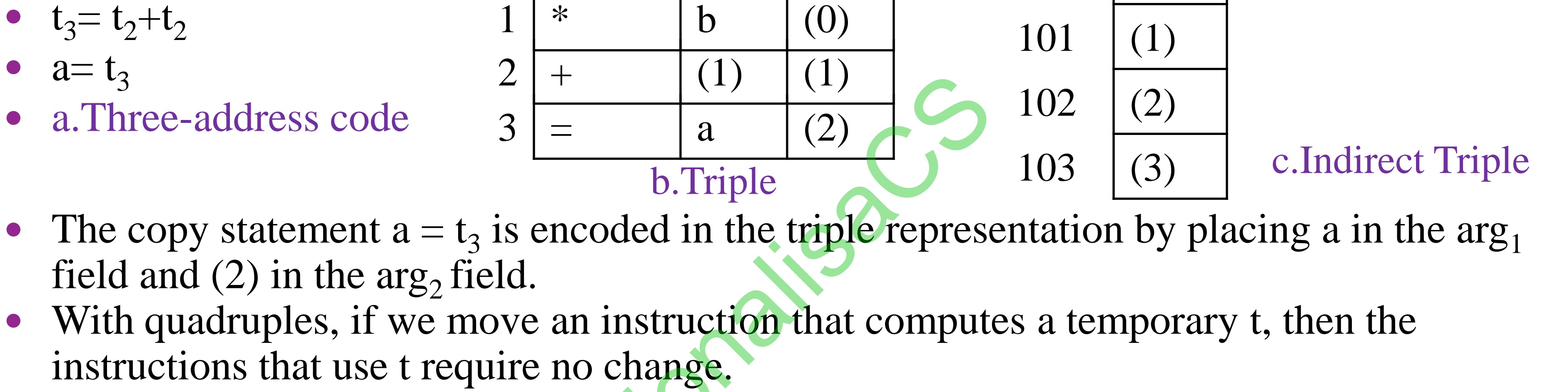
- **Ex 1**: Three-address code and quadruples for the assignment $\mathbf{a = b * - c + b * - c}$ **;**
- t_1=minus c
- $t_2=b*t_1$
- $t_3= t_2+t_2$
- $a= t_3$
- a.Three-address code

	op	*arg₁*	*arg₂*
0	minus	c	
1	*	b	(0)
2	+	(1)	(1)
3	=	a	(2)

b.Triple

	Instruction
100	(0)
101	(1)
102	(2)
103	(3)

c.Indirect Triple



- The copy statement $a = t_3$ is encoded in the triple representation by placing a in the arg_1 field and (2) in the arg_2 field.
- With quadruples, if we move an instruction that computes a temporary t, then the instructions that use t require no change.
- With triples, the result of an operation is referred to by its position , so moving an instruction may require us to change all references to that result.
- This problem does not occur with **indirect triples**.
- Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves.
- This implementation is called indirect triples.
- Use an array statement to list pointers to triples in the desired order.

https://www.youtube.com/@MonalisaCS

Static Single-Assignment Form

- Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations.
- All assignments in SSA are to variables with distinct names; hence the term *static single-assignment.*
- Every variable has single assignment , definition , meaning .
- Subscripts distinguish each definition of variables in the SSA representation.
- Ex 1: Intermediate program in three-address code and SSA

(a) Three-address code.	(b) Static single-assignment form
$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$

- The same variable may be defined in two different control-flow paths in a program.

MonalisaCS

Control Flow Graphs

- Control Flow Graph is a group of basic blocks.
- CFG has nodes and edges to define basic blocks and controls.
- The representation is constructed as follows :

➢ 1. Partition the intermediate code into basic blocks, which are maximal sequences of consecutive three-address instructions with the properties that

a) The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.

b) Control will leave the block without halting or branching, except possibly at the last instruction in the block.

➢ 2. The basic blocks become the nodes of a flow graph, whose edges indicate which blocks can follow which other blocks.

- **Basic Blocks**
- A basic block is a sequence of three address codes.
- Control enters only at begin of the sequence.
- Control leave only at the end of the sequence.

MonalisaCS

https://monalisacs.com/

- **Algorithm : Partitioning three-address instructions into basic blocks.**
- First, we determine those instructions in the intermediate code that are leaders , The rules for finding leaders are:

1. The first three-address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

- Then, for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program.
- **Flow Graphs:**
- Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph.
- The nodes of the flow graph are the basic blocks.
- There is an edge from block B to block C if and only if it is possible for the first instruction in block C to immediately follow the last instruction in block B .
- There is a conditional/unconditional jump from the end of B to the start of C.
- We say that B is a predecessor of C, and C is a successor of B.

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- We add two nodes, called the *entry* and *exit*, that do not correspond to executable intermediate instructions.
- There is an edge from the entry to the first executable node of the flow graph.
- There is an edge to the exit from last executed instruction of the program.
- Ex 1:
- 1) s=0
- 2) i=1
- 3) if i>n goto (8)
- 4) t=i*i
- 5) s=s+t
- 6) i=i+1
- 7) goto (3)
- 8) return s
- Rule 1=1
- Rule 2=3,8
- Rule 3=4,8
- Leader: 1,3,4,8
- B_1=1,2 ,B_2=3
- B_3=4,5,6,7 ,B_4=8

Entry

B_1: s=0, i=1

B_2: if i>n goto B_4

B_3: t=i*i, s=s+t, i=i+1, goto B_2

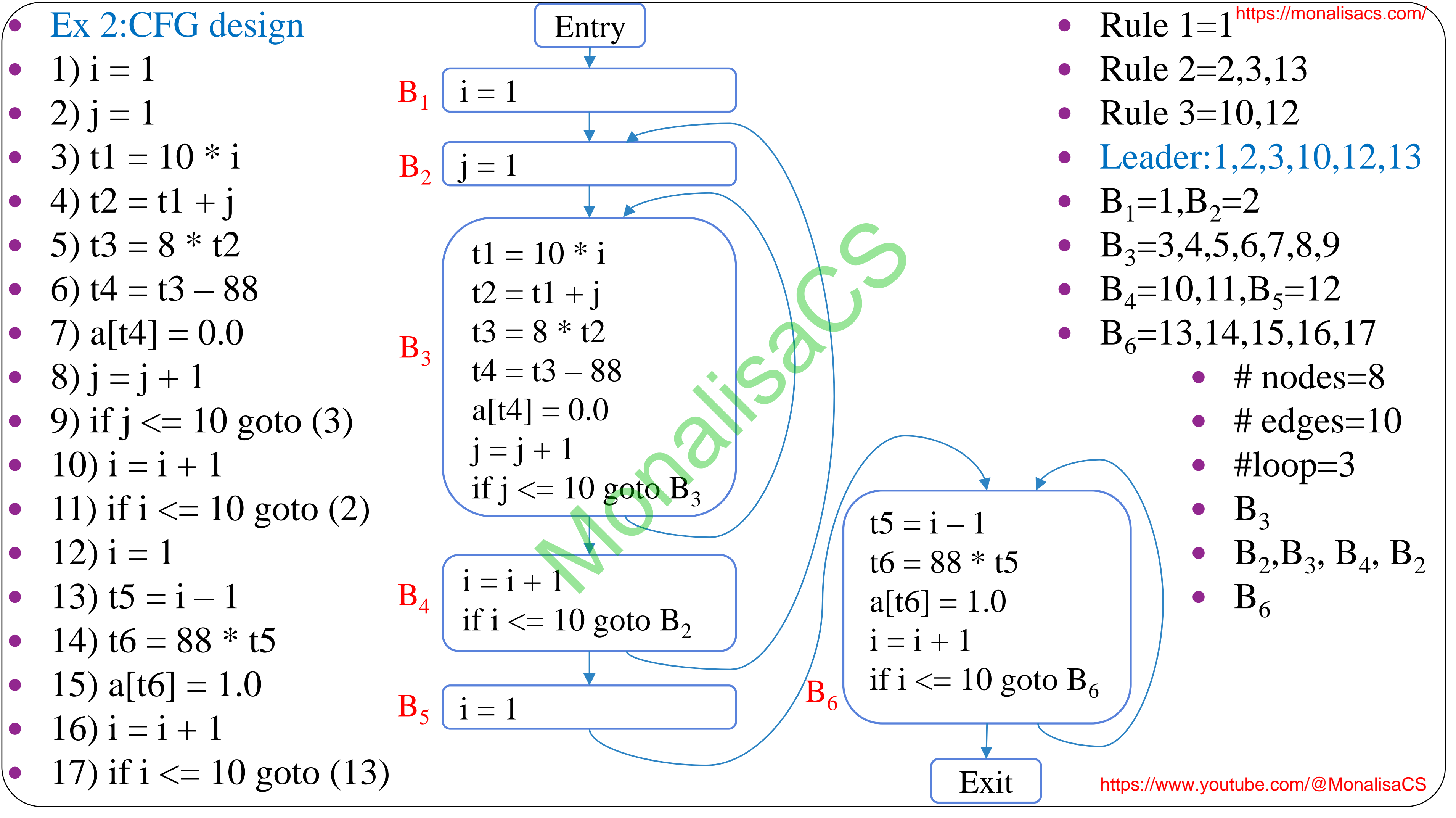
B_4: return s

Exit

- # nodes=6
- # edges= 7
- # loop=1
- $B_2 \rightarrow B_3 \rightarrow B_2$

MonalisaCS

https://www.youtube.com/@MonalisaCS


https://monalisacs.com/
Ex 2:CFG design
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 – 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i – 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
Entry
B_1
i = 1
B_2
j = 1
B_3
t1 = 10 * i
t2 = t1 + j
t3 = 8 * t2
t4 = t3 – 88
a[t4] = 0.0
j = j + 1
if j <= 10 goto B_3
B_4
i = i + 1
if i <= 10 goto B_2
B_5
i = 1
B_6
t5 = i – 1
t6 = 88 * t5
a[t6] = 1.0
i = i + 1
if i <= 10 goto B_6
Exit
MonalisaCS
Rule 1=1
Rule 2=2,3,13
Rule 3=10,12
Leader:1,2,3,10,12,13
B_1=1,B_2=2
B_3=3,4,5,6,7,8,9
B_4=10,11,B_5=12
B_6=13,14,15,16,17
nodes=8
edges=10
#loop=3
B_3
B_2,B_3, B_4, B_2
B_6
https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

Intermediate code representation

- Nonlinear: Syntax Tree ,DAG ,Control Flow Graph
- Linear :Postfix code, Three-address code , SSA code
- Ex: a=b*c + b*c
- Postfix code
- a b c * b c * + =
- Three-address code
- $t_1 = b*c$
- $t_2 = t_1 + t_1$
- $a = t_2$
- SSA code=Three-address code

CFG

Syntax Tree

DAG

MonalisaCS

https://www.youtube.com/@MonalisaCS

Next-Use Information

https://monalisacs.com/

- If the value of a variable that is currently in a register will never be referenced subsequently, then that register can be assigned to another variable.
- The *use* of a name in a three-address statement is defined as follows.
- Suppose i:x=a
- j:z=x+y
- control can flow from statement i to j along a path that has no intervening assignments to x, then we say statement j *uses* the value of x computed at statement i.
- We further say that x is *live* at statement i.
- Algorithm : Determining the liveness and next-use information for each statement in a basic block.
- METHOD: We start at the last statement in B and scan backwards to the beginning of B. At each statement i: x = y + z in B, we do the following:
- 1. Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y, and z.
- 2. In the symbol table, set x to "not live" and "no next use."
- 3. In the symbol table, set y and z to "live" and the next uses of y and z to i.

MonalisaCS

https://www.youtube.com/@MonalisaCS

Code Optimization

- Elimination of unnecessary instructions in object code, or the replacement of one sequence of instructions by a faster sequence of instructions that does the same thing is usually called "code improvement" or "code optimization".
- Types of optimization
- 1:Machine independent optimization
- 2:Machine dependent optimization
- **Machine independent optimization:**
- The process of optimizing intermediate code instruction is called as machine independent optimization
- Types of machine independent optimization
- 1.Local optimization :optimization within each basic block by itself
- 2.Global optimization : how information flows among the basic blocks of a program.
- **1.Local optimization**
- Local Common Subexpressions elimination
- Dead Code Elimination
- The Algebraic Optimization

MonalisaCS

https://monalisacs.com/

- We construct a DAG for a basic block as follows:
- 1. There is a node in the DAG for each of the initial values of the variables appearing in the basic block.
- 2. There is a node N associated with each statement s within the block . The children of N are those nodes corresponding to statements.
- 3. Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block.
- 4. Certain nodes are designated output nodes. These are the nodes whose variables are *live* on *exit* from the block; that is, their values may be used later, in another block of the flow graph.
- The DAG representation of a basic block help us several code improving transformations on the code represented by the block.
- a) We can eliminate *local common subexpressions*, that is, instructions that compute a value that has already been computed.
- b) We can eliminate *dead code*, that is, instructions that compute a value that is never used.
- c) We can reorder statements that do not depend on one another; such reordering may reduce the time a temporary value needs to be preserved in a register.
- d) We can apply algebraic laws to reorder operands of three-address instructions, and sometimes thereby simplify the computation.

