COMPILER DESIGN

HANDLED BY
DIVYA B
DEPT. OF CSE
VJEC, CHEMPERI

INTERMEDIATE CODE GENERATION

- The front-end of the compiler translates a source program into an intermediate representation from which the back end generates target code.

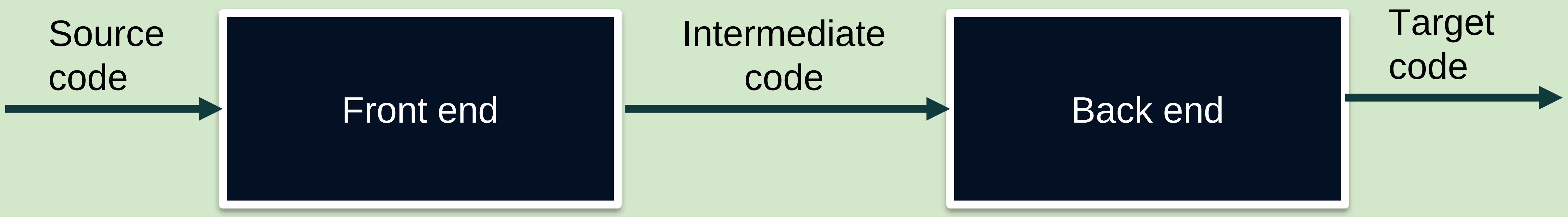
INTERMEDIATE CODE GENERATION

Need for ICC

1. If a compiler translates the source language to its target machine language without generating IC, then for each new machine, a full native compiler is required.
2. Synthesis part of back end depends on the target machine.
3. A machine Independent Code-Optimizer can be applied to the Intermediate Representation.

INTERMEDIATE CODE GENERATION

- IC Generation process should not be very complex.
- It shouldn’t be difficult to produce the target program from the intermediate code.



INTERMEDIATE LANGUAGES

- Abstract Syntax tree or Syntax Tree
- DAG (Directed Acyclic Graph)
- Postfix Notation
- Three Address Code

INTERMEDIATE LANGUAGES

Abstract Syntax tree or Syntax Tree

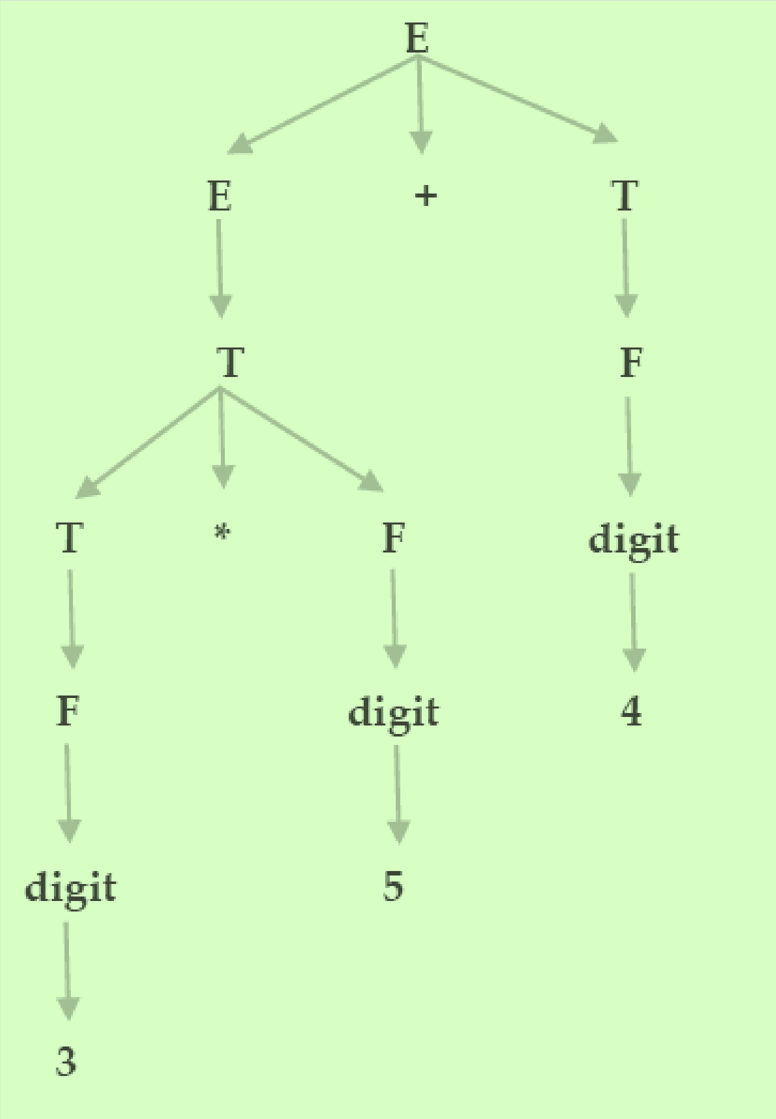
- ✓ Graphical Intermediate Representation.
- ✓ Syntax Tree depicts the hierarchical structure of a source program.
- ✓ Syntax tree (AST) is a condensed form of parse tree useful for representing language constructs.

Generate the parse tree and the syntax tree for 3 * 5 + 4.

