VI
Intermediate Code Generation

Shyamalendu Kandar
Assistant Professor,
Department of Information Technology
IIEST, Shibpur

Introduction

- program in high level language translated to machine code targeted to some processor.
- many a times compilers are designed to produce an intermediate output.
- It represents the input program in some hypothetical language or data structure.
- Some of these are known as intermediate code to signify that they are representations between the source language and target machine code.

Advantages of Intermediate code

- closer to target machine code, than the source language, hence easier to generate target code.
- simple enough to be Translated to assembly code.
- machine independent code optimization. This helps in faster generation of code.
- easier to generate more abstract code, not bothering too much about things like register allocations,
- compiler bugs can be spotted more easily.

Forms of Intermediate Representation

In general

- Syntax Tree
- Post Fix notation
- Three Address Code.

Sometimes in two

- Graphical Representation
  - ✓ **Syntax Tree**
  - ✓ **DAG**
- Three Address Code.

Some times in another two:

- High-level Intermediate Representation
  - ✓ **Syntax Tree**
  - ✓ **DAG**
  - ✓ **P-Code**
- Low level intermediate Representation (Three Address Code)

Syntax Tree

Syntax Tree

- Depicts the natural hierarchical structure of a source language.
- Can be produced by syntax directed definition. [Node are constructed]
- Can be represented in two ways
  1. Each node is represented as a record with a field for its operator and additional fields for pointer to its children.
  2. Node are allocated from an array of records and index of position of the node serves as the pointer to the node.

Example:

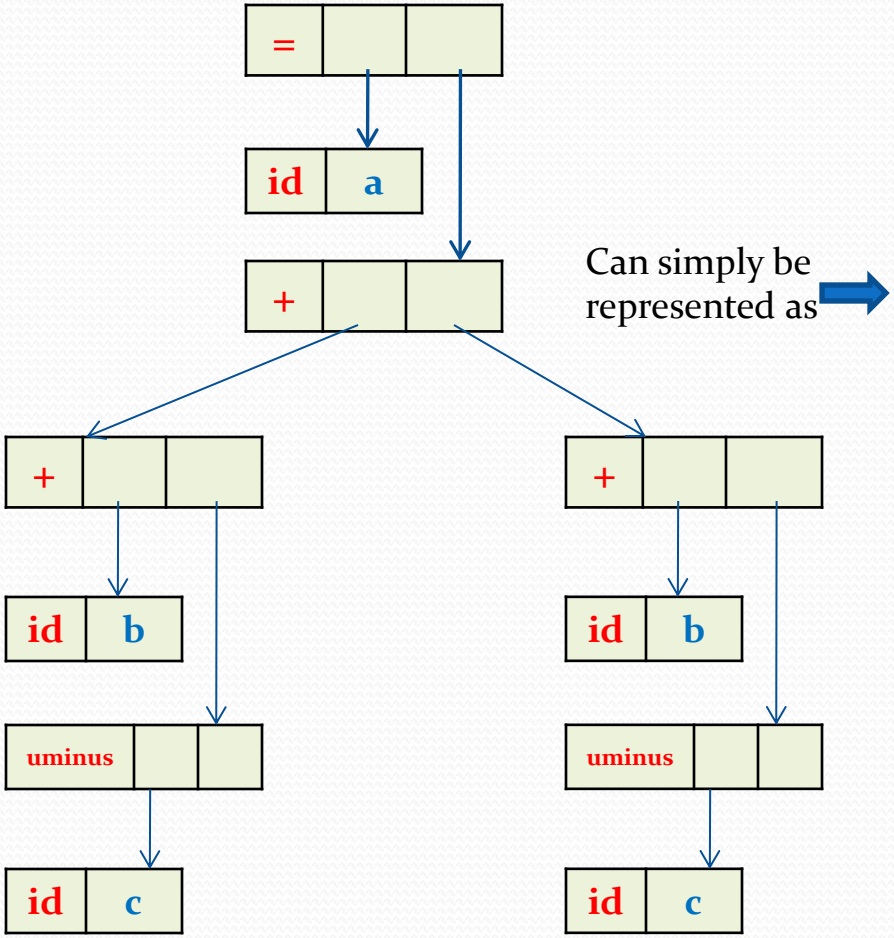
SDD for syntax tree for assignment statement