MonalisaCS

https://www.youtube.com/@MonalisaCS

- **Finding Local Common Subexpressions**
- Common subexpressions can be detected by noticing, as a new node M is about to be added.
- Ex :A DAG for the block
- a = b + c
- b = a – d
- c = b + c
- d = a - d

- There are only three non leaf nodes in the DAG. So the basic block can be replaced by a block with only three statements.
- as b is not live or dead on exit from the block, then we do not need to compute that variable.
- a = b + c
- d = a – d
- c = d + c
- **Dead code elimination**
- The code having 'no next use' or 'not live' are dead code
- Ex: x=a * b
- y=a + c
- z=y + d

MonalisaCS

https://monalisacs.com/

- ‘x’ have ‘no next use’ or ‘not live’
- x is dead code
- After removal of dead code
- y=a+c
- z=y+d

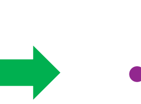
- **The Algebraic Optimization:**
- **1.Algebraic identities,** another important class of optimizations on basic blocks. arithmetic identities, such as $x + 0 = 0 + x = x$, $x - 0 = x$
- $x * 1 = 1 * x = x$, $x/1 = x$
- **2.Strength reduction** , replacing a more expensive operator by a cheaper one.
- *EXPENSIVE* *CHEAPER*
- x^2 = $x * x$
- $2 * x$ = $x + x$
- $x/2$ = $x * 0.5$
- **3.Constant folding** ,Here we evaluate constant expressions at compile time and replace the constant expressions by their values.
- $2 * 3.14 \Rightarrow 6.28$
- $x=2*3+y \Rightarrow x=6+y$;

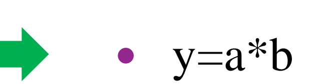
MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- **4.algebraic Simplification,** Apply algebraic transformations such as commutativity and associativity
- Ex 1: a = b + c
- t = c + d
- e = t + b
- a = b + c
- e = a + d
- If t is not needed outside this block, we can change this sequence to
- Ex 2:x * y- x * z ⇒ x * (y- z)
- Ex 3: y=x+a
- z=y-a
- w=z*b
- Backward substitution
- w=z*b =(y-a)*b= (x+a-a)*b=x*b
- w=x*b
- **5.Copy Propagation ,**coping constant or variable from one statement to other .
- Ex 1: x=2
- y=x*b
- y=2*b
- Ex 2: x=a
- y=x*b





- y=a*b

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

Representation of Array References

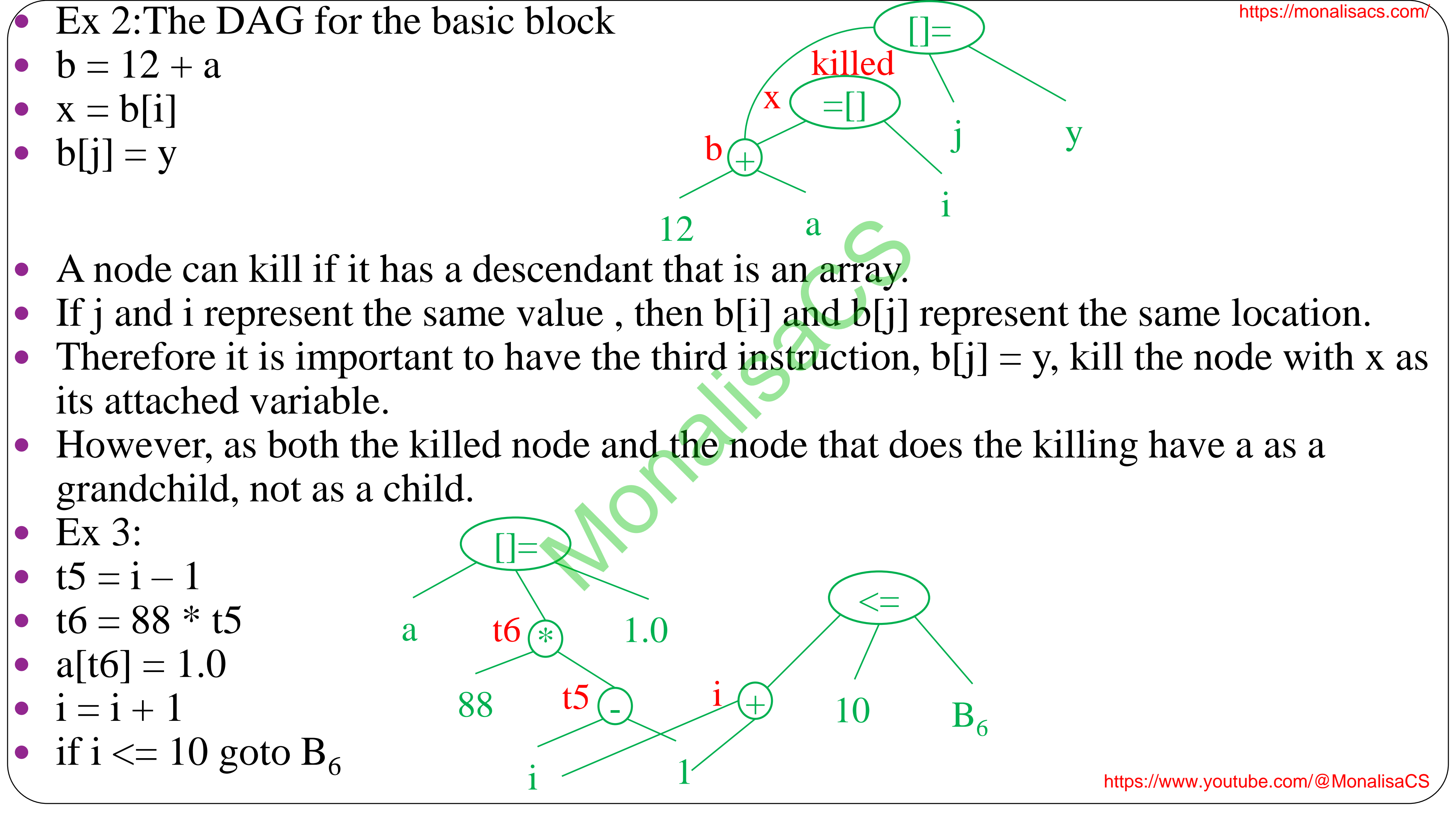
- The proper way to represent array in a DAG is as follows.
- 1. x = a[i], is represented by creating a node with operator =[ ] and two children a, and index i. Variable x becomes a label of this new node.
- 2.a[j] = y, is represented by a new node with operator [ ]= and three children representing a, j and y. There is no variable labeling this node.
- What is different is that the creation of this node *kills* all currently constructed nodes whose value depends on a.
- A node that has been killed cannot receive any more labels; that is, it cannot become a common subexpression.
- Ex 1:The DAG for the basic block
- x = a[i]
- a[j] = y
- z = a[i]

- a[j] = y
- z = a[i]

- The node N for x is created first, but when the node labeled [ ]= is created, N is killed.
- Thus, when the node for z is created, it cannot be identified with N, and a new node with the same operands a and i must be created instead.

MonalisaCS

https://www.youtube.com/@MonalisaCS


https://monalisacs.com/
Ex 2:The DAG for the basic block
b = 12 + a
x = b[i]
b[j] = y
[]=
killed
x
=[]
b
+
j
y
i
12
a
A node can kill if it has a descendant that is an array.
If j and i represent the same value , then b[i] and b[j] represent the same location.
Therefore it is important to have the third instruction, b[j] = y, kill the node with x as its attached variable.
However, as both the killed node and the node that does the killing have a as a grandchild, not as a child.
Ex 3:
t5 = i – 1
t6 = 88 * t5
a[t6] = 1.0
i = i + 1
if i <= 10 goto B_6
[]=
a
t6
*
1.0
<=
88
t5
-
i
+
10
B_6
i
1
MonalisaCS
https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- **2.Global Optimizations:** The optimization at program level is called as global optimization.
- Global optimizations are based on data- flow analyses , which are algorithms to gather information about a program.

➢ Types of global optimization

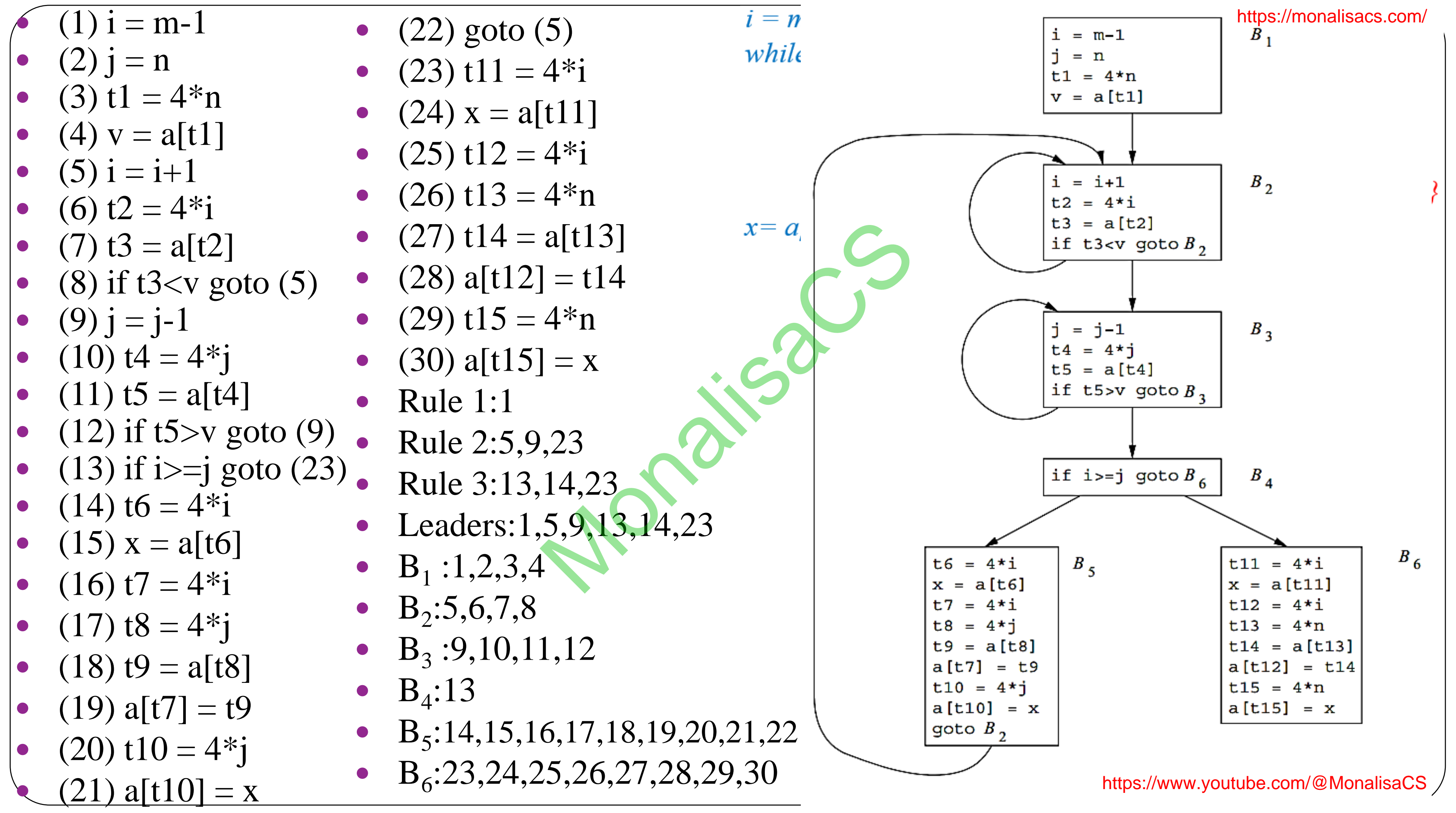
1. Global Common Subexpressions
2. Copy Propagation
3. Dead-Code Elimination
4. Code Motion
5. Induction Variables and Reduction in Strength

- We shall use a fragment of a quicksort to illustrate code-improving transformations.

```
i = m-1; j = n; v = a[n];
while (1)          {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */  }
x= a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
```

MonalisaCS

https://www.youtube.com/@MonalisaCS



(1) i = m-1
(2) j = n
(3) t1 = 4*n
(4) v = a[t1]
(5) i = i+1
(6) t2 = 4*i
(7) t3 = a[t2]
(8) if t3<v goto (5)
(9) j = j-1
(10) t4 = 4*j
(11) t5 = a[t4]
(12) if t5>v goto (9)
(13) if i>=j goto (23)
(14) t6 = 4*i
(15) x = a[t6]
(16) t7 = 4*i
(17) t8 = 4*j
(18) t9 = a[t8]
(19) a[t7] = t9
(20) t10 = 4*j
(21) a[t10] = x
(22) goto (5)
(23) t11 = 4*i
(24) x = a[t11]
(25) t12 = 4*i
(26) t13 = 4*n
(27) t14 = a[t13]
(28) a[t12] = t14
(29) t15 = 4*n
(30) a[t15] = x
Rule 1:1
Rule 2:5,9,23
Rule 3:13,14,23
Leaders:1,5,9,13,14,23
B1 :1,2,3,4
B2:5,6,7,8
B3 :9,10,11,12
B4:13
B5:14,15,16,17,18,19,20,21,22
B6:23,24,25,26,27,28,29,30
MonalisaCS
https://monalisacs.com/
i = m-1
j = n
t1 = 4*n
v = a[t1]
B1
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
B2
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
B3
if i>=j goto B6
B4
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
B5
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
B6
https://www.youtube.com/@MonalisaCS