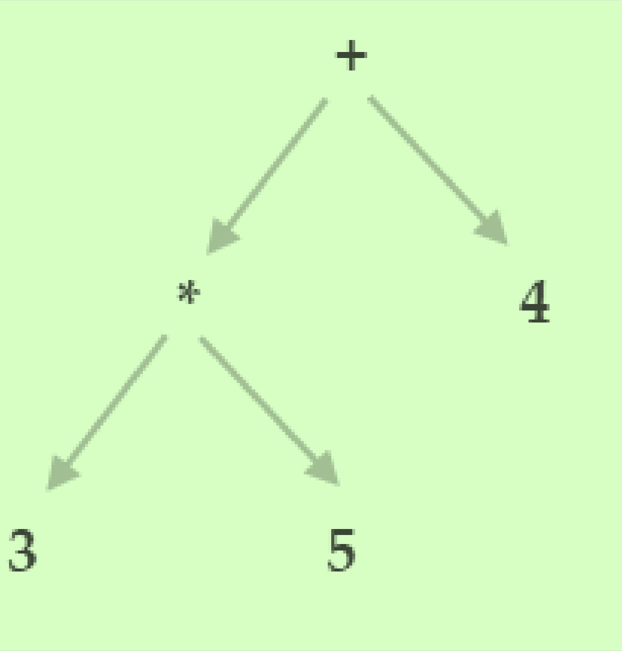
Grammar

E → E + T
E → E - T
E → T
T → T * F
T → F
F → digit

Parse Tree



Syntax Tree



Parse Tree	Syntax Tree
A parse tree is a graphical representation of a replacement process in a derivation	A syntax tree (AST) is a condensed form of parse tree
Each interior node represents a grammar rule	Each interior node represents an operator
Each leaf node represents a terminal	Each leaf node represents an operand
Parse tree represent every detail from the real syntax	Syntax tree does not represent every detail from the real syntax Eg : No parenthesis

INTERMEDIATE LANGUAGES

Constructing Syntax Tree for Expression

- ✓ Each node in a syntax tree can be implemented in a record with several fields.
- ✓ In the node of an operator, one field contains operator and remaining field contains pointer to the nodes for the operands.
- ✓ When used for translation, the nodes in a syntax tree may contain addition of fields to hold the values of attributes attached to the node.

INTERMEDIATE LANGUAGES

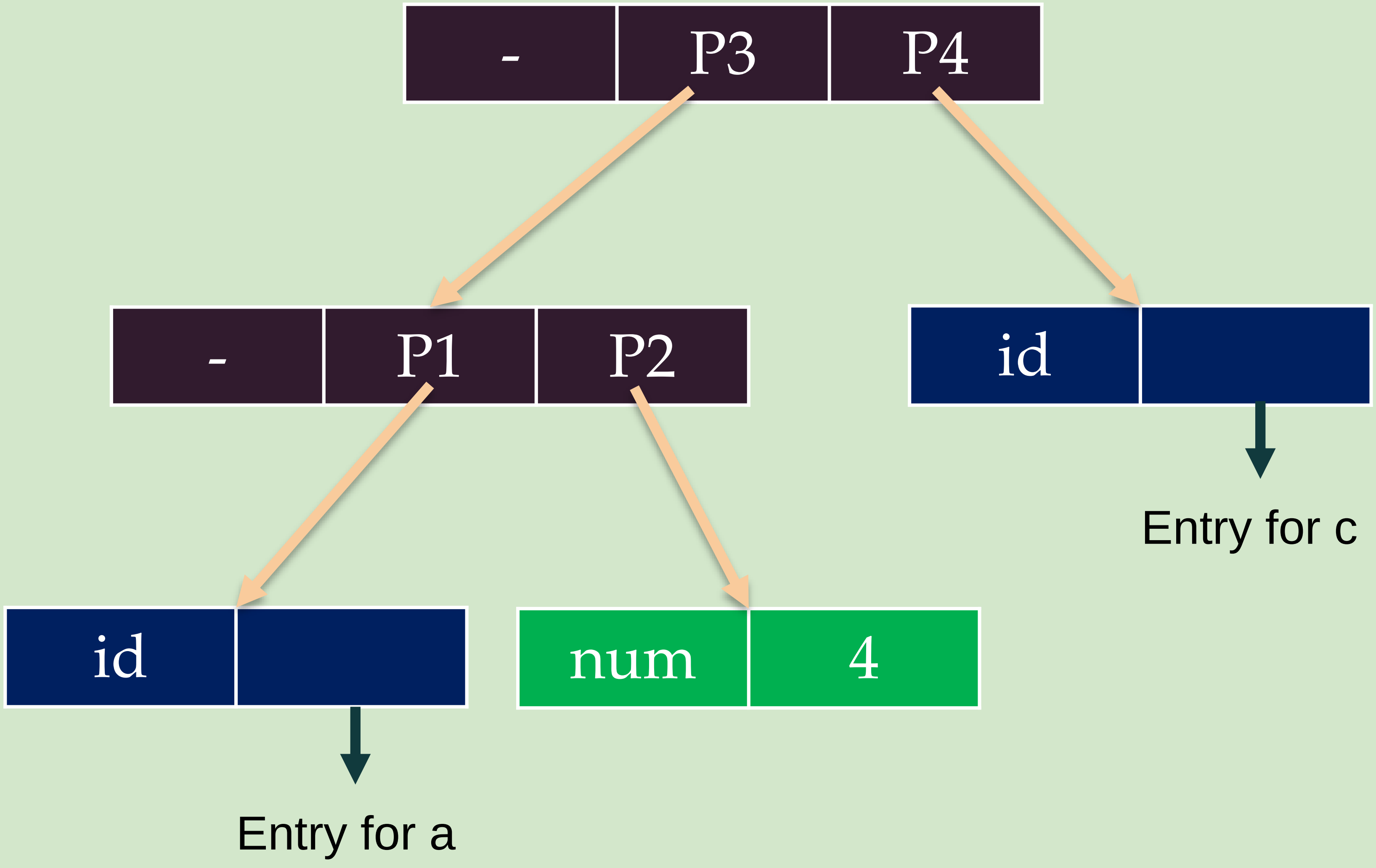
Constructing Syntax Tree for Expression

- ✓ **mknode(op,left,right)**: creates an operator node with label op and two fields containing pointers to left and right.
- ✓ **mkleaf(id,entry)**: creates an identifier node with label id and a field containing entry, a pointer to the symbol table entry for identifier.
- ✓ **mkleaf(num,val)**: creates a number node with label num and a field containing val, the value of the number.
- ✓ Such functions return a pointer to a newly created node.