Production	Semantic Rule
S →id:=E	S.nptr = mknode("=", mkleaf(id, id.place), E.nptr)
$E \rightarrow E_1 + E_2$	E.nptr = mknode("+", E_1.nptr , E_2.nptr)
$E \rightarrow E_1 * E_2$	E.nptr = mknode("*", E_1.nptr , E_2.nptr)
$E \rightarrow -E_1$	E.nptr = mknode("uminis", E_1.nptr)
$E \rightarrow (E_1)$	E.nptr =E_1.nptr
E →id	E.nptr=mkleaf(id, id.place)

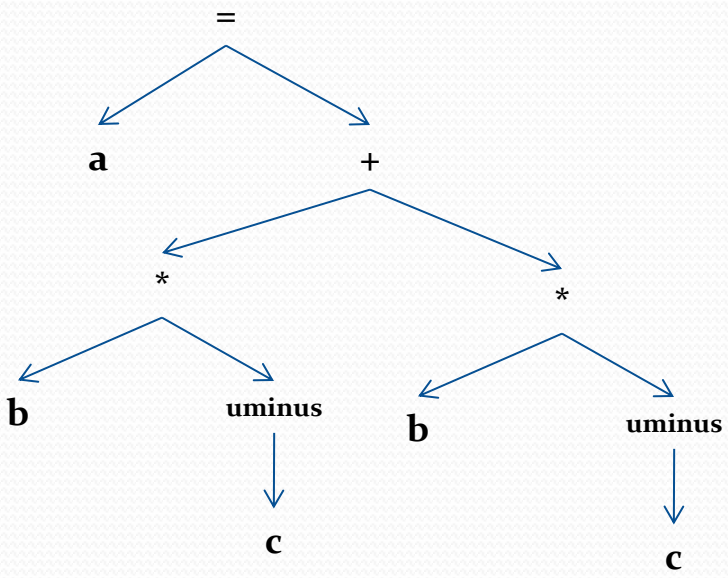
Syntax Tree

- Syntax tree for a=b*-c+b*-c

Type 1:



Can simply be represented as



Assignment

- Design Syntax Tree for a+a*(b-c)+(b-c)*d

Syntax Tree

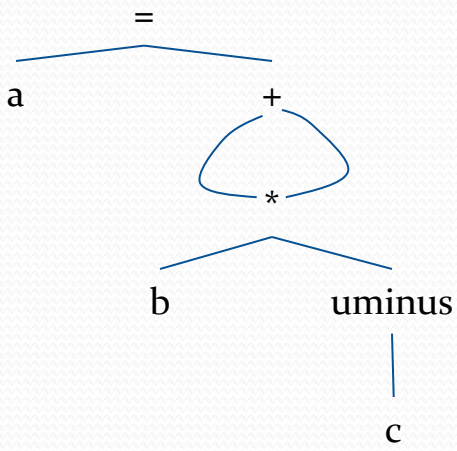
Type 2:

0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	=	9	8
11			

DAG

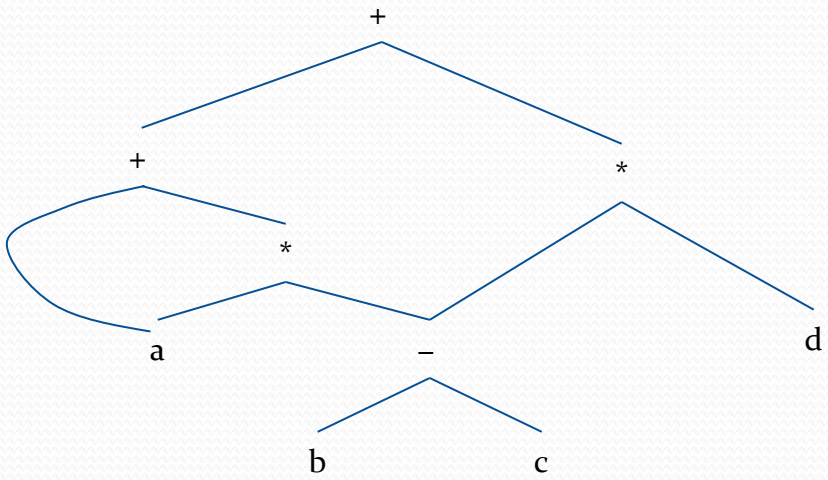
- Directed Acyclic graph.
- DAG gives same information of syntax tree but in a more compact way.
- Common sub-expressions are identified.
- Unique node for each value.