- **Local Common Subexpression Elimination** B_5

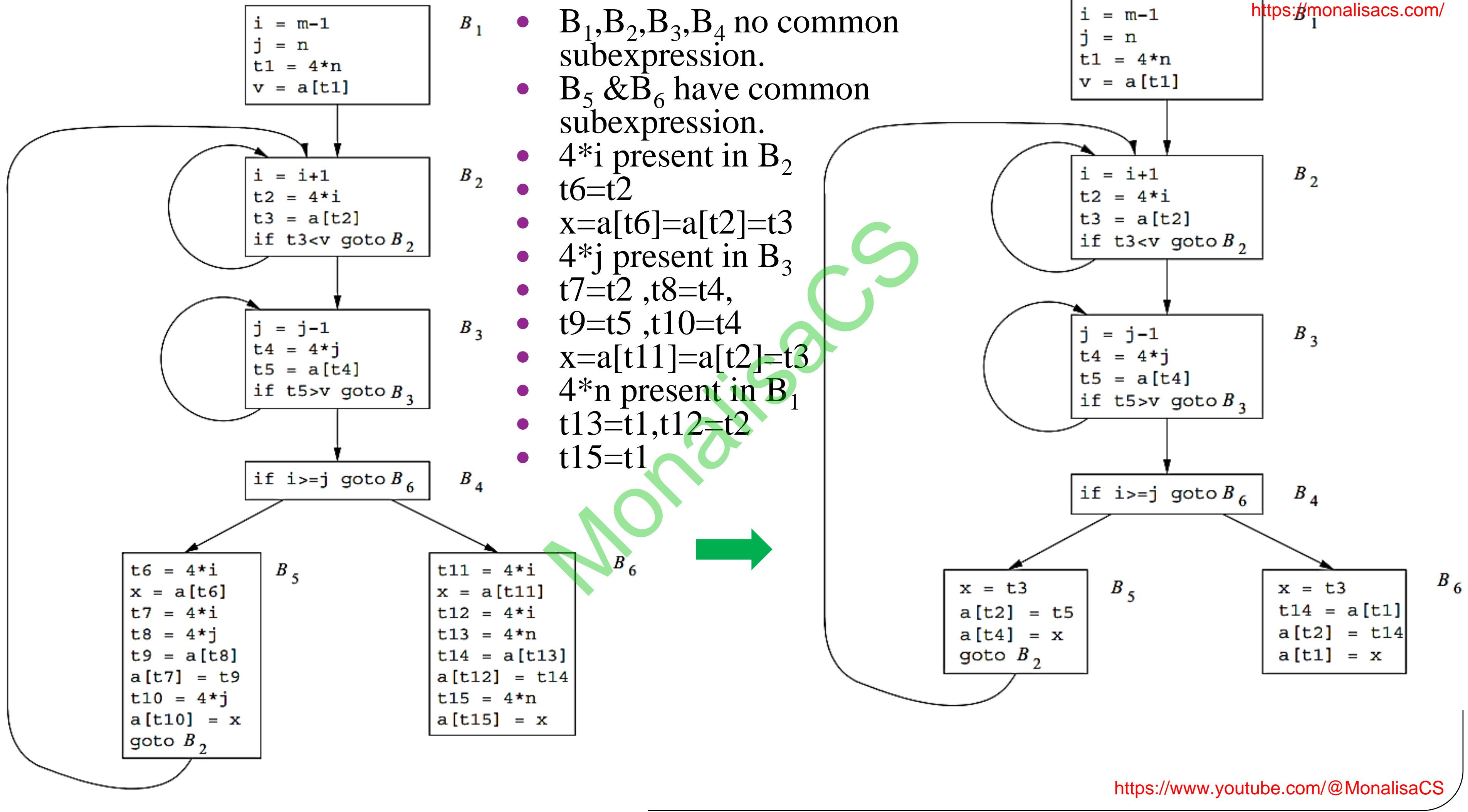
- t6 = 4*i
- x = a[t6]
- t7 = 4*i
- t8 = 4*j
- t9 = a[t8]
- a[t7] = t9
- t10 = 4*j
- a[t10] = x
- goto B_2

→

- t6 = 4*i
- x = a[t6]
- t8 = 4*j
- t9 = a[t8]
- a[t6] = t9
- a[t8] = x
- goto B_2

- **1.Global Common Subexpressions Elimination**
- An occurrence of an expression E is called a common subexpression if E was previously computed and the values of the variables in E have not changed since the previous computation.
- We avoid recomputing E if we can use its previously computed value;

MonalisaCS



MonalisaCS
https://monalisacs.com/
https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

2.Copy Propagation

- Assignments of the form u = v called copy statements, or copies for short.
- copy-propagation transformation is to use v for u.
- Block B_5 after copy propagation

x = t3 a[t2] = t5 a[t4] = x goto B2	→	x = t3 a[t2] = t5 a[t4] = t3 goto B_2	→	a[t2] = t5 a[t4] = t3 goto B_2

3.Dead-Code Elimination

- A variable is *live* at a point in a program if its value can be used subsequently; otherwise, it is *dead* or *no next use* at that point.
- A *dead (or useless*) code statements compute values that never get used.
- Ex : *if (debug) print ...*
- *debug = FALSE*
- If copy propagation replaces debug by FALSE, then the print statement is dead because it cannot be reached.
- One advantage of copy propagation is that it often turns the copy statement into dead code.
- For ex., copy propagation followed by dead-code elimination removes the assignment to x

MonalisaCS

https://www.youtube.com/@MonalisaCS

4.Code Motion

- Loops are a very important place for optimizations, especially the inner loops where programs tend to spend the bulk of their time.
- The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
- Modification that decreases the amount of code in a loop is *code motion*.
- Ex 1:

```
while (i <= limit-2)
```

→

```
t = limit-2
while (i <= t)
```

- Now, the computation of limit-2 is performed once, before we enter the loop.
- Ex 2:

```
int i=1,a=2;
while (i≤50)
{
        int j=i+a*3;
        printf(j);
        i++;
}
```

→

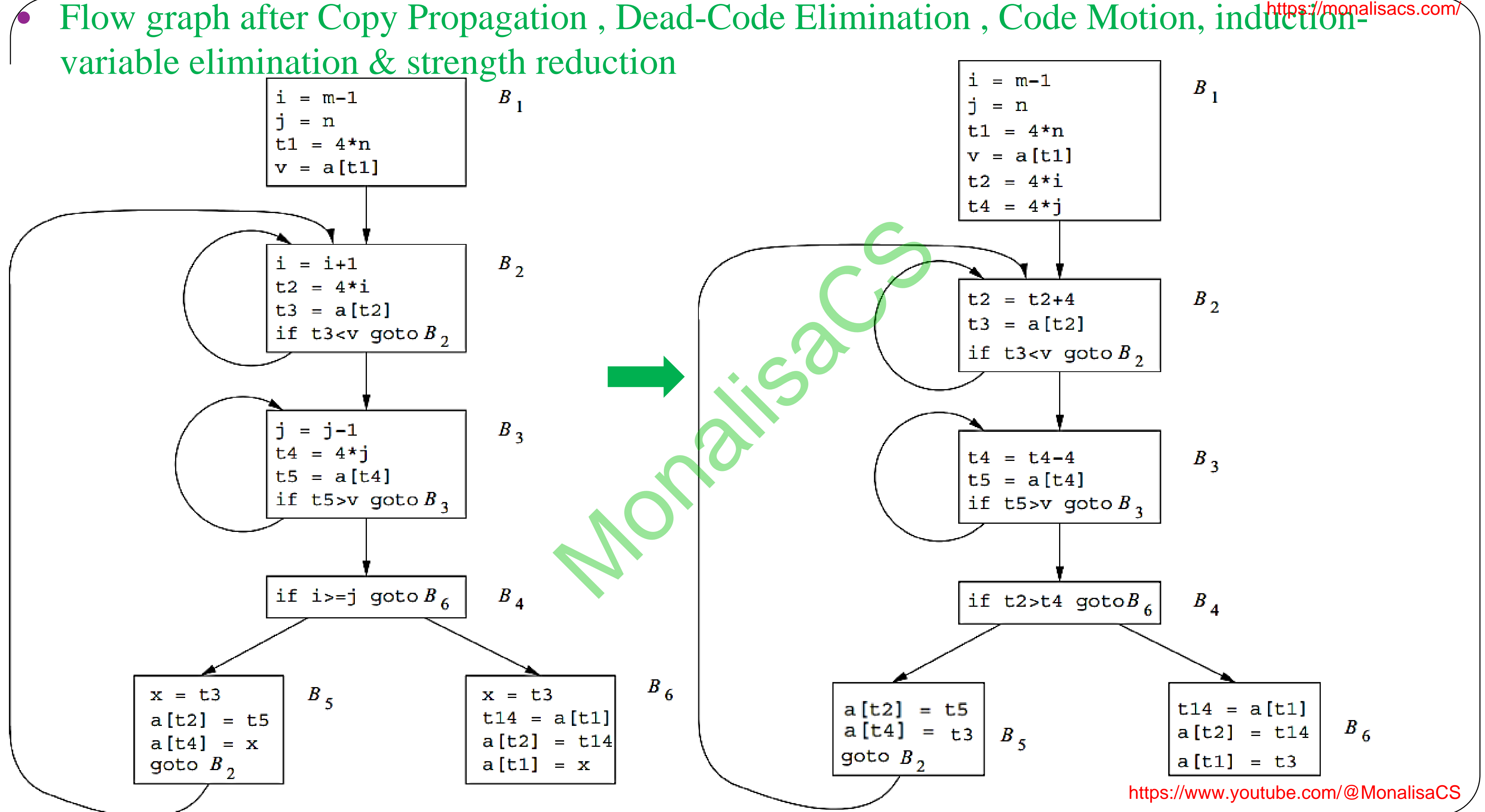
```
int i=1,a=2;
int b=a*3;
while (i≤50)
{
        int j=i+b;
        printf(j);
        i++;
}
```

MonalisaCS

5.Induction Variables and Reduction in Strength

- A variable x is said to be an *induction variable* if there is a positive or negative constant c such that each time x is assigned, its value increases by c.
- The transformation of replacing an expensive operation, such as multiplication, by a cheaper one , such as addition, is known as *strength reduction*.
- When processing loops, it is useful to work “inside-out”; that is, we shall start with the inner loops and proceed to progressively larger, surrounding loops.
- B_3: j=j-1
- t4=4*j →
  - t4=t4-4
- The values of j and t4 remain in lock step;
- Every time the value of j decreases by 1, the value of t4 decreases by 4, because 4* j is assigned to t4.
- Eliminate the induction variable j=j-1 .
- B_2: i=i+1
- t2=4*i →
  - t2=t2+4
- Every time the value of i increases by 1, the value of t2 increases by 4, because 4* i is assigned to t2.
- Move t2=4*i , t4=4*j statement to B_1

MonalisaCS



Flow graph after Copy Propagation , Dead-Code Elimination , Code Motion, induction-variable elimination & strength reduction
https://monalisacs.com/
i = m-1
j = n
t1 = 4*n
v = a[t1]
B1
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
B2
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
B3
if i>=j goto B6
B4
x = t3
a[t2] = t5
a[t4] = x
goto B2
B5
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x
B6
i = m-1
j = n
t1 = 4*n
v = a[t1]
t2 = 4*i
t4 = 4*j
B1
t2 = t2+4
t3 = a[t2]
if t3<v goto B2
B2
t4 = t4-4
t5 = a[t4]
if t5>v goto B3
B3
if t2>t4 goto B6
B4
a[t2] = t5
a[t4] = t3
goto B2
B5
t14 = a[t1]
a[t2] = t14
a[t1] = t3
B6
MonalisaCS
https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

Data-Flow Analysis

- All the optimizations depend on data- flow analysis.
- *Data-flow analysis* derive information about the flow of data along program execution paths .
- Flow graph tells us about the possible execution paths.
- We may define an *execution path* (or just *path*) from point p_1 to point p_n to be a sequence of points p_1, p_2,…p_n such that for each i = 1, 2 ,…. n-1,either

1. p_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that same statement, or
2. p_i is the end of some block and p_{i+1} is the beginning of a successor block.

- Example of data-flow analysis :
- The shortest execution path consists of the program points (1,2,3,4,9).
- The next shortest path executes one iteration of the loop and consists of the points (1, 2, 3, 4, 5, 6, 7, 8, 3, 4, 9).
- First time program point (5) is executed, the value of a is 1 due to definition d_1.

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- In subsequent iterations, d_3 the value of a is 243 .
- It is not possible to keep track of all the program states for all possible paths.
- We do not keep track of entire states ;rather, we abstract out certain details, keeping only the data we need for the purpose of the analysis.
- 1.We may summarize all the program states at point (5) by saying that the value of a is one of {1, 243}, and it may be defined by one of $\{d_1, d_3\}$.
- The definitions that may reach a program point along some path are known as *reaching definitions*.
- 2.If we are interested in implementing constant folding.
- If a use of the variable x is reached by only one definition , then we can simply replace x by the constant.
- On the other hand, several definitions of x may reach a single program point, then we cannot perform constant folding on x.
- We may simply describe certain variables as "*not a constant*," instead of collecting all their possible values or all their possible definitions.

```
(1)                               B1
     d1 : a = 1
(2)
(3)  if read()<=0 goto B4         B2
(4)
(5)                               B3
(6)  d2 : b = a
(7)  d3 : a = 243
(8)     goto B2
(9)                               B4
```

MonalisaCS

https://www.youtube.com/@MonalisaCS

The Data-Flow Analysis Schema

- We associate with every program point a *data-flow value* that represents an abstraction of the set of all possible program states.
- We denote the data-flow values before and after each statements by IN[s] and OUT[s].
- The data-flow problem is to find a solution to a set of constraints on the IN[s]'s and OUT[s]'s, for all statements s.
- There are two sets of constraints : Transfer functions and Control-Flow Constraints
- **Transfer Functions**
- The data-flow values before and after a statement are constrained by the semantics of the statement.
- If variable a has value v before executing statement b = a, then both a and b will have the value v after the statement.
- This relationship between the data-flow values before and after the assignment statement is known as a *transfer function*.
- Transfer functions information may propagate forward along execution paths, or it may flow backwards up the execution paths.
- In a *forward-flow problem*, the transfer function f_s of a statement s, takes the data-flow value before the statement and produces a new data-flow value after the statement. $OUT[s] = f_s(IN[s])$.