E.g. Construct Syntax tree for a – 4 + c.

- ✓ The tree is constructed bottom up.

P1 = mkleaf(id,entry a)
P2 = mkleaf(num, 4)
P3 = mknode(-, P1, P2)
P4 = mkleaf(id,entry c)
P5 = mknode(+, P3, P4)



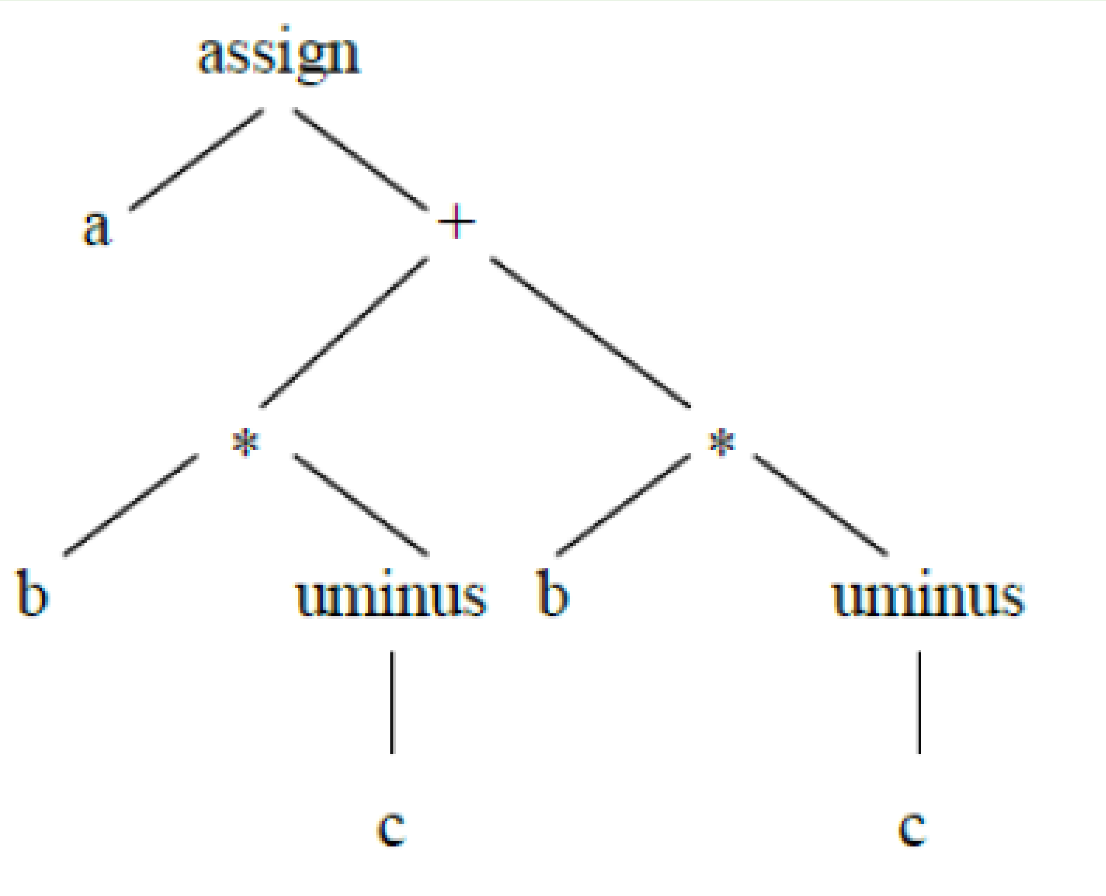
INTERMEDIATE LANGUAGES

Directed Acyclic Graph (DAG)

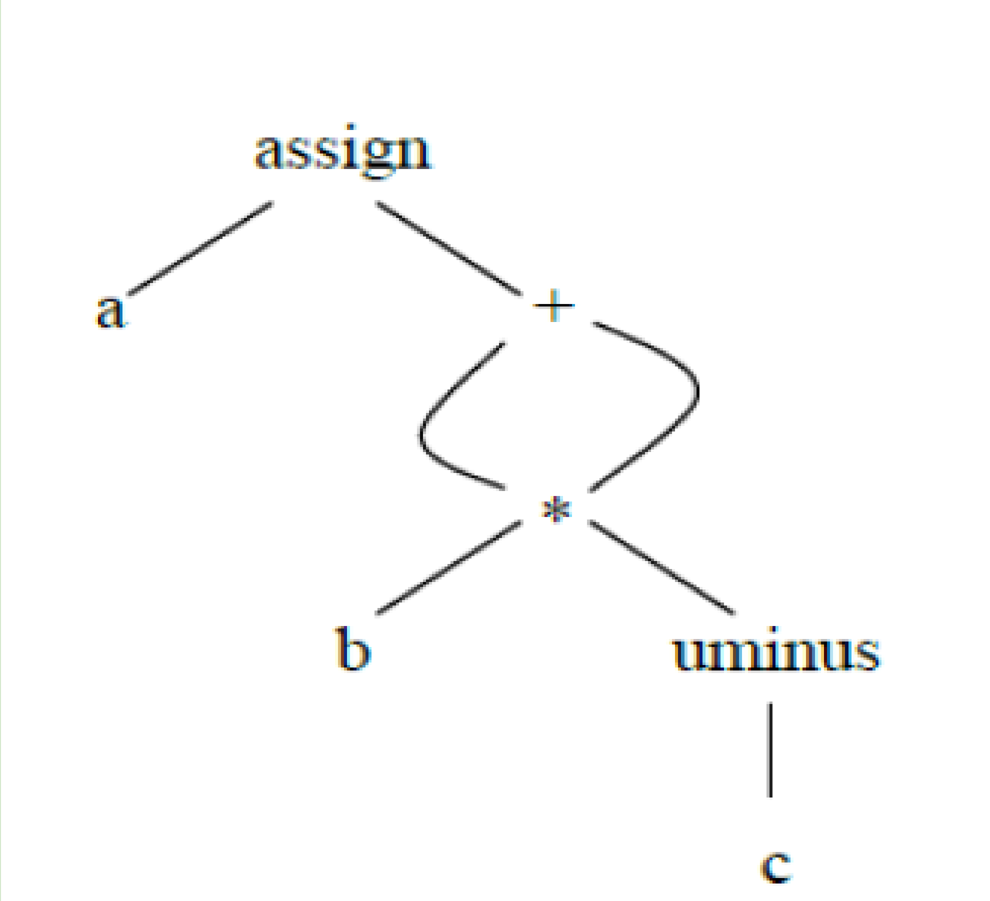
- ✓ Graphical Intermediate Representation.
- ✓ DAG also gives the hierarchical structure of source program but in a more compact way because common sub expressions are identified.