a=b*-c+b*-c



DAG

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

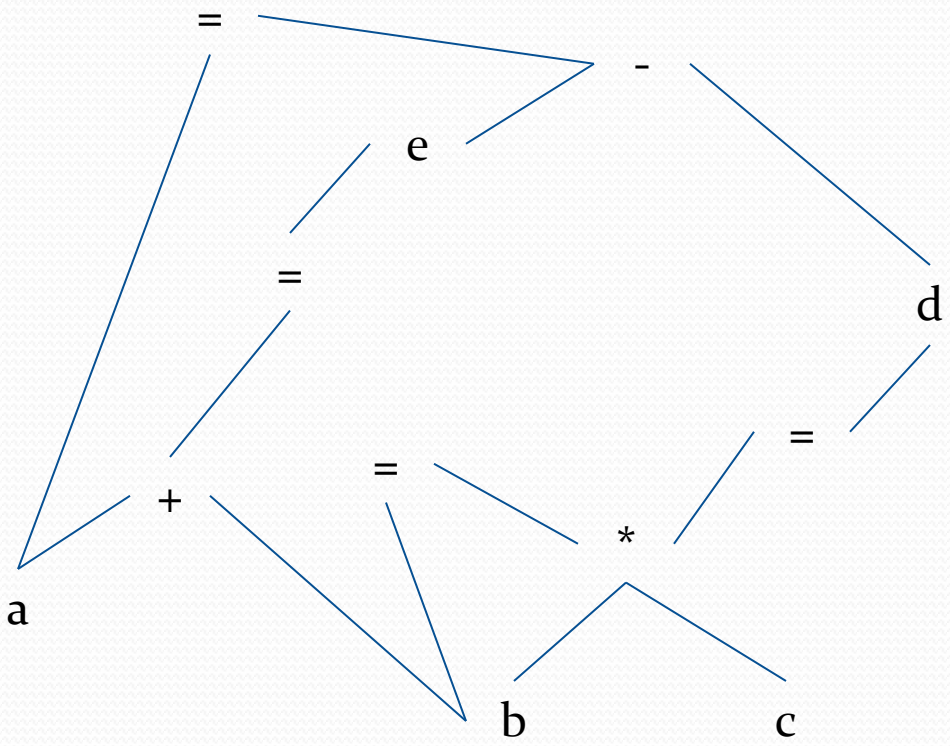


SDD to produce syntax Tree or DAG

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node =$ **new** $Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node =$ **new** $Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow$ **id**	$T.node =$ **new** $Leaf($**id**, **id**.$entry)$
6)	$T \rightarrow$ **num**	$T.node =$ **new** $Leaf($**num**, **num**.$val)$

DAG

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



Postfix Notation

- Also called suffix notation or reverse polish notation.

Mapping program forms to Postfix:

a. Mathematical and boolean expressions

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

b.Unary operators

- -a → a –

c.Assignment

- a = a + b → a_i a b + =

d.Goto statements

- goto L1 → L1 jump_to

e.

Three Address Code

- Represented as TAC or 3AC
- is an intermediate code used by optimizing compilers to aid in the implementation of code-improving transformations.
- Each TAC instruction has **at most three addresses** [two operands and one for result]
- is typically a combination of assignment and a binary operator.

x:=y op z

Here x, y and z are names, constants or compiler generated temporaries. op is the operator.

- one calculation is composed of several smaller ones.
- In three-address code, this would be broken down into several separate instructions.

Example:

x+y*z

t_1 : y*z

t_2 : x+t_1

t_1 and t_2 are compiler generated temporary variable.