MonalisaCS

https://monalisacs.com/

- In a *backward-flow problem*, the transfer function f_s for statement s converts a data-flow value after the statement to a new data-flow value before the statement.
- $IN[s] = fs(OUT[s])$
- **Control-Flow Constraints**
- If a block B consists of statements $s_1, s_2, \ldots s_n$ in that order, then the control-flow value out of s_i is the same as the control-flow value into s_{i+1}.
- That is, $IN[s_{i+1}] = OUT[s_i]$, for all $i = 1,2,\ldots\ldots,n-1$.
- The set of dentitions reaching the leader statement of a basic block is the union of the definitions after the last statements of each of the predecessor blocks.
- **Data-Flow Schemas on Basic Blocks**
- If s_1 is the first statement of basic block B, then $IN[B] = IN[s_1]$,if s_n is the last statement of basic block B, then $OUT[B] = OUT[s_n]$.
- In a *forward-flow problem* $OUT[B] = f_B(IN[B])$. [transfer function of block is f_B]
- $IN[B] = \cup_{P \text{ a predecessor of } B} OUT[P]$
- In a *backward-flow problem* $IN[B] = f_B(OUT[B])$
- $OUT[B] = \cup_{S \text{ a successor of } B} IN[S]$

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

Reaching Definitions

- A definition *d* reaches a point *p* if there is a path from *d* to *p*, such that *d* is not "**killed**" along that path.
- We kill a definition of a variable *x* if there is any other definition of *x* anywhere along the path.
- If a definition *d* of some variable *x* reaches point *p*, then *d* is the place at which the value of *x* used at *p* was last defined.
- Example :
- All the definitions in block B_1 reach beginning of block B_2.
- The definition d_5: j = j-1 in block B_2 also reaches the beginning of block B_2, because no other definitions of j can be found in the loop leading back to B_2.
- This definition, however, kills the definition d_2: j = n, preventing it from reaching B3 or B4.
- The statement d_4: i = i+1 in B_2 does not reach the beginning of B_2 , because the variable i is always redefined by d_7: i = u3.
- The definition d_6: a = u2 also reaches the beginning of block B_2.

ENTRY
B_1: d_1 : i = m-1; d_2 : j = n; d_3 : a = u1
B_2: d_4 : i = i+1; d_5 : j = j-1
B_3: d_6 : a = u2
B_4: d_7 : i = u3
EXIT

MonalisaCS

https://www.youtube.com/@MonalisaCS

- **Transfer Equations for Reaching Definitions :**
- Consider a definition d: $u = v+w$
- This statement **"generates"** a definition d of variable u and **"kills"** all the other definitions in the program that define variable u.
- The transfer function of definition d can be expressed as $f_d(x) = gen_d \; U(x- kill_d)$
- Where $gen_d = \{d\}$, the set of definitions generated by the statement, and $kill_d$ is the set of all other definitions of u in the program.
- Suppose block B has n statements, with transfer functions $f_i(x) = gen_i \; U(x- kill_i)$ for $i = 1, 2, \ldots\ldots, n$. Then the transfer function for block B may be written as:
- $$f_B(x) = gen_B \; U(x- kill_B)$$
- Where $kill_B = kill_1 \; U \; kill_2 \; U \ldots\ldots \; U kill_n$
- and $gen_B=gen_n \; U(gen_{n-1}- kill_n)U(gen_{n-2}-kill_{n-1}-kill_n) \; U \ldots.U(gen_1-kill_2-kill_3-\ldots.- kill_n)$
- The ***gen*** set contains all the definitions inside the block that are "visible"immediately after the block .we refer to them as *downwards exposed.*
- A basic block's ***kill*** set is union of all the definitions killed by the individual statements .
- gen takes precedence over kill.
- In *gen-kill form,* kill set is applied before the gen set.

MonalisaCS

- Ex 1 : d_1: $a = 3$
  d_2: $a = 4$
- The gen set for the basic block is {d2} since d1 is not downwards exposed.
- $gen_B = gen_2 U(gen_1 - kill_2) = d_2\ U(d_1 - d_1) = d_2$
- The kill set contains both d1 and d2, since d1 kills d2 and vice versa.
- $kill_B = kill_1\ U\ kill_2 = d_2\ U d_1 = d_1, d_2$
- Since the subtraction of the kill set precedes the union operation with the gen set, the result of the transfer function for this block always includes definition d2.
- Ex 2 :Flow graph for illustrating reaching definitions
- $gen_{B_1} = \{d_1, d_2, d_3\}$, $kill_{B_1} = \{d_4, d_5, d_6, d_7\}$
- $gen_{B_2} = \{d_4, d_5\}$, $kill_{B_2} = \{d_1, d_2, d_7\}$
- $gen_{B_3} = \{d_6\}$, $kill_{B_3} = \{d_3\}$
- $gen_{B_2} = \{d_7\}$,$kill_{B_2} = \{d_1, d_4\}$

ENTRY

B_1
d_1 : i = m-1
d_2 : j = n
d_3 : a = u1

B_2
d_4 : i = i+1
d_5 : j = j-1

B_3
d_6 : a = u2

B_4
d_7 : i = u3

EXIT

https://monalisacs.com/
MonalisaCS
https://www.youtube.com/@MonalisaCS

- **Control-Flow Equations for Reaching Definitions :**
- $OUT[P] \subseteq IN[B]$ whenever there is a control-flow edge from P to B.
- $IN[B] = \cup_{P \text{ a predecessor of } B} OUT[P]$
- We refer to union as the meet operator for reaching definitions.
- **Iterative Algorithm for Reaching Definitions**
- Every control-flow graph has two empty basic blocks, an entry node and an exit node.
- Since no definitions reach the beginning of the graph, the transfer function for the entry block returns ϕ as an answer. That is, $OUT[entry] = \emptyset$.
- For all basic blocks B other than entry, $OUT[B] = gen_B \cup (IN[B] - kill_B)$
- $IN[B] = \cup_{P \text{ a predecessor of } B} OUT[P]$
- Iterative algorithm to compute reaching definitions

1) $OUT[entry] = \emptyset$;
2) *for (each basic block B other than entry)* $OUT[B] = \emptyset$;
3) *while (changes to any OUT occur)*
4) *for (each basic block B other than entry) {*
5) $IN[B] = \cup_{P \text{ a predecessor of } B} OUT[P]$
6) $OUT[B] = gen_B \cup (IN[B] - kill_B)$;

}

MonalisaCS

https://monalisacs.com/

Live-Variable Analysis

- In *live-variable analysis* we wish to know for variable x and point p whether the value of x at p could be used along some path in the flow graph starting at p.
- If so, we say x is *live* at p; otherwise, x is *dead* at p.
- An important use for live-variable information is **register allocation** for basic blocks.
- After a value is computed in a register, and used within a block, it is not necessary to store that value if it is dead at the end of the block.
- Also, if all registers are full and we need another register, we should favor using a register with a dead value, since that value does not have to be stored.
- Another use is **dead code elimination** .
- In data-flow equations IN[B] and OUT[B], represent the set of variables live at the points immediately before and after block B, respectively.
- These equations can also be derived by first defining the transfer functions of individual statements and composing them to create the transfer function of a basic block.

1. def_B as the set of variables defined (i.e., definitely assigned values) in B prior to any use of that variable in B, and
2. use_B as the set of variables whose values may be used in B prior to any definition of the variable.

MonalisaCS

https://www.youtube.com/@MonalisaCS

- Ex:block B_2 i=i+1
- j=j-1 $use_{B_2}=\{i , j\}$, $def_{B_2} =\{i , j\}$, as well.
- Any variable in use_B must be considered *live* on entrance to block B, while definitions of variables in def_B are dead at the beginning of B.
- Thus, the equations relating *def* and *use* to the IN and OUT are defined as follows:
- $IN[exit] = \emptyset$
- No variables are live on exit from the program
- And for all basic blocks B other than exit,
- $IN[B] = use_B \; U \; (OUT[B] - def_B)$
- A variable is live coming into a block if either it is used before redefinition in the block or it is live coming out of the block and is not redefined in the block.
- $OUT[B] = U_{S \text{ a successor of } B} \; IN[S]$
- A variable is live coming out of a block if and only if it is live coming into one of its successors.
- Relationship between equations for liveness and the reaching-definitions
- 1.Both sets of equations have union as the meet operator.
- In each data-flow schema we propagate information along paths, and we care only about whether any path with desired properties exist.

MonalisaCS

https://monalisacs.com/

- 2.However, information flow for liveness travels "backward", opposite to the direction of control flow,
- Because in this problem we want to make sure that the use of a variable x at a point p is transmitted to all points prior to p in an execution path, so that we may know at the prior point that x will have its value used.
- To solve a backward problem, instead of initializing OUT[entry], we initialize IN[exit].
- Sets IN and OUT have their roles interchanged, and *use* and *def* substitute for *gen* and *kill*, respectively.
- As for reaching definitions, the solution to the liveness equations is not necessarily unique, and we want the solution with the smallest sets of live variables.
- Algorithm : Live-variable analysis

1) *IN[exit] = ∅;*
2) *for (each basic block B other than exit) IN[B] = ∅;*
3) *while (changes to any IN occur)*
4) *for (each basic block B other than exit) {*
5) $OUT[B] = \bigcup_{S \text{ a successor of } B} IN[S]$
6) $IN[B] = use_B \cup (OUT[B] - def_B)$

}

MonalisaCS

https://www.youtube.com/@MonalisaCS

Difference Between Reaching Definition & Live variable analysis

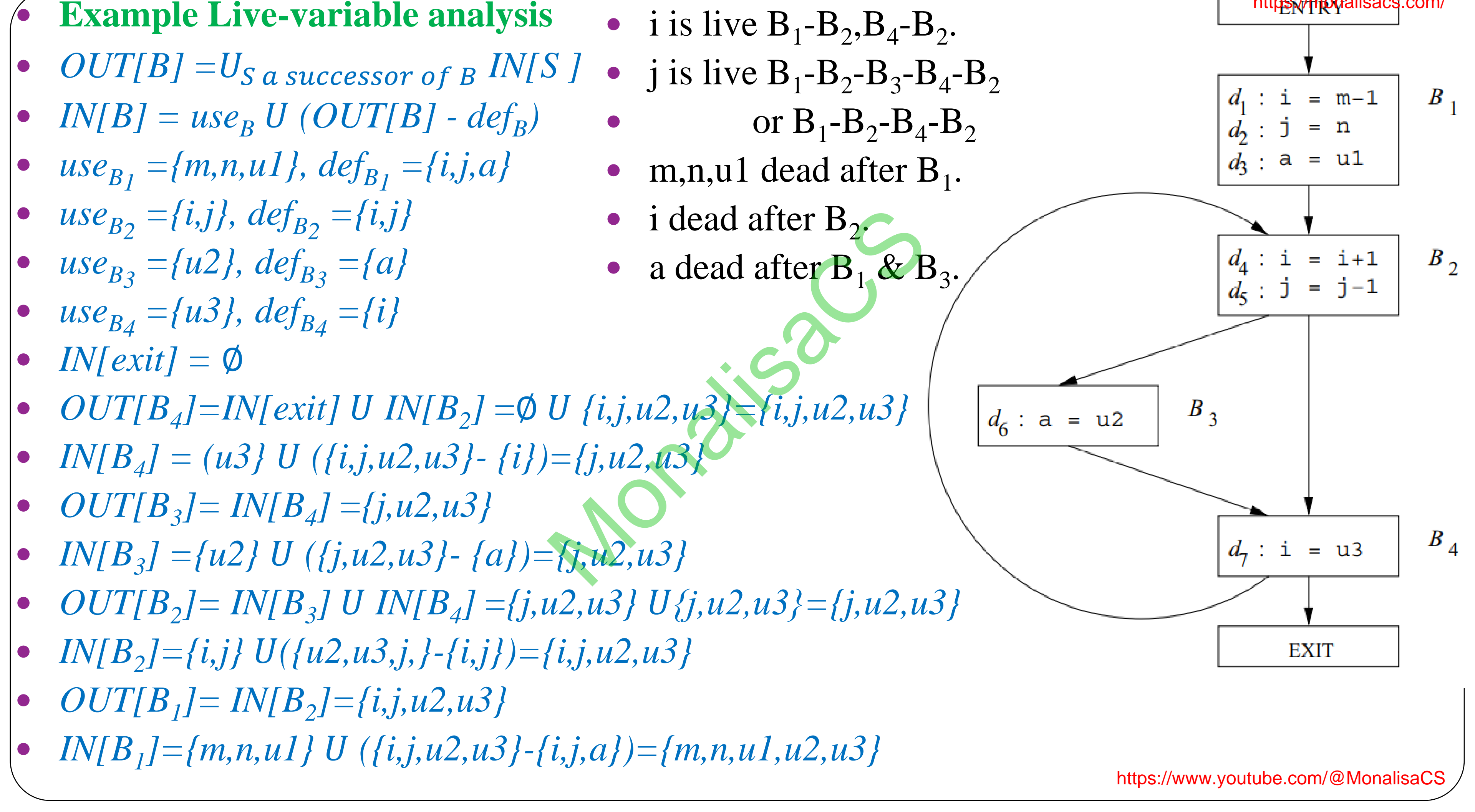
https://monalisacs.com/

	Reaching Definition	Liveness Analysis
Domain	Sets of definitions	Set of variable
Direction	Forwards	Backwards
Transfer Function($f_B(x)$)	$gen_B \; U (x - kill_B)$	$use_B \; U \; (x - def_B)$
Boundary	$OUT[entry] = \emptyset$	$IN[exit] = \emptyset$
Meet($\wedge$)	U	U
Equations	$OUT[B] = f_B\,(IN[B])$. $IN[B] = U_{P \text{ a predecessor of } B}\; OUT[P\,]$	$IN[B] = f_B\,(OUT[B])$ $OUT[B] = U_{S \text{ a successor of } B}\; IN[S\,]$
Initialize	$OUT[B] = \emptyset$	$IN[B] = \emptyset$

- Variable b is use in 4, so b is live on the (3 → 4) edge.
- 3 does not assign into b, b is also live on the (2→3) edge
- Statement 2 assigns b, so any value of b on the (1→2) and (5→2) edges are not needed, so b is dead along these edges
- b’s live range is (2→3→4)
- a’ is live from (1→2) and again (4→5→2) a is dead (2→3→4)

MonalisaCS

https://www.youtube.com/@MonalisaCS

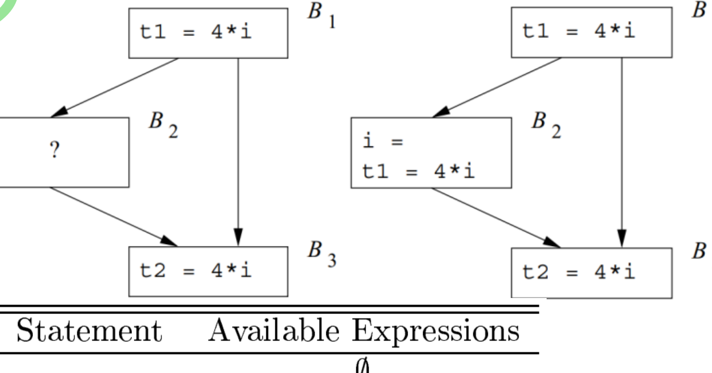


https://monalisacs.com/
MonalisaCS
https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

Available Expressions

- An expression x + y is *available* at a point p if every path from the entry node to p evaluates x + y, and after the last such evaluation prior to reaching p.
- A block generates expression x + y if it definitely evaluates x + y and does not subsequently define x or y.
- The primary use of available-expression information is for detecting **global common subexpressions.**
- The expression 4* i in block B_3 will be a common subexpression if 4*i is available at the entry point of B_3
- It will be available if i is not assigned a new value in block B_2, or 4* i is recomputed after i is assigned in B_2.
- Ex :After the first, b + c is available.After the second statement, a-d becomes available, but b + c is no longer available, as b has been redefined.
- The third statement does not make b + c available again, because the value of c is immediately changed.
- After the last statement, a-d is no longer available, as d has changed.
- Thus no expressions are generated, all expressions involving a, b, c, or d are killed.