Constructing DAG for Expression

$$a = b * -c + b * -c$$



Syntax Tree



DAG

INTERMEDIATE LANGUAGES

Postfix

- ✓ Linearized representation of syntax tree.
- ✓ In postfix notation, each operator appears immediately after its last operand.
- ✓ Operators can be evaluated in the order in which they appear in the string.

INTERMEDIATE LANGUAGES

Postfix rules

- ✓ If E is a variable or constant, then the postfix notation for E is E itself.
- ✓ If E is an expression of the form E1 op E2 then postfix notation for E is E1′ E2′ op, here E1′ and E2′ are the postfix notations for E1and E2, respectively.
- ✓ If E is an expression of the form (E), then the postfix notation for E is the same as the postfix notation for E.
- ✓ For unary operation –E the postfix is E-
- ✓ Postfix notation of an infix expression can be obtained using stack.

Find the postfix of 9-(5+2)

952+-

INTERMEDIATE LANGUAGES

Three Address Code

- ✓ The reason for the term “three address code” is that each statement contains 3 addresses at most. Two for the operands and one for the result.

INTERMEDIATE LANGUAGES

General form of Three Address Code

- **a = b op c**
  - **a, b, c** are the operands that can be names, constants or compiler generated temporaries.
  - **op** represents operator, such as fixed or floating point arithmetic operator or a logical operator on Boolean valued data.

INTERMEDIATE LANGUAGES

General form of Three Address Code

- ✓ A source language expression like **x + y * z** might be translated into a sequence

  t1 = y*z

  t2 = x+t1 where, **t1** and **t2** are compiler generated temporary names.

INTERMEDIATE LANGUAGES

Advantages of Three Address Code

- ✓ The unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.
- ✓ The use of names for the intermediate values computed by a program allows three-address code to be easily rearranged - unlike postfix notation.

Types of Three Address Statements

- ✓ Assignment statements

  x = y op z, where op is a binary arithmetic or logical operation.
- ✓ Unary operations

  x = ***op*** **y**, where op is a unary operation . Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that for example, convert a fixed-point number to a floating-point number.

Types of Three Address Statements

- ✓ Copy statements

  x = y where the value of y is assigned to x.
- ✓ Unconditional jump

  goto L The three-address statement with label L is the next to be executed.

Types of Three Address Statements

- ✓ Conditional jump

  if x relop y goto L This instruction applies a relational operator **(<, =, =, etc,)** to **x** and y, and executes the statement with label **L** next if x stands in relation **relop** to **y.** If not, the three-address statement following if **x relop y goto L** is executed next, as in the usual sequence.

Types of Three Address Statements

- ✓ Procedure call and return

  param x and **call p, n** for procedure calls and **return y**, where **y** representing a returned value is optional.
  Their typical use is as the sequence of three-address statements
  param x_1
  param x_2
  ……….
  param x_n
  call p,n
  generated as part of the call procedure **p(x_1 , x_2, . . . , x_n)**
  The integer **n** indicates the number of actual-parameters.

Types of Three Address Statements

- ✓ Indexed Assignments

  Indexed assignments of the form **x = y[i] or x[i] = y**
- ✓ **Address and pointer assignments**

  Address and pointer operator of the form **x = &y**, **x = *y** and ***x = y**

Implementation of Three Address Statements

- ✓ In a compiler, three address statements can be implemented as records with fields for the operator and the operands. Three such, representations are
  - ✓Quadruples
  - ✓Triples
  - ✓Indirect triples

Implementation of Three Address Statements

- ✓ Quadruples
  - A quadruple is a record structure with four fields, which are ***op, ag1, arg2*** and ***result.***
  - The op field contains an internal code for the operator. The three address statement **x = y op z** is represented by placing **y** in ***arg1,*** **z** in ***arg2*** and **x** in ***result***.
  - The contents of ***arg1, arg2***, and ***result*** are normally pointers to the symbol table entries for the names represented by these fields. If so temporary names must be entered into the symbol table as they are created.

Implementation of Three Address Statements

- ✓ Quadruples
  - A quadruple is a record structure with four fields, which are ***op, ag1, arg2*** and ***result.***
  - The op field contains an internal code for the operator. The three address statement **x = y op z** is represented by placing **y** in ***arg1,*** **z** in ***arg2*** and **x** in ***result***.
  - The contents of ***arg1, arg2***, and ***result*** are normally pointers to the symbol table entries for the names represented by these fields. If so temporary names must be entered into the symbol table as they are created.