Three Address Code

- $x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)$

$t_1 := b * b$

$t_2 := 4 * a$

$t_3 := t_2 * c$

$t_4 := t_1 - t_3$

$t_5 := sqrt(t_4)$

$t_6 := - b$

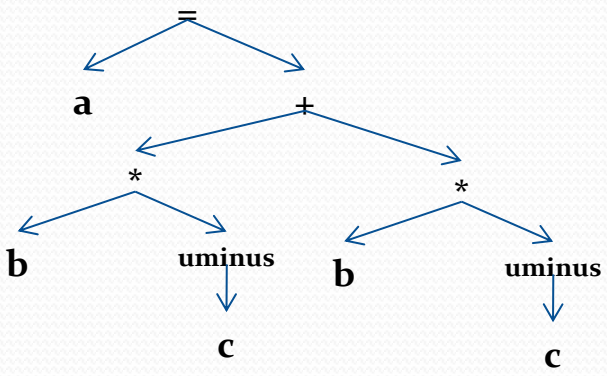
$t_7 := t_5 + t_6$

$t_8 := 2 * a$

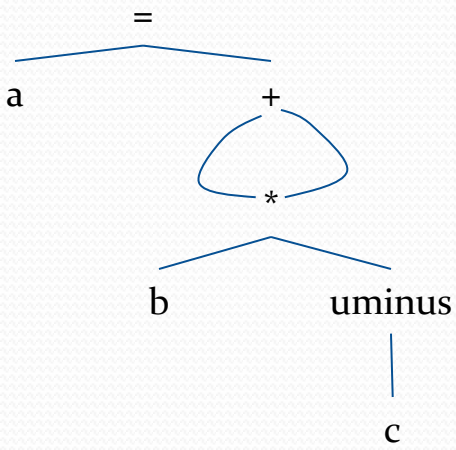
$t_9 := t_7 / t_8$

$x := t_9$

3AC corresponding to Syntax Tree & DAG



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



$t_1 := -c$
$t_2 := b * t_1$
$t_5 := t_2 + t_2$
$a := t_5$

Types of Three-Address statements

- a) Assignment Statement: x := y op z op is binary arithmetic or logical operator.
- b) Assignment Instruction : x := op y op is unary operation like unary minus, logical negation, shift operation etc.
- c) copy statement : x:=y
- d) Unconditional Jump : goto L
- e) Conditional Jump : if x relop y goto L. relop is the relational operator, L is the label.
- f) Indexed Assignments : x : y[i] or x[i]:=y
- g) Address and pointer assgnment: x:= &y, or x=*y

Implementation of Three Address Statements

- 3AC is an abstract form of intermediate code.
- These statements can be implemented as records with fields for the operator and operands.
- Such representations are quadruples, triples and indirect triples.

Quadruples:

Record structure with four fields (op, arg1, arg2 and result)

a := b * -c + b *-c

t1:= -c
t2:= b*t1
t3:=-c
t4:=b*t3
t5:=t2+t4
a:=t5

	op	arg1	arg2	result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Implementation of Three Address Statements

Triples

- temporary value by the position of the statement that computes it, avoids entering temporary names into the symbol table.
- three-address statement can be represented by records with only three fields: op, arg1 and arg2.
- The fields arg1 and arg2, for the arguments of op. arg1 and arg2 are either pointer to the symbol table (for programmers defined names or constants) or pointers into the triple structure (for temporary values).
- three fields are used, known as triples

a := b * -c + b *-c

t1:= -c
t2:= b*t1
t3:=-c
t4:=b*t3
t5:=t2+t4
a:=t5

	op	arg1	arg2
(0)-	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

x[i] :=y

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

x :=y[i]

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

Implementation of Three Address Statements

Indirect Triples:

Another implementation of three-address code that has been considered is that of listing pointers to triples, rather than listing of triples themselves.

This implementation is called indirect triples.

	state
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	**arg1**	**arg2**
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

3AC for relational and logical statement

a reop b	**t_1 =b** **t_2 =a** **t_3 = t_2 reop t_1**
reop may be ==, <, >, <=, >=, !=	
same is for logical operator like &&, || etc.	

3AC for Boolean Expression

- Boolean operator : or, and, not
- or, and are left-associative.
- or has lowest precedence, then and, then not.