Statement	Available Expressions
	$\emptyset$
a = b + c	
	$\{b+c\}$
b = a - d	
	$\{a-d\}$
c = b + c	
	$\{a-d\}$
d = a - d	
	$\emptyset$

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

Code Generation

- The input to the code generator is the intermediate representation, along with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the IR .
- The many choices for the IR include three-address representations such as quadruples, triples, indirect triples; virtual machine representations such as bytecodes and stack-machine code; linear representations such as post fix notation , SSA code; and graphical representations such as syntax trees and DAG's.

❖ The Target Program

- The most common target-machine architectures are RISC (reduced instruction set computer), CISC (complex instruction set computer), and stack based.
- A **RISC** machine has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture.
- A **CISC** machine has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.
- In a **stack-based machine**, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack.
- To achieve high performance the top of the stack is typically kept in registers.

MonalisaCS

https://www.youtube.com/@MonalisaCS

❖ Register Allocation:

- A key problem in code generation is deciding what values to hold in what registers.
- Registers are the fastest computational unit on the target machine , but we usually do not have enough of them to hold all values.
- Values not held in registers need to reside in memory.
- Instructions involving register operands are faster than those involving operands in memory, so efficient utilization of registers is important.
- The use of registers is often subdivided into two subproblems:

1. *Register allocation*, during which we select the set of variables that will reside in registers at each point in the program.
2. *Register assignment*, during which we pick the specific register that a variable will reside in.

❖ Operations on Target Machine

- Our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps.

❖ *Load operations*: The instruction *LD dst, addr* loads the value in location *addr* into location *dst* . This instruction denotes the assignment *dst* = *addr*.

- *LD r, x* which loads the value in location *x* into register *r*.
- *LD* r_1, r_2 is a *register-to-register copy* in which the contents of register r_2 are copied into register r_1.

MonalisaCS

https://monalisacs.com/

- *Store operations:* The instruction *ST x, r* stores the value in register *r* into the location *x*. This instruction denotes the assignment *x* = *r*.
- *Computation* operations of the form *OP dst, src_1, src_2*, where OP is a operator like *ADD* or *SUB*, and *dst*, src_1, and src_2 are locations, not necessarily distinct.
- For example, *SUB* r_1, r_2, r_3 computes $r_1 = r_2 - r_3$.
- *Unconditional jumps:* The instruction *BR L* causes control to branch to the machine instruction with label L. (*BR* stands for *branch*.)
- *Conditional jumps* of the form *Bcond r, L*, where *r* is a register, *L* is a label , and *cond* stands for any of the common tests on values in the register r.
- For example, *BLTZ r, L* causes a jump to label *L* if the value in register *r* is less than zero, and allows control to pass to the next machine instruction if not.

Addressing Modes:

- A location can be a **variable name** *x* referring to the memory location that is reserved for *x* (that is, the l-value of *x*).
- A location can also be an **indexed address** of the form *a(r)*, where *a* is a variable and *r* is a register.
- The memory location denoted by *a(r)* is computed by taking the l-value of *a* and adding to it the value in register *r*.
- For example, the instruction *LD R1, a(R2)* has the effect of setting R1 = contents(a + contents(R2)), where contents(x) denotes the contents of the register or memory location represented by x.

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- Indexed address is useful for accessing arrays, where a is the base address , and r holds the number of bytes past that address.

❖ A memory location can be an **integer indexed** by a register.

- For example, *LD R1, 100(R2)* has the effect of setting R1 = contents(100 +contents(R2)).
- That is loading into R1 the value in the memory location obtained by adding 100 to the contents of register R2.
- This is useful for following pointers.

❖ **Two indirect addressing modes**: *r means the memory location found in the location represented by the contents of register r and *100(r) means the memory location found in the location obtained by adding 100 to the contents of r.

- For example, LD R1, *100(R2) has the effect of setting R1 = contents(contents(100 + contents(R2))),
- That is, of loading into R1 the value in the memory location stored in the memory location obtained by adding 100 to the contents of register R2.

❖ **Immediate constant addressing mode:** The constant is prefixed by #.

- The instruction LD R1, #100 loads the integer 100 into register R1, and ADD R1, R1, #100 adds the integer 100 into register R1.
- Comments at the end of instructions are preceded by //.

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

Examples of machine instructions

Ex 1:x = y + z:

- LD R0, y // R0 = y
- ADD R0, R0, z // R0 = R0 + z
- ST x, R0 // x = R0
- Ex 2:x = y – z
- LD R1, y // R1 = y
- LD R2, z // R2 = z
- SUB R1, R1, R2 // R1 = R1 - R2
- ST x, R1 // x = R1
- Ex 3: a = b + c
- d = a + e would be translated into
- LD R0, b // R0 = b
- ADD R0, R0, c // R0 = R0 + c
- ST a, R0 // a = R0
- LD R0, a // R0 = a
- ADD R0, R0, e // R0 = R0 + e
- ST d, R0 // d = R0
- The fourth statement is redundant since it loads a value that has just been stored.
- The quality of the generated code is determined by its speed and size.

- If the target machine has an "increment" instruction (INC),then Three address statement a = a + 1 implemented efficiently by the single instruction INC a.
- Ex 4:a=a+1
- LD R0, a // R0 = a
- ADD R0, R0, #1 // R0 = R0 + 1
- ST a, R0 // a = R0

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- Suppose a is an array whose elements are 8-byte values, perhaps real numbers.
- Also assume elements of a are indexed starting at 0.
- Ex 5: b = a[i]
- LD R1, i // R1 = i
- MUL R1, R1, 8 // R1 = R1 * 8
- LD R2, a(R1) // R2 = contents(a + contents(R1))
- ST b, R2 // b = R2
- Ex 6: a[j] = c
- LD R1, c // R1 = c
- LD R2, j // R2 = j
- MUL R2, R2, 8 // R2 = R2 * 8
- ST a(R2), R1 // contents(a + contents(R2)) = R1
- Ex 7: x = *p
- LD R1, p // R1 = p
- LD R2, 0(R1) // R2 = contents(0 + contents(R1))
- ST x, R2 // x = R2
- Ex 8: *p = y
- LD R1, p // R1 = p
- LD R2, y // R2 = y
- ST 0(R1), R2 // contents(0 + contents(R1)) = R2

- Ex 9: if x < y goto L
- LD R1, x // R1 = x
- LD R2, y // R2 = y
- SUB R1, R1, R2 // R1 = R1 - R2
- BLTZ R1, L // if R1 < 0 jump to L

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

❖ Register and Address Descriptors

- In order to make decisions, we require a data structure that tells us what program variables currently have their value in a register, and which register or registers, if so.

❖ 1. For each available register, a *register descriptor* keeps track of the variable names whose current value is in that register.

- Initially, all register descriptors are empty. As the code generation progresses, each register will hold the value of zero or more names.

❖ 2. For each program variable, an *address descriptor* keeps track of the location or locations where the current value of that variable can be found.

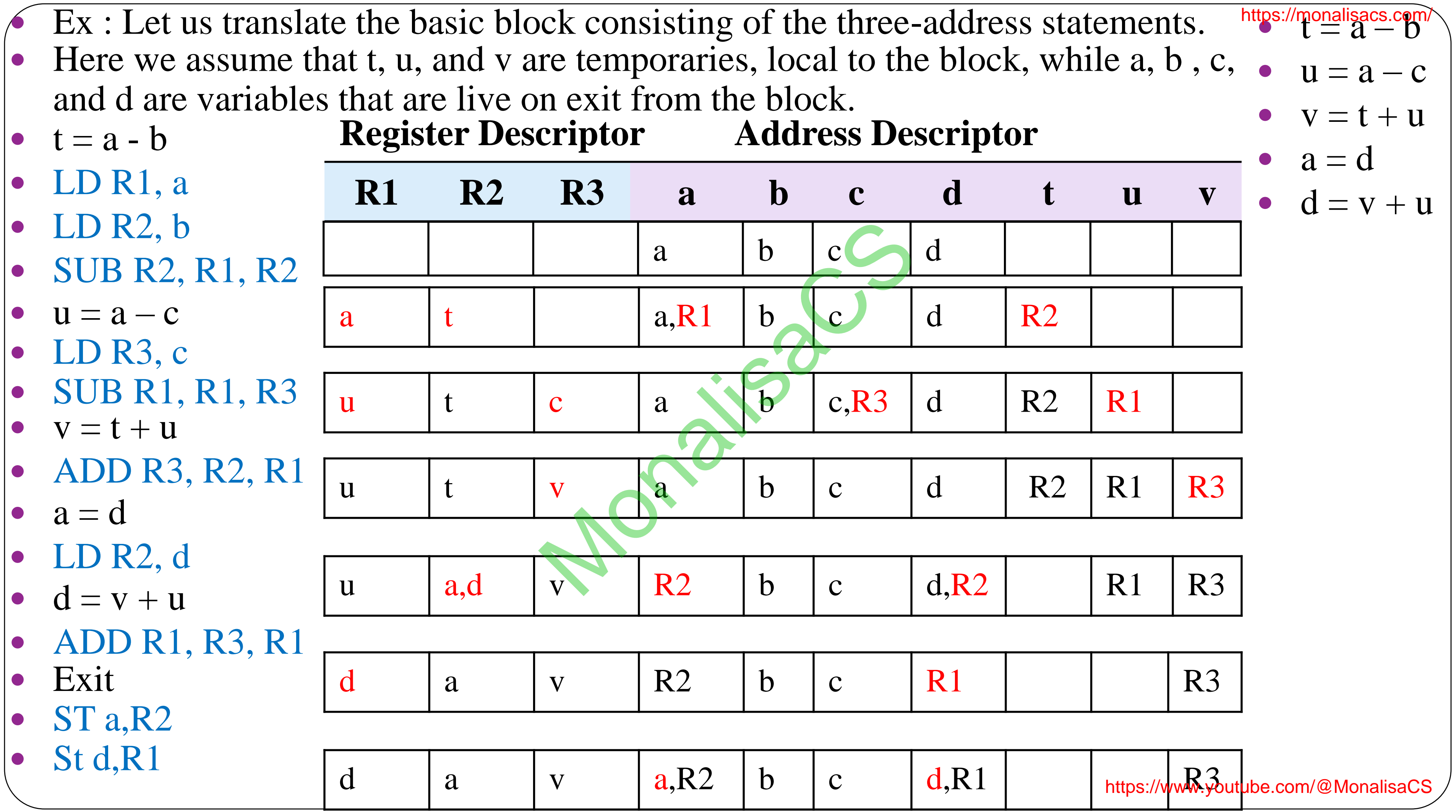
- The location might be a register, a memory address, a stack location, or some set of more than one of these.
- The information can be stored in the symbol-table entry for that variable name.
- Function *getReg(I)*, which selects registers for each memory location associated with the three-address instruction I .
- Function *getReg* has access to the register and address descriptors for all the variables of the basic block, and also have access to certain data-flow information such as the variables that are live on exit from the block.

https://www.youtube.com/@MonalisaCS
MonalisaCS

https://monalisacs.com/

- The rules for *register descriptor* and *address descriptor* are as follows:
- 1. For the instruction LD R, x
- (a) Change the register descriptor for register R so it holds only x.
- (b) Change the address descriptor for x by adding register R as an additional location.
- 2. For the instruction ST x, R, change the address descriptor for x to include its own memory location.
- 3. For an operation such as ADD R_x, R_y, R_z implementing a three-address instruction $x = y + z$
- (a) Change the register descriptor for R_x so that it holds only x.
- (b) Change the address descriptor for x so that its only location is R_x.
- Note that the memory location for x is not now in the address descriptor for x.
- (c) Remove R_x from the address descriptor of any variable other than x.
- 4. When we process a copy statement x = y, after generating the load for y into register Ry:
- (a) Add x to the register descriptor for R_y.
- (b) Change the address descriptor for x so that its only location is R_y.