Implementation of Three Address Statements

- ✓ Quadruples
  - The statement **x := op y**, where op is a unary operator is represented by placing **op** in the operator field, **y** in the argument field & n in the result field. The ***arg2*** is not used.
  - A statement like **param t1** is represented by placing **param** in the operator field and t1 in the arg1 field. Neither ***arg2*** nor result field is used.
  - Unconditional & Conditional jump statements are represented by placing the target in the result field.

Implementation of Three Address Statements

- ✓ Triples
  - In triples representation, the use of temporary variables is avoided & instead reference to instructions are made to three address statements can be represented by records with only there fields OP, arg1 & arg2.
  - Since, there fields are used this intermediated code formal is known as triples.
  - Advantages
    - No need to use temporary variable which saves memory as well as time.

Implementation of Three Address Statements

- ✓ Triples
  - Disdvantages
    - Triple representation is difficult to use for optimizing compilers. Because for optimization statements need to be shuffled.
    - E.g., statement 1 can come down or statement 2 can go up etc.
    - So the reference we used in the representation will change.

Translate a + b * c / e ^ f + b * a to qudruple and triple.

First construct the three address code.

t1 = e ^ f
t2 = b * c
t3 = t2 / t1
t4 = b * a
t5 = a + t3
t6 = t5 + t4

Location	OP	arg1	arg2	Result
(0)	^	e	f	t1
(1)	*	b	c	t2
(2)	/	t2	t1	t3
(3)	*	b	a	t4
(4)	+	a	t3	t5
(5)	+	t3	t4	t6

Translate a + b * c | e ^ f + b * a to qudruple and triple.

First construct the three address code.

t1 = e ^ f

t2 = b * c

t3 = t2 / t1

t4 = b * a

t5 = a + t3

t6 = t5 + t4

Location	OP	arg1	arg2
(0)	^	e	f
(1)	*	b	c
(2)	/	(1)	(0)
(3)	*	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

A ternary operation like x[i] = y requires two entries in the triple structure while x = y[i] is naturally represented as two operations.

Location	op	arg1	arg2
(0)	[ ]=	x	i
(1)	assign	(0)	y

x[i] = y

Location	op	arg1	arg2
(0)	=[ ]	y	i
(1)	assign	x	(0)

x = y[i]

Implementation of Three Address Statements

- ✓ Indirect Triples
  - In triples representation, the use of temporary variables is avoided & instead reference to instructions are made to three
  - This representation is an enhancement over triple representation.
  - It uses an additional instruction array to led the pointer to the triples in the desired order.
  - It allows the optimizers to easily reposition the sub-expression for producing the optimized code.

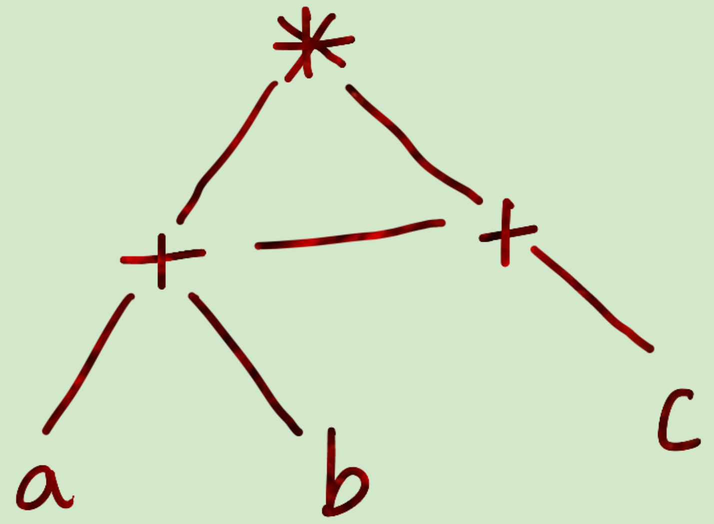
	Statement
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

Location	op	arg1	arg2
(0)	^	E	f
(1)	*	B	c
(2)	/	(1)	(0)
(3)	*	B	a
(4)	+	A	(2)
(5)	+	(4)	(3)