Example:
a or b and not c

t_1 := not c
t_2 := a and t_1
t_3 := a or t_2

3AC for Boolean Expression

$E \rightarrow E_1$ or E_2

$E \rightarrow E_1$ and E_2

$E \rightarrow$ not E_1

$E \rightarrow E_1$ or E_2	***E_1.true := E.true*** ***E_1.false := newlabel*** ***E_2.true := E.true*** ***E_2.false := E.false*** ***E.code := E_1.code | | generate(E_1.false) | | E_2.code [Evaluate E_1.code .if it is false, then evaluate E_2.code]***
$E \rightarrow E_1$ and E_2	***E_1.true := newlabel*** ***E_1.false := E.false*** ***E_2.true := E.true*** ***E_2.false := E.false*** ***E.code := E_1.code | | generate(E_1.true) | | E_2.code [Evaluate E_1.code .if it is true, then evaluate E_2.code]***
$E \rightarrow$ not E_1	**E_1.true =E.false** **E_1.false =E.true** ***E.code := E_1.code*** **[Evaluate only *E_1.code*]**

3AC for if-then-else

if cond then	quadruples for cond, result in t1 if not t1 goto else label
statement	quadruples for then statement goto endif label:
else	else label:
else statements;	quadruples for else statements
end if	endif label:

Example:

```
if (a < (b + c))
    a = a - c;
```

3AC

$t_1 = c$
$t_2 = b + t_1$
$t_3 = a$
$t_4 = t_3 < t_2$
if $!t_4$ goto L1:
  $a = t_3 - t_1$
  $t_3 = a$
L1:

Example:

```
if (a < (b + c))
    a = a - c;
else
    a=a+c;
```

3AC

$t_1 = c$
$t_2 = b + t_1$
$t_3 = a$
$t_4 = t_3 < t_2$
if $!t_4$ goto L1: //else
  $a = t_3 - t_1$
  $t_3 = a$
goto L2: //Else omitted
L1:
  $t_3 = t_3 + t_1$
L2:

3AC for loop

For loop

```
for(i=0;i<n;i++)
{
a=b+1;
a=a*a;
}
```

```
        i=0;
L1:
        t1 =i<n;
        if(t1) goto L2;
        goto L3;
L4:     i++;
        goto L1;
L2:     t2=b;
        t2=t2+1
        a=t2 ;
        t3=a*a;
        a=t3 ;
        goto L4
L3:
```

do while

```
i=0;
do{
a=b+1;
a=a*a;
i++;
}while(i<n);
```

```
        i=0;
L1:
        t2=b;
        t2=t2+1
        a=t2 ;
        t3=a*a;
        a=t3 ;
        i++;
        t1 =i<n;
        if(t1) goto L1;
        goto L2;
L2:
```

While Loop

```
i=0;
while(i<n){
a=b+1;
a=a*a;
i++;
}
```

```
        i=0;
L1:
        t1 =i<n;
        if( t1 ) goto L2;
        goto L3;
L2:     t2=b;
        t2=t2+1
        a=t2 ;
        t3 =a*a;
        a= t3 ;
        i++;
        goto L1
L3:
```

Example

- while((A<C) and (B>D)) do
  if(A<1) then C=C+1
  else
  while (A<=D) do
  A=A+3

L_1:
t_1 = A<C
t_2 =B>D
t_3 = t_1 and t_2
if(t_3) goto L_2
goto L_3
L_2:
t_4=A<1
if !t_4 goto L_4
t_5 =C
t_5 =t_5+1
C=t_5
goto L_1
L_4 :
t_6 = A<=D
if(t_6) goto L_5
goto L_6
L_5:
t_7 =A
t_7 =t_7 +3
A = t_7
goto L_5:
L_6 :
goto L_1
L_3:

Assignment

- Construct 3AC for the following.

a)

```
if(a<b)
  c=a+b
else if(a==b)
  c=a*b;
else
  c=a-b;
```

b) if (x+2) > 3 * (y- 1)+4 then z := 0;

3AC for block of codes - Example

```
x:=1;
y:=x+10;
while (x<y) {
    x:=x+1;
    if (x%2==1) then y:=y+1;
    else y:=y-2;
}
```

➔

```
01: mov   1,,x
02: add   x,10,t1
03: mov   t1,,y
04: lt    x,y,t2
05: jmpf  t2,,17
06: add   x,1,t3
07: mov   t3,,x
08: mod   x,2,t4
09: eq    t4,1,t5
10: jmpf  t5,,14
11: add   y,1,t6
12: mov   t6,,y
13: jmp   ,,16
14: sub   y,2,t7
15: mov   t7,,y
16: jmp   ,,4
17:
```