MonalisaCS
https://www.youtube.com/@MonalisaCS



R1	R2	R3	a	b	c	d	t	u	v
			a	b	c	d			
a	t		a,R1	b	c	d	R2		
u	t	c	a	b	c,R3	d	R2	R1	
u	t	v	a	b	c	d	R2	R1	R3
u	a,d	v	R2	b	c	d,R2		R1	R3
d	a	v	R2	b	c	R1			R3
d	a	v	a,R2	b	c	d,R1			R3

https://monalisacs.com/
MonalisaCS
https://www.youtube.com/@MonalisaCS

- **Design of the Function *getReg***
- Let x = y + z .First, we must pick a register for y and a register for z.
- The rules for picking register Ry are as follows:

1. If y is currently in a register, pick a register already containing y as Ry.
2. If y is not in a register, but there is a empty register ,pick one such register as Ry.
3. When y is not in a register, and there is no empty register . We need to pick one of the allowable registers anyway, and we need to make it safe to reuse.

- Let R be a candidate register, and suppose v is one of the variables that the register descriptor for R says is in R. The possibilities are:

a) If the address descriptor for v says that v is somewhere besides R , then we are OK.
b) If v is x, the variable being computed by instruction I , and x is not also one of the other operands of instruction I ,then we are OK.
c) If v is not used later (that is, after the instruction I , there are no further uses of v, and if v is live on exit from the block, then v is recomputed within the block),then we are OK.
d) If we are not OK by one of the first three cases, then we need to generate the store instruction *ST v, R* to place a copy of v in its own memory location.

- This operation is called a **spill**.

MonalisaCS

Code Optimization

- **Machine dependent optimizations:**
- Machine dependent optimizations are based on register allocation and utilization of special machine-instruction sequences.
- **Peephole Optimization:**
- A simple but effective technique for locally improving the target code is peephole optimization.
- Which is done by examining a sliding window of target instructions (called the peephole) and replacing instruction sequences within the peephole by a shorter or faster sequence..
- Peephole optimization can also be applied directly after intermediate code generation
- The peephole is a small, sliding window on a program.
- Characteristic of peephole optimizations:

1. Redundant-instruction elimination
2. Flow-of-control optimizations
3. Algebraic simplifications
4. Use of machine idiom

❖ **Eliminating Redundant Loads and Stores**

- If we see the instruction in sequence Load ,Store or Store & Load
- LD R0, a
- ST a, R0
- Redundant loads and stores of this nature code can be eliminated.

MonalisaCS

Eliminating Unreachable Code

- Another opportunity for peephole optimization is the removal of unreachable instructions.
- An unlabeled instruction immediately following an unconditional jump may be removed.
- Ex: if debug == 1 goto L1
- goto L2
- L1: print debugging information
- L2:
- One obvious peephole optimization is to eliminate jumps over jumps.
- Thus , no matter what the value of debug, the code sequence above can be replaced by
- if debug != 1 goto L2
- print debugging information
- L2:
- If debug is set to 0 at the beginning of the program, constant propagation would transform this sequence into
- if 0 != 1 goto L2
- print debugging information
- L2:
- Now the argument of the first statement always evaluates to true, so the statement can be replaced by goto L2.
- Then all statements that print debugging information are unreachable and can be eliminated.

MonalisaCS

Flow-of-Control Optimizations

- Simple intermediate code produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps.
- These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations.
- We can replace the sequence

```
        goto L1                      goto L2
        ...              ⇒           .....
L1:  goto L2                 L1: goto L2
```

- If there are no jumps to L1, then it may be possible to eliminate the statement L1.

```
        if a < b goto L1                 if a < b goto L2
        .....                ⇒           .....
L1: goto L2                              L1: goto L2
```

- Suppose there is a unconditional jump to L1 and L1 contain a conditional goto.

```
goto L1                              if a < b goto L2
      . . .                          goto L3
L1: if a < b goto L2      ⇒          . . .
L3:                                  L3
```

MonalisaCS

- **Algebraic Simplification and Reduction in Strength:**
- These algebraic identities can also be used by a peephole optimizer to eliminate three-address statements such as
- x + 0 = 0 + x = x ,   x- 0 = x
- x * 1 = 1* x = x ,   x/1 = x in the peephole.
- Reduction in strength transformations can be applied in the peephole to replace expensive operations by equivalent cheaper ones on the target machine.
- *EXPENSIVE*   *CHEAPER*
- x^2 =   x * x
- 2 * x   =   x + x
- x/2   =   x* 0.5
- Fixed-point multiplication or division by a power of two is cheaper to implement as a shift.
- **Use of Machine Idioms**
- The target machine may have hardware instructions to implement specific operations efficiently.
- For ex, some machines have auto-increment and auto-decrement addressing modes.
- These add or subtract one from an operand before or after using its value.

MonalisaCS

Loop Optimization

- The optimization which perform over the loop called as loop optimization .
- Loop Jamming , Loop Unrolling ,Code Motion
- Loop Jamming : Combining two or more loops in a single loop having same index variable & number of iteration .
- **Initial Code:**

```
1. for(int i=0; i<10; i++)
        {printf("TOC");}
2. for(int i=0; i<10; i++)
        {printf("CD"); }
```

→

```
for(int i=0; i<10; i++)
        {printf("TOC");
        printf("CD");}
```

- Loop Unrolling : We basically remove or reduce iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.

```
while ( i<50)
  {printf("TOC");
   i++
}
```

→

```
while ( i<50)
  {printf("TOC");
   i++
   printf("TOC");
   i++
}
```

MonalisaCS

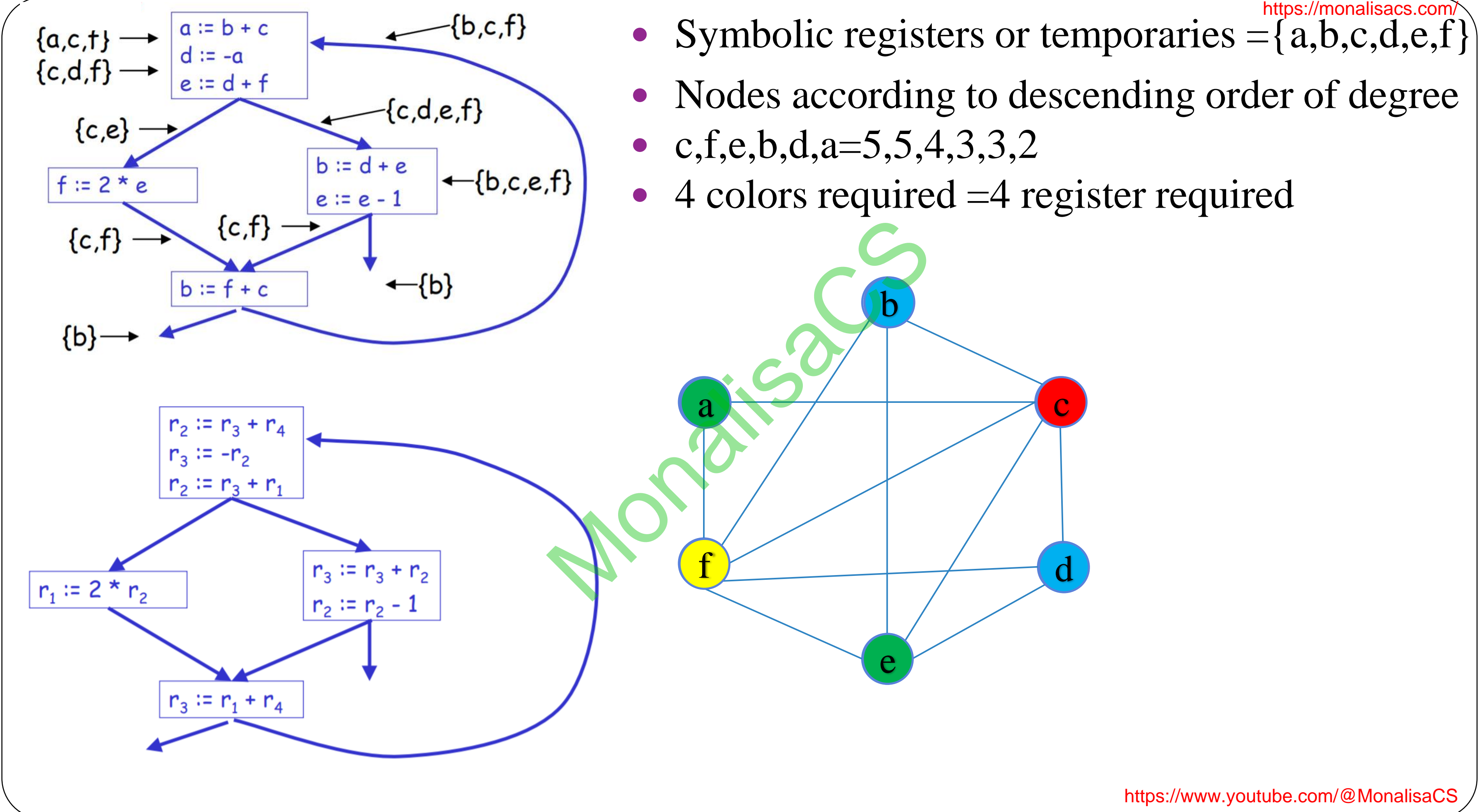
https://monalisacs.com/

Register Allocation by Graph Coloring Algorithm

- Register allocation is the process of determining which value should be placed into which registers and at what times during execution of program.
- Graph coloring is a simple , systematic technique for allocating registers and managing register spills.
- In the method, two passes are used.
- 1st pass: target-machine instructions are selected as though there are an infinite number of symbolic registers;
- In effect, names used in the intermediate code become names of registers and the three-address instructions become machine-language instructions.
- Once the instructions have been selected, a second pass assigns physical registers to symbolic ones.
- The goal is to find an assignment that minimizes the cost of spills.
- 2nd pass: For each procedure a *register-interference graph* is constructed in which the nodes are symbolic registers and an edge connects two nodes if one is live at a point where the other is defined.
- Graph Coloring :A coloring of a graph is an assignment of colors to nodes, such that nodes connected by an edge have different colors.
- A graph is k-colorable if it has a coloring with k colors
- colors = registers.
- We need to assign colors (registers) to graph nodes (temporaries)

MonalisaCS

https://www.youtube.com/@MonalisaCS

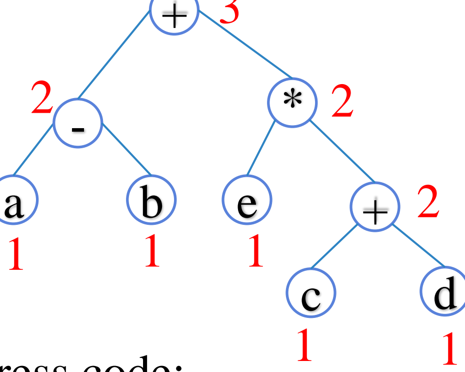

https://monalisacs.com/
{a,c,f}
{c,d,f}
a := b + c
d := -a
e := d + f
{b,c,f}
{c,e}
{c,d,e,f}
f := 2 * e
b := d + e
e := e - 1
{b,c,e,f}
{c,f}
{c,f}
{b}
b := f + c
{b}
$r_2 := r_3 + r_4$
$r_3 := -r_2$
$r_2 := r_3 + r_1$
$r_1 := 2 * r_2$
$r_3 := r_3 + r_2$
$r_2 := r_2 - 1$
$r_3 := r_1 + r_4$
Symbolic registers or temporaries ={a,b,c,d,e,f}
Nodes according to descending order of degree
c,f,e,b,d,a=5,5,4,3,3,2
4 colors required =4 register required
b
a
c
f
d
e
MonalisaCS
https://www.youtube.com/@MonalisaCS

Optimal Code Generation for Expression Tree

- We introduce a numbering scheme for the nodes of an expression tree (a syntax tree for an expression) that allows us to generate optimal code for an expression tree.
- **Ershov Numbers**
- We begin by assigning to each node of an expression tree a number that tells how many registers are needed to evaluate that node without storing any temporaries.
- These numbers are sometimes called Ershov numbers, after A. Ershov,who used a similar scheme for machines with a single arithmetic register.
- For our machine model, the rules are:

1. Label all leaves 1.
2. The label of an interior node with one child is the label of its child.
3. The label of an interior node with two children is

a) The larger of the labels of its children, if those labels are different.
b) One plus the label of its children if the labels are the same.



- Ex :Expression tree for expression (a- b) + e ∗ (c + d) or the three-address code:
- t1 = a - b
- t2 = c + d
- t3 = e * t2
- t4 = t1 + t3

- 3 registers are required.

MonalisaCS

Generating Code From Labeled Expression Trees

- METHOD: The steps below are applied, starting at the root of the tree.
- If label k, then only k registers will be used. There is a "base" $b \geq 1$ for the registers ,actual registers used are R_b, R_{b+1},….. R_{b+k-1}. The result always appears in R_{b+k-1}.

1. To generate machine code for an interior node with label k and two equal labels children:
   a) Recursively generate code for the right child, using base b + 1. The result of the right child appears in register R_{b+k-1}.
   b) Recursively generate code for the left child, using base b; the result appears in R_{b+k-2}.
   c) Generate the instruction OP R_{b+k-1}, R_{b+k-2}, R_{b+k-1}, OP is the operation for the interior node.
2. Suppose we have an interior node with label k and children with unequal labels. Then "big" child, has label k, and "little" child, has some label $m < k$:
   a) Recursively generate code for the big child, using base b; appears in register R_{b+k-1}.
   b) Recursively generate code for the little child, using base b; appears in register R_{b+m-1}.
   c) Generate the instruction OP R_{b+k-1} ,R_{b+m-1}, R_{b+k-1} or the instruction OP R_{b+k-1}, R_{b+k-1}, R_{b+m-1}, depending on whether the big child is the right or left child, respectively.
3. For a leaf representing operand x, if the base is b generate the instruction LD R_b, x.