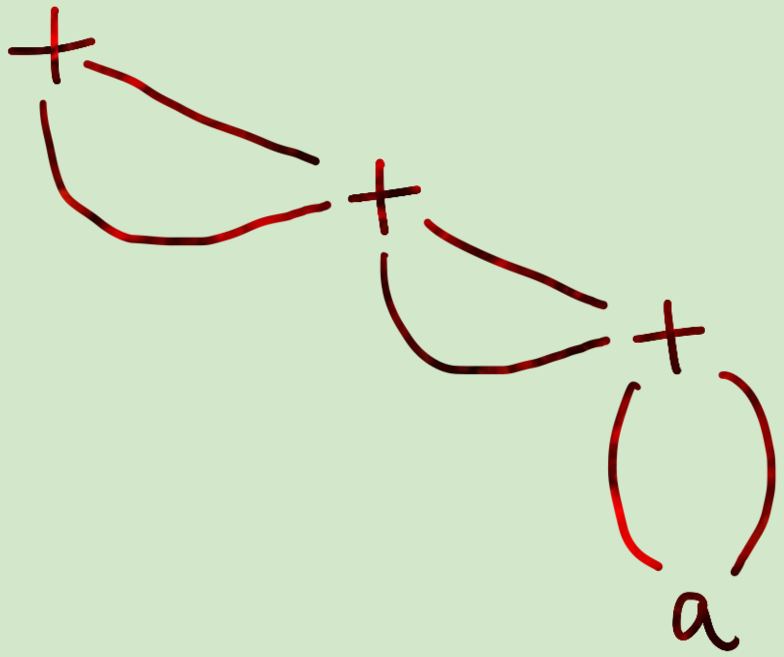
Construct DAG for the following

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

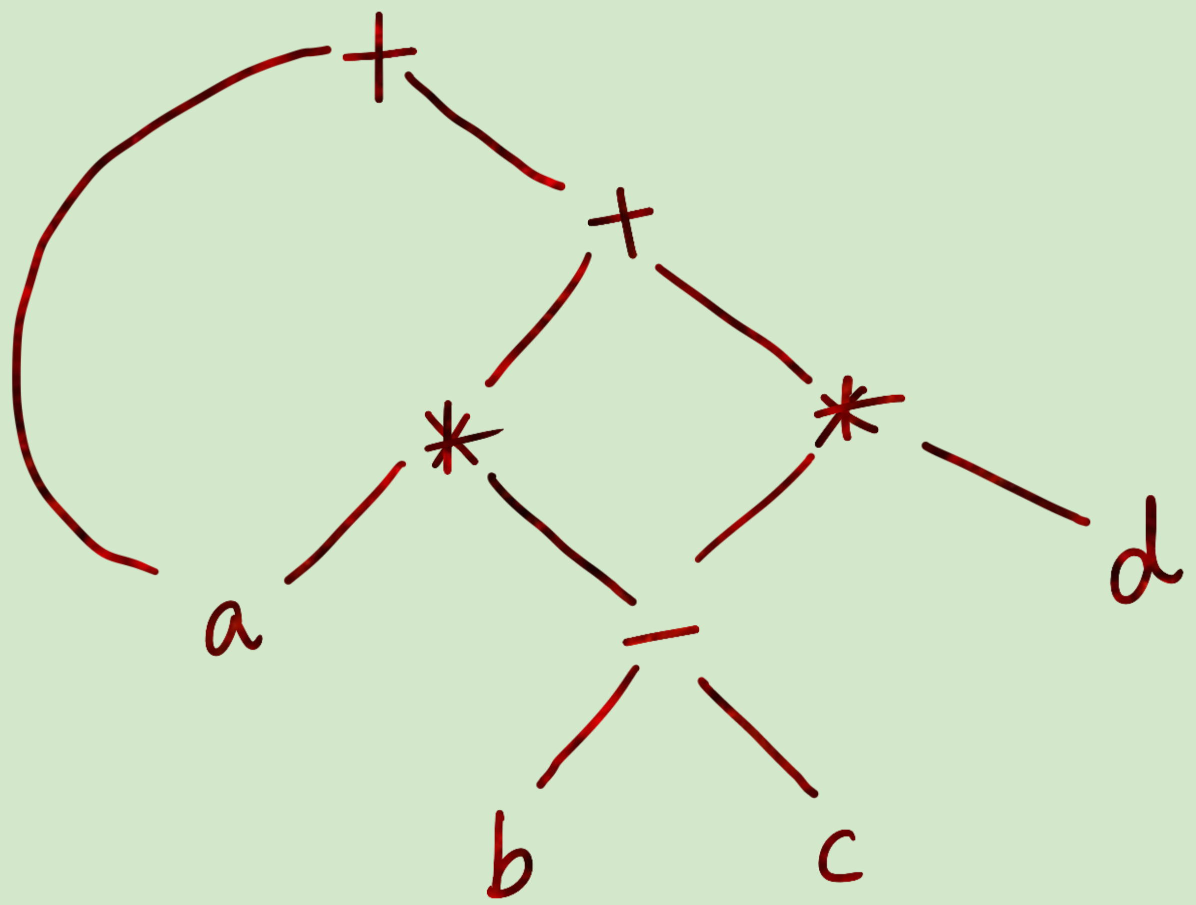
- a=(a*b+c)-(a*b+c)



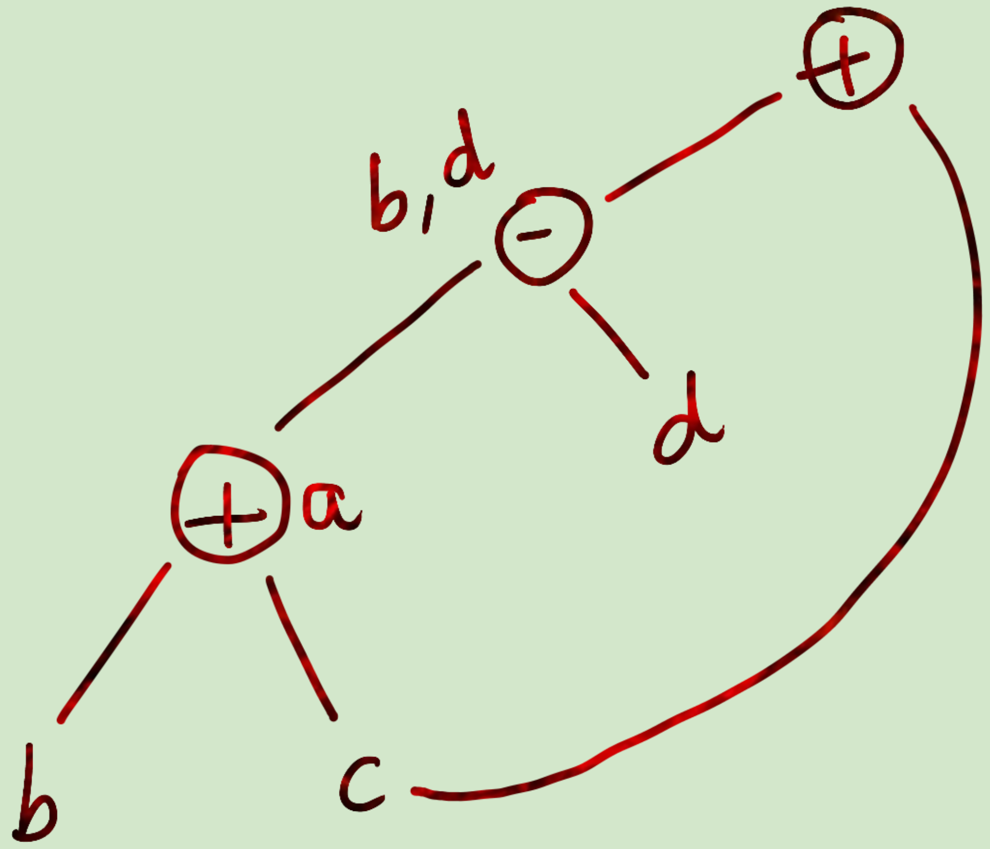
DAG for (a+b) * (a+b+c)