Assignment

Find the 3AC for the following block of codes

a)

```
while a<b do
    if c<d then
        x:=y+z
    else
        x=y-z
```

Backpatching

- Generating SDD in Boolean expression needs jump to some label.
- Label can not be specified in a single pass---two passes are required.
- Construct a syntax tree for input, then traverse the tree in depth first order, to compute the translation given to the definition.
- In first pass---leave the label of jump unspecified with 'goto' statement.
- label is filled in second pass---traversing tree in depth first order.
- This subsequent filling of labels is known as backpatching .

Backpatching

backpatching can be used to generate code for Boolean expression and flow of control in one pass

Quadruples are stored into a quadruple array.

Labels are indices into the array.

Three functions are used to manipulate the lists of labels.

a) makelist(i) : creates a new list with a single entry i---that is an index into the array of quadruples.

b) mergelist($list_1$, $list_2$): Returns a new list pointer p by concatenating lists pointed by $list_1$ and $list_2$.

c) backpatch(list, target): inserts 'target' as the target label for each of the statements on the list pointed to by p.

Backpatching-example

Production	SDD
$E \rightarrow E_1$ or ME_2	{backpatch(E_1.falselist, M.quad); E.truelist=merge(E_1.truelist, E_2.truelist); E.falselist=E_2.falselist}δ_1
| E_1 and ME_2	{backpatch(E_1.truelist, M.quad); E.truelist=E_2 .truelist; E.falselist=merge(E_1.falselist, E_2.falselist)}δ_2
| not E_1	{E.truelist=E_1 .falselist; E.falselist=E_1 .truelist;}
| (E_1)	{E.truelist=E_1 .truelist; E.falselist=E_1 .falselist;}
| id_1 relop id_2	{E.truelist=makelist(nextquad); E.falselist=makelist(nextquad+1) emit('if' **id_1.place relop.op id_2.place** 'goto _') emit('goto__')} δ_3
| true	{E.truelist=makelist(nextquad); emit('goto__')}
| false	{E.falselist=makelist(nextquad); emit('goto__')}
$M \rightarrow \epsilon$	{M.quad=nextquad} δ_4

M is a dummy non-terminal with attribute M.quad, which can hold the address of a quadruple.

- δ1: If E_1 is false then we need to search for E_2.code. The M.quad [Number of the first statement for quadraple for E_2] is inserted in backpatch. E is true if any of E_1 or E_2 is true.

- δ2: If E_1 is true then we need to search for E_2.code. The M.quad [Number of the first statement for quadraple for E_2] is inserted in backpatch. E is true if both E_1 and E_2 are true. But E_2 is evaluated if and only E_1 is true. Thus E.true= E_2.true. But E is false if any of E_1 or E_2 is false.

- δ3:If E_1 is true makelist strote the index of the beginning of the quadraple. If E_1 is false, makelist stores the index of the next statement. It also emits (creates) two(one for true and another for false) goto statements with label as blank.

- δ4: The variable nextquad holds the index of the next quadruple to follow. This value will be backpatched into E_1.truelist and E_1.falselist when required.

Backpatching-Example

- a<b or c<d and e<f

a) Construction of quadruple for statements with relational operator. [E →id_1 relop id_2]

i) a<b

100: if a<b goto ___

101: goto ____

ii) c<d

102: if c<d goto ___

103: goto ___

iii) e<f

104: if e<f goto ___

105: goto ___

Backpatching-Example

b) Construction of quadruples for Boolean expression [E → E_1 and ME_2]

i) c<d and e<f

102: if c<d goto 104 [Semantic action is backpatch(E_1.truelist, M.quad). Here it is backpatch({102}, 104)]

103: goto __

104: if e<f goto __

105: goto __

ii) a<b or c<d and e<f

100:if a<b goto __

101:goto 102 [Semantic action is backpatch(E_1.falselist, M.quad). Here it is backpatch({101}, 102)]