MonalisaCS

https://monalisacs.com/

- Ex : Since the label of the root is 3, the result will appear in R3, and only R1, R2, and R3 will be used. The base for the root is b = 1.
- Since the root has children of equal labels, right child first, When we generate code for the right child of the root, we find the big child is the right child and the little child is the left child.
- We thus generate code for the right child first, with b = 2.
- LD R3, d
- LD R2, c
- ADD R3, R2, R3
- LD R2, e
- MUL R3, R2, R3
- For the left child of the root , base 1.
- LD R2, b
- LD R1, a
- SUB R2, R1, R2
- ADD R3, R2, R3

t1 = a - b
t2 = c + d
t3 = e * t2
t4 = t1 + t3

- interior node
  - =
  - Right child , base=b+1,R_{b+k-1}
  - Left child , base=b,R_{b+k-2}
  - OP R_{b+k-1}, R_{b+k-2}, R_{b+k-1}
  - ≠
  - Big child, base=b, R_{b+k-1}
  - little child, base b, R_{b+m-1}
  - OP R_{b+k-1} ,R_{b+m-1}, R_{b+k-1} or OP R_{b+k-1}, R_{b+k-1}, R_{b+m-1},
- leaf LD R_b, x.

MonalisaCS

https://www.youtube.com/@MonalisaCS

Run-Time Environments

- The compiler creates and manages a run-time environment in which it assumes its target programs are being executed.
- This environment deals with a variety of issues such as

1. The layout and allocation of storage locations for the objects named in the source program,
2. The mechanisms used by the target program to access variables,
3. The linkages between procedures,
4. The mechanisms for passing parameters,
5. The interfaces to the operating system, input/output devices, and other programs.

- Storage Organization:
- The executing target program runs in its own **logical address space** in which each program value has a location.
- The management and organization of this logical address space is shared between the compiler, operating system, and target machine.
- The operating system maps the logical addresses into physical addresses, which are usually spread throughout memory.
- The run-time storage comes in blocks of contiguous bytes, where a byte is the smallest unit of addressable memory.
- A byte is eight bits and four bytes form a machine word . Multibyte objects are stored in consecutive bytes and given the address of the first byte.

MonalisaCS

- The amount of storage needed for a name is determined from its type.
- A character array of length 10 needs only enough bytes to hold ten characters, a compiler may allocate 12 bytes to get the proper alignment , leaving 2 bytes unused.
- Space left unused due to alignment considerations is referred to as *padding*.
- When space is premium, a compiler may *pack* data so that no padding is left.
- *Subdivision of run-time memory into code and data areas→*

Code
Static
Heap
↓ Free Memory ↑
Stack

- ➢ **Code** :The size of the target code is fixed at compile time, so the compiler can place the executable target code in a statically determined area *Code*.
- ➢ **Static**: The size of some program data objects, such as global constants, and data generated by the compiler, such as information to support garbage collection, may be known at compile time, and these data objects can be placed in another statically determined area called *Static*.
- ➢ **Stack & Heap** :To maximize the utilization of space at run time, the other two areas, Stack and Heap, are at the opposite ends of the remainder of the address space.
- These areas are dynamic; their size can change as the program executes.
- These areas grow towards each other as needed.
- The *stack* is used to store data structures called activation records that get generated during procedure calls.
- The stack grows towards lower addresses, the heap towards higher.

https://monalisacs.com/
MonalisaCS
https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- Many programming languages allow the programmer to allocate and deallocate data under program control.
- For example, C has the functions malloc and free that can be used to obtain and give back arbitrary chunks of storage. The *heap* is used to manage this kind of long-lived data.
- Static Versus Dynamic Storage Allocation:
- Static and dynamic distinguish between compile time and run time, respectively.
- A storage-allocation decision is static, if it can be made by the compiler looking only at the text of the program, not at what the program does when it executes.
- A decision is dynamic if it can be decided only while the program is running.
- Many compilers use some combination of *Stack storage*, *Heap storage* for dynamic storage allocation.
- To support heap management, "garbage collection" enables the run-time system to detect useless data elements and reuse their storage.
- Automatic garbage collection is an essential feature of many modern languages.
- Stack Allocation of Space:
- Almost all languages use procedures, functions, or methods.
- Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.
- <u>**Activation Trees**</u>
- Stack allocation would not be feasible if procedure calls, or activations of procedures, did not nest in time.

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- We can represent the activations of procedures during the running of an entire program by a tree, called an *activation tree*.
- Each *node* corresponds to one activation, and the root is the activation of the "*main*" procedure that initiates execution of the program.
- At a node for an activation of procedure p,the children correspond to activations of the procedures called by this activation of p.
- We show these activations in the order that they are called, from left to right.
- If an activation of procedure p calls procedure q, then that activation of q must end before the activation of p can end.
- The use of a run-time stack is enabled by several useful relationships between the activation tree and the behavior of the program:
- 1. The sequence of procedure calls corresponds to a *preorder traversal* of the activation tree.
- 2. The sequence of returns corresponds to a *postorder traversal* of the activation tree.
- 3. Suppose that control lies within a particular activation of some procedure , corresponding to a node N of the activation tree.
- Then the activations that are currently open (live) are node N and its ancestors.
- The order in which these activations were called is the order in which they appear along the path to N , starting at the root, and they will return in the reverse of that order.

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

Ex 1: Construct Activation tree for Quick sort Program

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

- Possible activations for the program
- enter main()
- enter readArray()..leave readArray()
- enter quicksort(1,9)
- enter partition(1,9)..leave partition(1,9)
- enter quicksort(1,3)
- ...leave quicksort(1,3)
- enter quicksort(5,9)
- …leave quicksort(5,9)
- leave quicksort(1,9)
- leave main()

MonalisaCS

https://monalisacs.com/

- Ex 2:Activation tree for f(5)

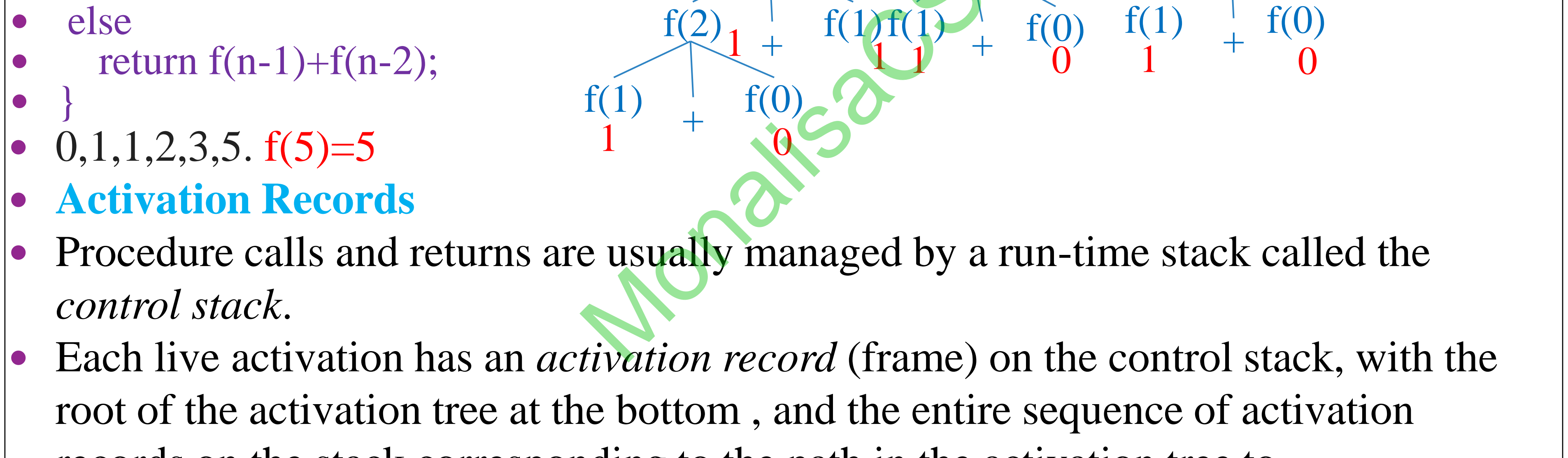
```
int f(int n)
{
 if (n==0)
     return 0;
 if (n==1)
     return 1;
 else
    return f(n-1)+f(n-2);
}
```

```
f(5) 5
├── f(4) 3
│   ├── f(3) 2
│   │   ├── f(2) 1
│   │   │   ├── f(1) 1
│   │   │   ├── +
│   │   │   └── f(0) 0
│   │   ├── +
│   │   └── f(1) 1
│   ├── +
│   └── f(2) 1
│       ├── f(1) 1
│       ├── +
│       └── f(0) 0
├── +
└── f(3) 2
    ├── f(2) 1
    │   ├── f(1) 1
    │   ├── +
    │   └── f(0) 0
    ├── +
    └── f(1) 1
```

- 0,1,1,2,3,5. f(5)=5

Activation Records

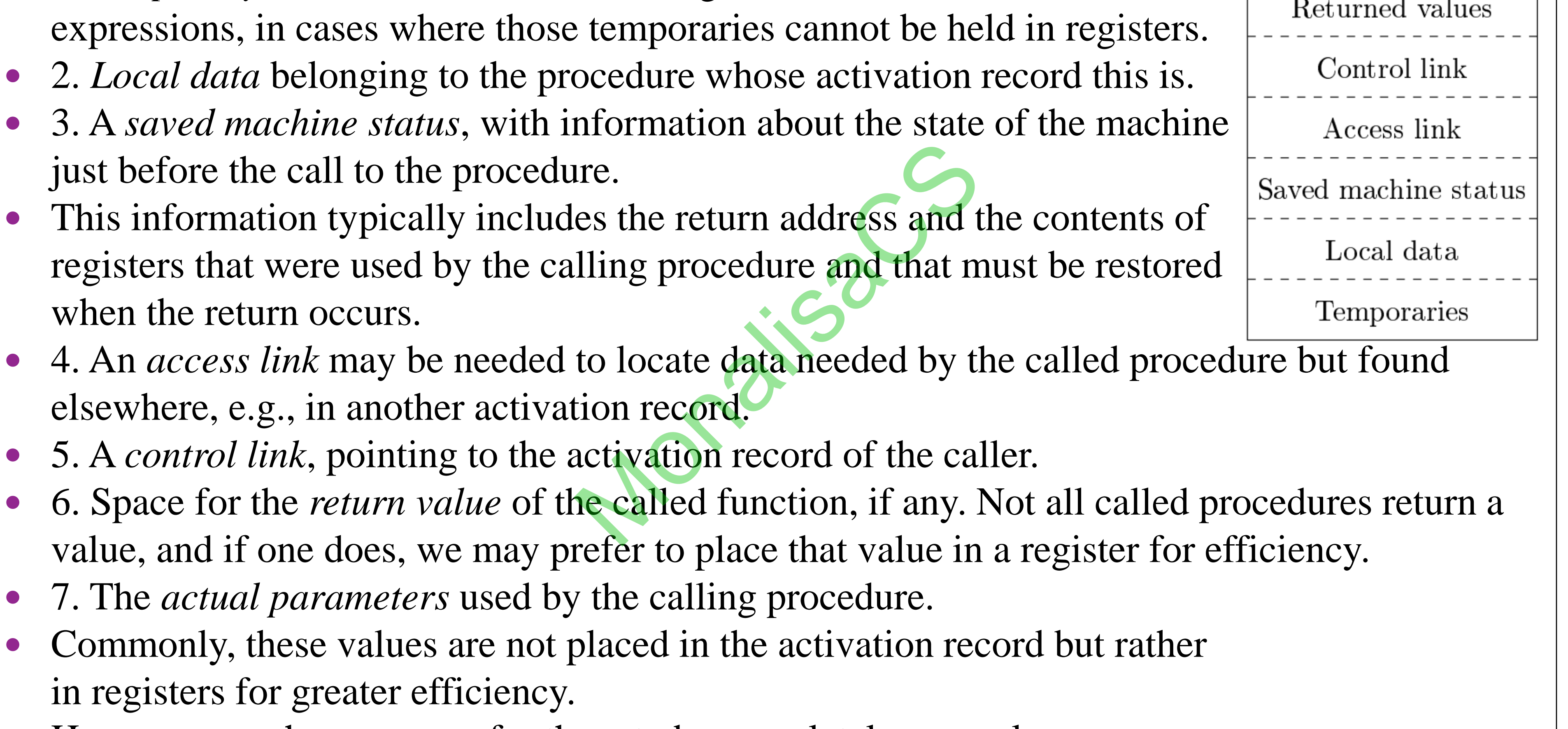
- Procedure calls and returns are usually managed by a run-time stack called the *control stack*.
- Each live activation has an *activation record* (frame) on the control stack, with the root of the activation tree at the bottom , and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides.
- The latter activation has its record at the top of the stack.