DAG for (((a+a)+(a+a))+(((a+a)+(a+a)))



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

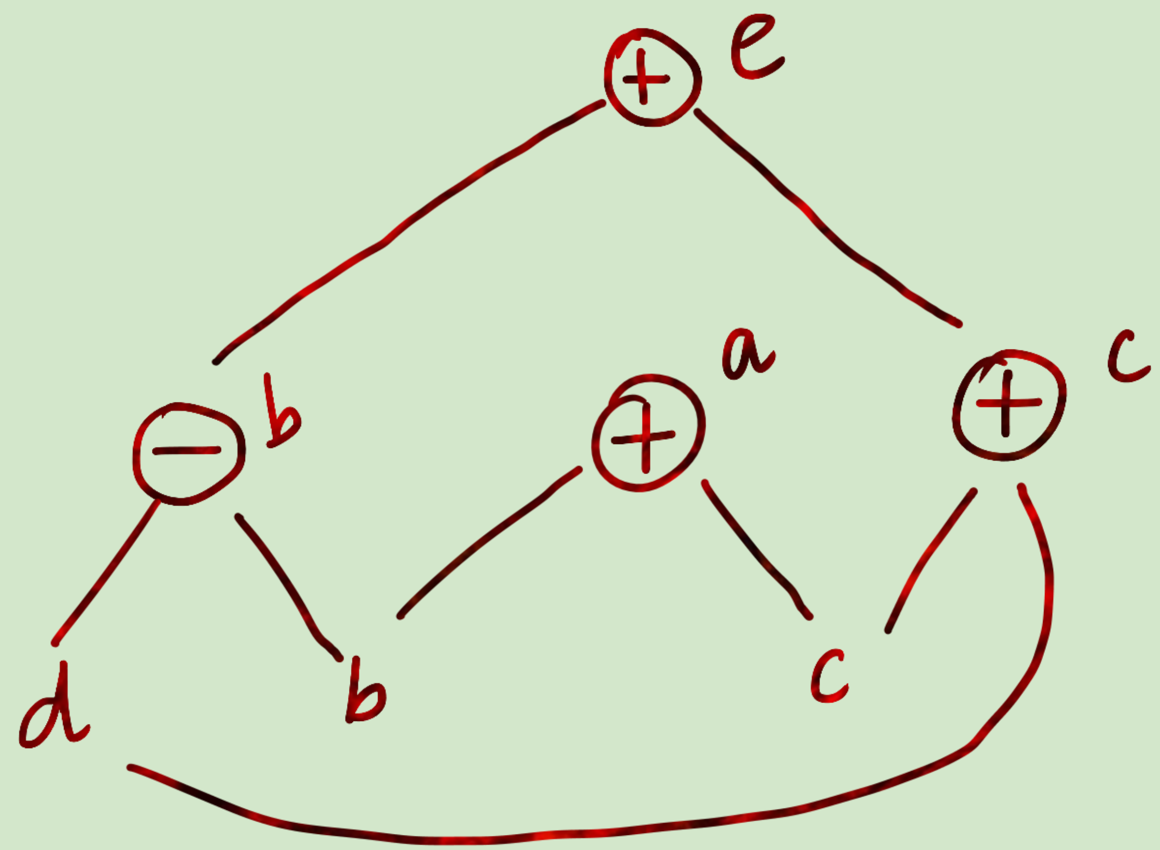


DAG for $a = b + c$

$b = a - d$

$c = b + c$

$d = a - d$

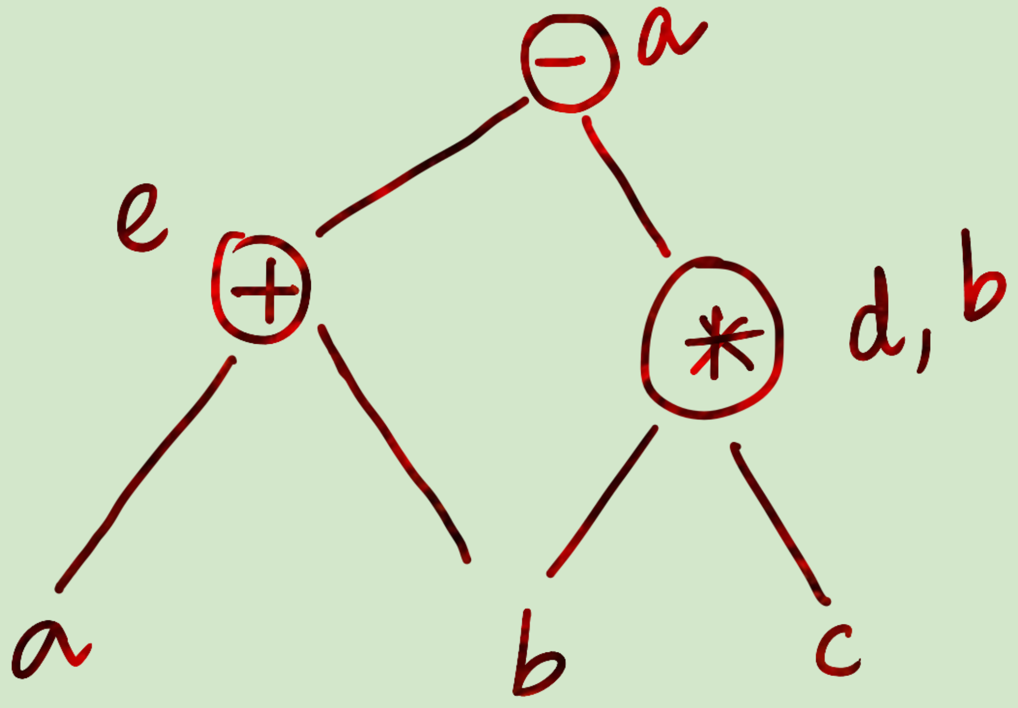


DAG for a = b+c

b = b−d

c = c+d

e = b+c

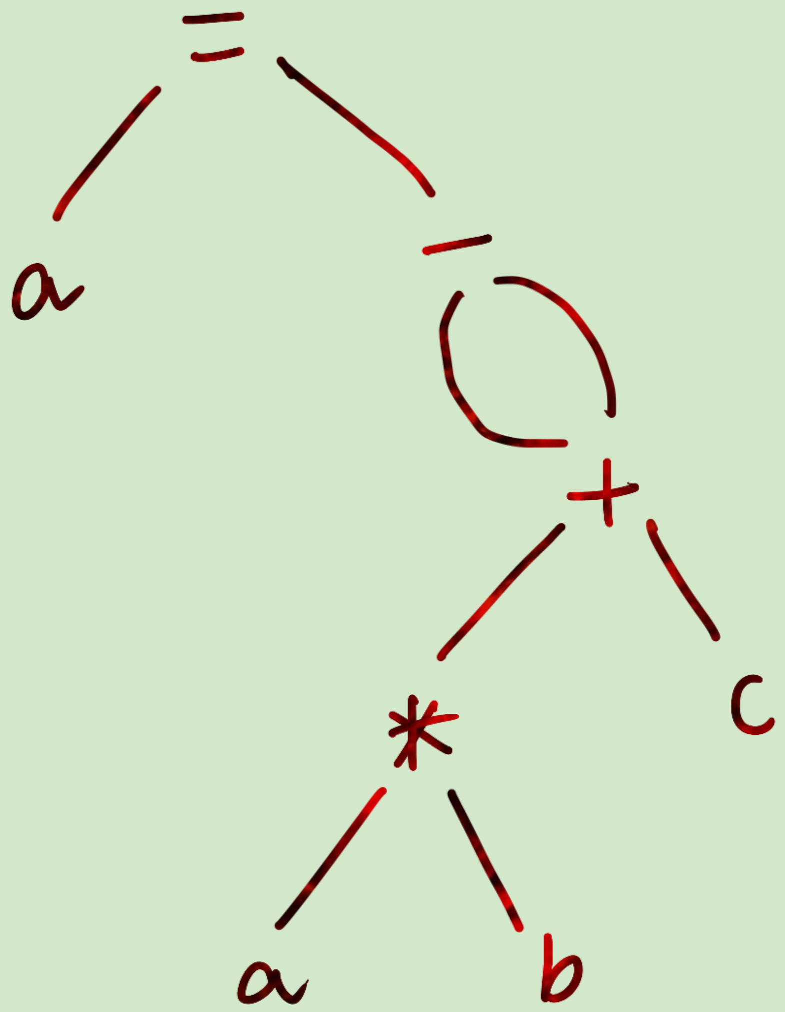


DAG for $d = b*c$

$e = a+b$

$b = b*c$

$a = e-d$



DAG for $a = (a*b+c)-(a*b+c)$

Construct quadruples, triples and indirect triples

A=-B*(C/D)

Three address codes are
T1 = -B
T2 = C/D
T3 = T1 * T2
A = T3

Location	op	arg1	arg2	result
(0)	uminus	B		T1
(1)	/	C	D	T2
(2)	*	T1	T2	T3
(3)	=	T3		A

Quadruple

Construct quadruples, triples and indirect triples

A=-B*(C/D)

Three address codes are
T1 = -B
T2 = C/D
T3 = T1 * T2
A = T3

Location	op	arg1	arg2
(0)	uminus	B	
(1)	/	C	D
(2)	*	(0)	(1)
(3)	=	A	(2)

Triple

Construct quadruples, triples and indirect triple

A=-B*(C/D)

	Statement
21	(0)
22	(1)
23	(2)
24	(3)

Location	op	arg1	arg2
(0)	uminus	B	
(1)	/	C	D
(2)	*	(0)	(1)
(3)	=	A	(2)

Indirect Triple

Construct quadruples, triples and indirect triples

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

Three address codes are
t1 = a+b
t2 = c+d
t3 = t1*t2
t4 = t1+c
t5 = t3–t4

Location	op	arg1	arg2	result
(0)	+	a	b	t1
(1)	=	c	d	t2
(2)	*	t1	t2	t3
(3)	+	t1	c	T4
(4)	-	t3	t4	t5

Quadruple

Divys-Compiler Design PPT

Construct quadruples, triples and indirect triples

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

Three address codes are
t1 = a+b
t2 = c+d
t3 = t1*t2
t4 = t1+c
t5 = t3–t4

Location	op	arg1	arg2
(0)	+	a	b
(1)	=	c	d
(2)	*	(0)	(1)
(3)	+	(0)	c
(4)	-	(2)	(3)

Triple