MonalisaCS

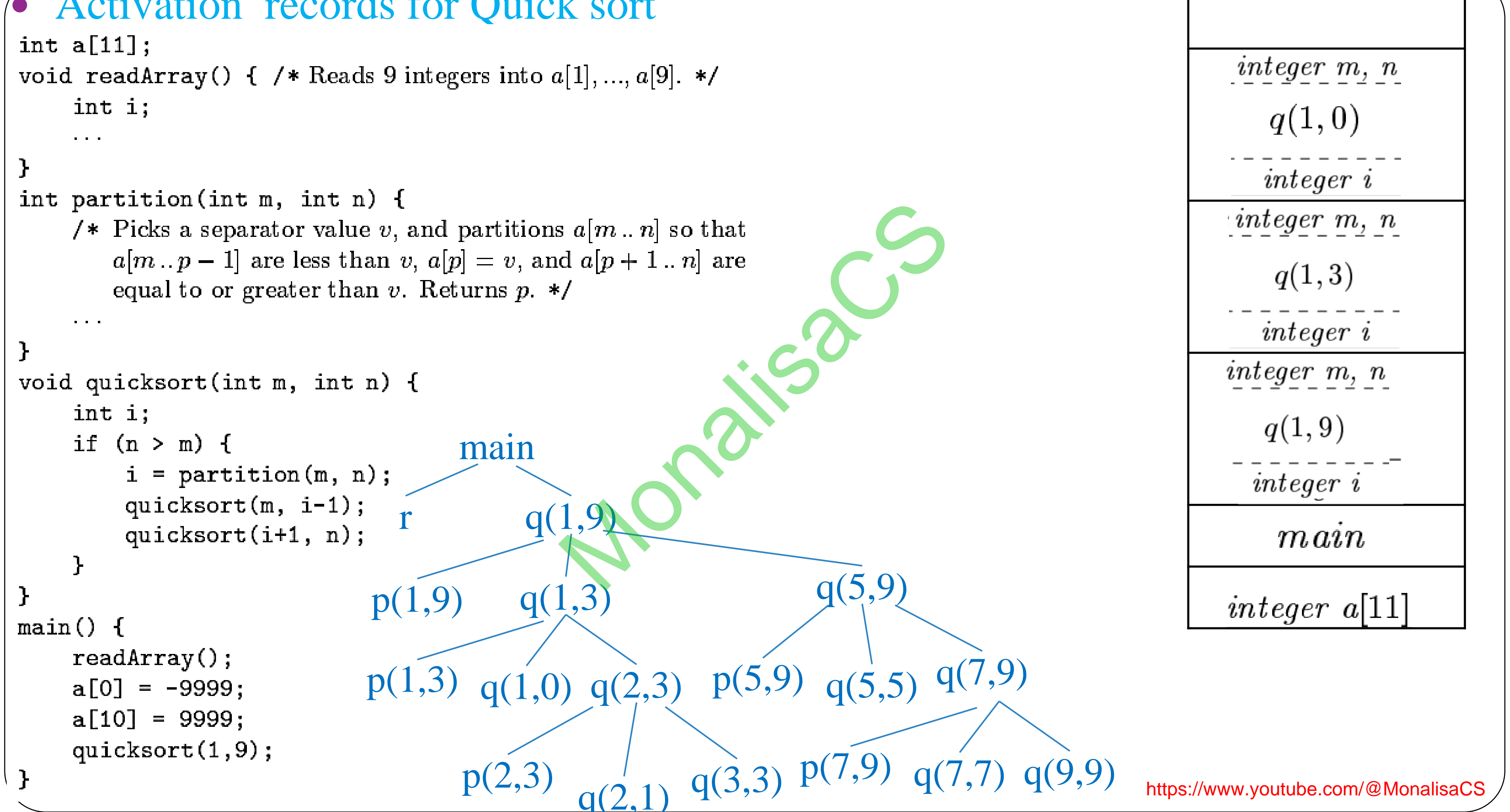
https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- A general activation record →
- 1. *Temporary values*, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.
- 2. *Local data* belonging to the procedure whose activation record this is.
- 3. A *saved machine status*, with information about the state of the machine just before the call to the procedure.
- This information typically includes the return address and the contents of registers that were used by the calling procedure and that must be restored when the return occurs.
- 4. An *access link* may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record.
- 5. A *control link*, pointing to the activation record of the caller.
- 6. Space for the *return value* of the called function, if any. Not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.
- 7. The *actual parameters* used by the calling procedure.
- Commonly, these values are not placed in the activation record but rather in registers for greater efficiency.
- However, we show a space for them to be completely general.



MonalisaCS
https://www.youtube.com/@MonalisaCS


https://monalisacs.com/
Activation records for Quick sort
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
int i;
...
}
int partition(int m, int n) {
/* Picks a separator value v, and partitions a[m .. n] so that
a[m .. p − 1] are less than v, a[p] = v, and a[p + 1 .. n] are
equal to or greater than v. Returns p. */
...
}
void quicksort(int m, int n) {
int i;
if (n > m) {
i = partition(m, n);
quicksort(m, i-1);
quicksort(i+1, n);
}
}
main() {
readArray();
a[0] = -9999;
a[10] = 9999;
quicksort(1,9);
}
main
r
q(1,9)
p(1,9)
q(1,3)
q(5,9)
p(1,3)
q(1,0)
q(2,3)
p(5,9)
q(5,5)
q(7,9)
p(2,3)
q(2,1)
q(3,3)
p(7,9)
q(7,7)
q(9,9)
integer m, n
q(1, 0)
integer i
integer m, n
q(1, 3)
integer i
integer m, n
q(1, 9)
integer i
main
integer a[11]
MonalisaCS
https://www.youtube.com/@MonalisaCS

- **Calling Sequences**
- Procedure calls are implemented by what are known as *calling sequences*, which consists of code that allocates an activation record on the stack and enters information into its fields.
- A *return sequence* is similar code to restore the state of the machine so the calling procedure can continue its execution after the call.
- The code in a calling sequence is often divided between the calling procedure (the "caller") and the procedure it calls (the "callee").
- **Access to Nonlocal Data on the Stack**
- In the C family of languages, all variables are defined either within a single function or outside any function ("globally").
- A global variable v has a scope consisting of all the functions that follow the declaration of v, except where there is a local definition of the identifier v.
- Variables declared within a function have a scope consisting of that function only, or part of it, if the function has nested blocks.
- Allocation of storage for variables and access to those variables is simple:
- 1. Global variables are allocated static storage.
- The locations of these variables remain fixed and are known at compile time.
- So to access any not local variable we use the statically determined address.

MonalisaCS

- 2. Any other name must be local to the activation at the top of the stack.
- We may access these variables through the *top_sp* pointer of the stack.
- An important benefit of static allocation for global is that declared procedures may be passed as parameters or returned as results , with no substantial change in the data-access strategy.
- The scope of a declaration of x is the region of the program in which uses of x refer to this declaration.
- A language uses static scope or lexical scope if it is possible to determine the scope of a declaration by looking only at the program.
- Otherwise , the language uses dynamic scope.
- With dynamic scope, as the program runs , the same use of x could refer to any of several different declarations of x.
- Most languages, such as C and Java, use static scope.
- **Nesting Depth**
- Let us give nesting depth 1 to procedures that are not nested within any other procedure.
- For example, all C functions are at nesting depth 1.
- However, if a procedure p is defined immediately within a procedure at nesting depth i, then give p the nesting depth i + 1

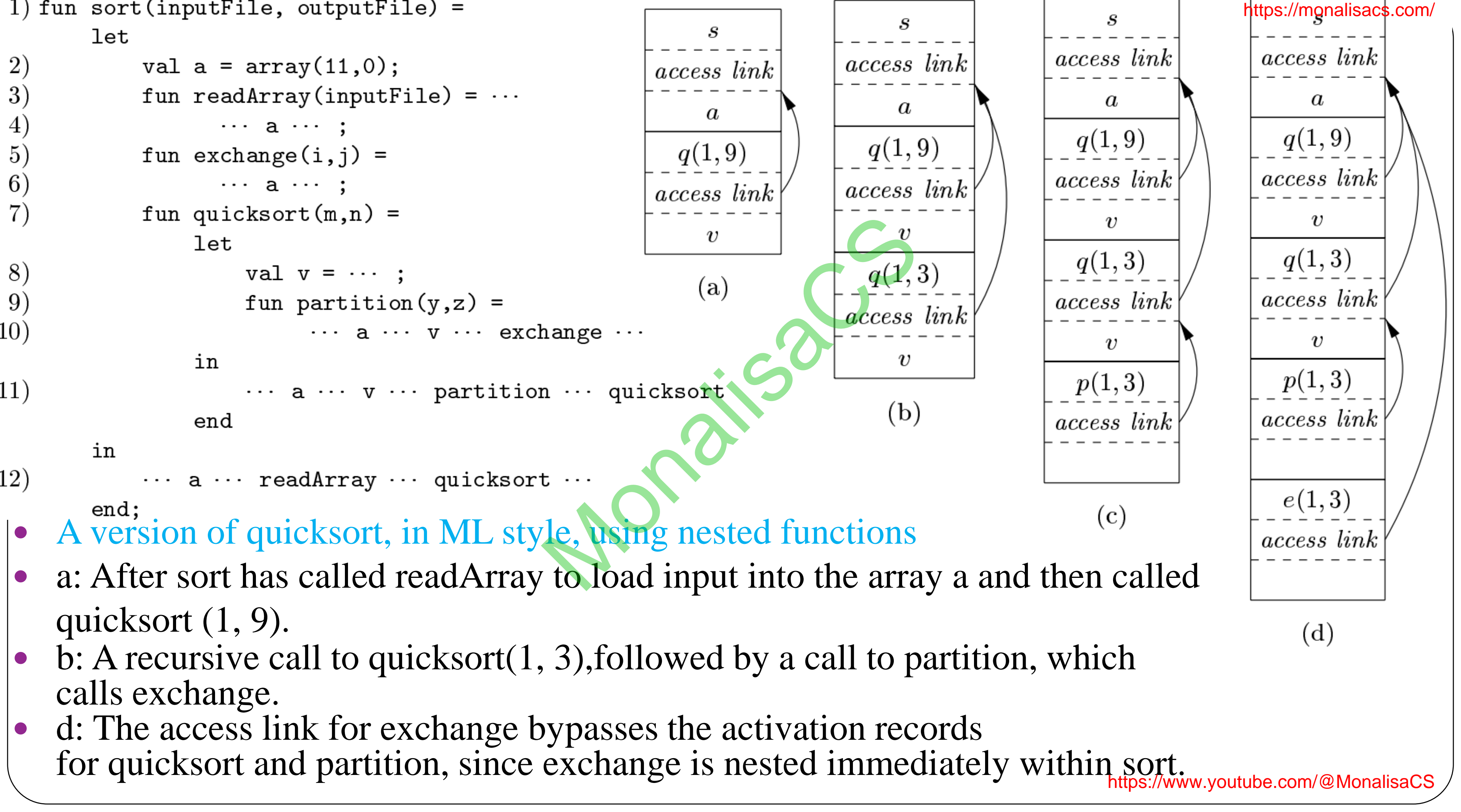
MonalisaCS

https://monalisacs.com/

- **Access Links**
- A direct implementation of the normal static scope rule for nested functions is obtained by adding a pointer called the *access link* to each activation record.
- If procedure p is nested immediately within procedure q in the source code, then the access link in any activation of p points to the most recent activation of q.
- Note that the nesting depth of q must be exactly one less than the nesting depth of p.
- Access links form a chain from the activation record at the top of the stack to a sequence of activations at progressively lower nesting depths.
- Along this chain are all the activations whose data and procedures are accessible to the currently executing procedure.
- Suppose that the procedure p at the top of the stack is at nesting depth n_p, and p needs to access x, which is an element defined within some procedure q that surrounds p and has nesting depth n_q.
- $n_q \leq n_p$, with equality only if p and q are the same procedure.
- To find x, we start at the activation record for p at the top of the stack and follow the access link n_p- n_q times , from activation record to activation record.
- Since the compiler knows the layout of activation records, x will be found at some fixed offset from the position in q's activation record.

MonalisaCS

https://www.youtube.com/@MonalisaCS



https://monalisacs.com/
1) fun sort(inputFile, outputFile) =
let
2) val a = array(11,0);
3) fun readArray(inputFile) = ···
4) ··· a ··· ;
5) fun exchange(i,j) =
6) ··· a ··· ;
7) fun quicksort(m,n) =
let
8) val v = ··· ;
9) fun partition(y,z) =
10) ··· a ··· v ··· exchange ···
in
11) ··· a ··· v ··· partition ··· quicksort
end
in
12) ··· a ··· readArray ··· quicksort ···
end;
s
access link
a
q(1, 9)
access link
v
(a)
s
access link
a
q(1, 9)
access link
v
q(1, 3)
access link
v
(b)
s
access link
a
q(1, 9)
access link
v
q(1, 3)
access link
v
p(1, 3)
access link
(c)
s
access link
a
q(1, 9)
access link
v
q(1, 3)
access link
v
p(1, 3)
access link
e(1, 3)
access link
(d)
• A version of quicksort, in ML style, using nested functions
• a: After sort has called readArray to load input into the array a and then called quicksort (1, 9).
• b: A recursive call to quicksort(1, 3),followed by a call to partition, which calls exchange.
• d: The access link for exchange bypasses the activation records for quicksort and partition, since exchange is nested immediately within sort.
MonalisaCS
https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

Heap Management

- The heap is the portion of the store that is used for data that lives indefinitely, or until the program explicitly deletes it.
- Local variables typically become inaccessible when their procedures end.
- **The Memory Manager:**
- Memory manager allocates and deallocates space within the heap;
- It serves as an interface between application programs and the operating system.
- The memory manager keeps track of all the free space in heap storage at all times.
- It performs two basic functions:
- **Allocation**. When a program requests memory for a variable or object , the memory manager produces a chunk of contiguous heap memory of the requested size.
- If possible, it satisfies an allocation request using free space in the heap;
- If no chunk of the needed size is available, it seeks to increase the heap storage space by getting consecutive bytes of virtual memory from the operating system.
- **Deallocation.** The memory manager returns deallocated space to the pool of free space, so it can reuse the space to satisfy other allocation requests.
- Here are the properties we desire of memory managers:
- **Space Efficiency.** A memory manager should minimize the total heap space needed by a program.
- Space efficiency is achieved by minimizing “fragmentation”.

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- **Program Efficiency.** A memory manager should make good use of the memory subsystem to allow programs to run faster.
- **Low Overhead.** Because memory allocations and deallocations are frequent operations in many programs, it is important that these operations be as efficient as possible.
- That is, we wish to minimize the overhead the fraction of execution time spent performing allocation and deallocation.
- Reducing Fragmentation
- At the beginning of program execution, the heap is one contiguous unit of free space.
- As the program allocates and deallocates memory, this space is broken up into free and used chunks of memory, and the free chunks need not reside in a contiguous area of the heap.
- We refer to the free chunks of memory as holes.
- Garbage Collection
- Data that cannot be referenced is known as garbage.
- Many high-level programming languages remove the burden of manual memory management from the programmer by offering automatic garbage collection, which deallocates unreachable data.

MonalisaCS

https://www.youtube.com/@MonalisaCS