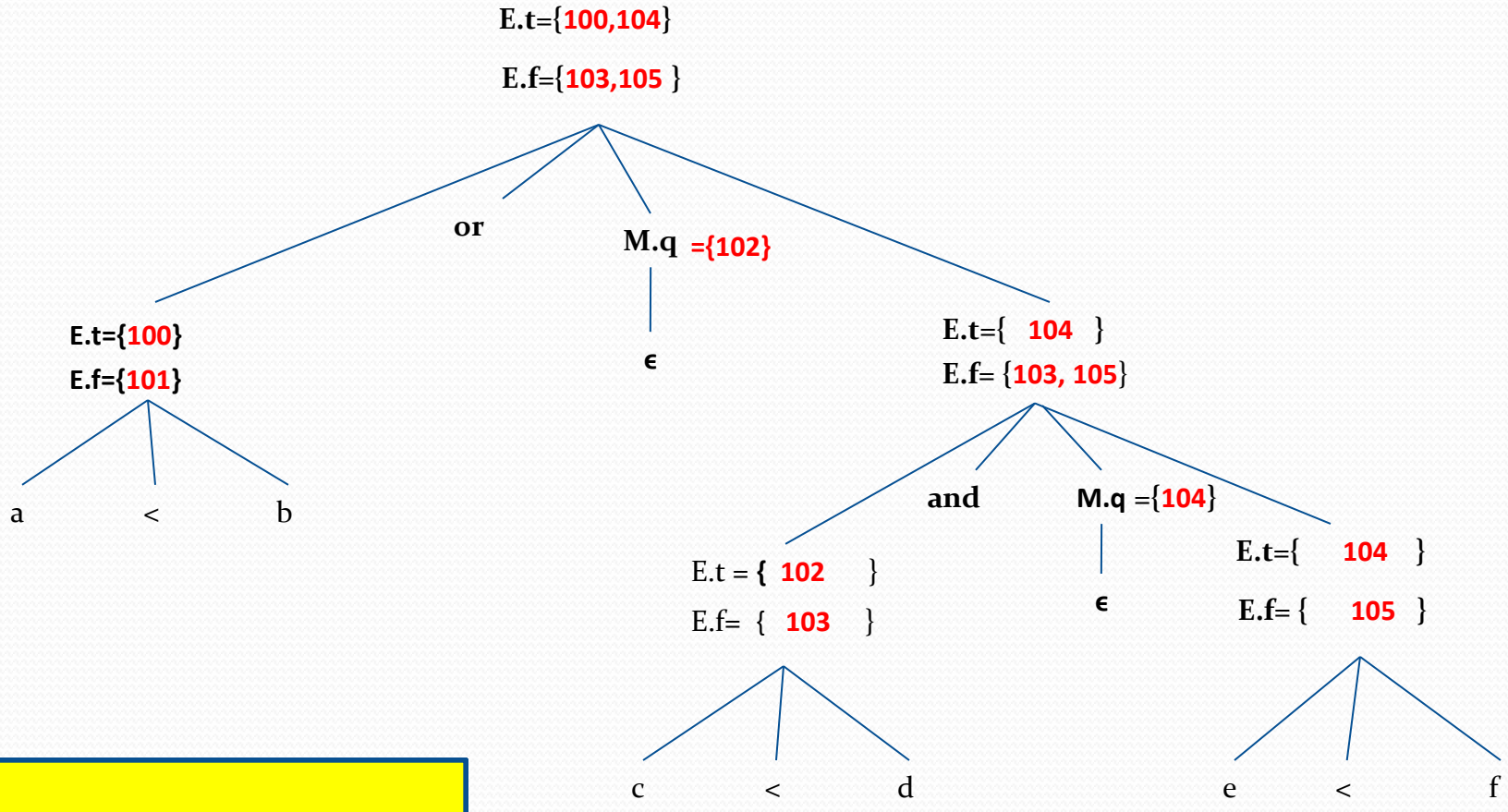
102: if c<d goto 104

103: goto __

104: if e<f goto __

105: goto __

Backpatching-Example



M.quad=nextquad

References

- Alfred V. *Aho*, Ravi. *Sethi*, Jeffrey D. *Ullman*,**Compilers: Principles, Techniques, and Tools**
- RobinAnil "Three Address Code Generation for Control Statements"
- Maggie Johnson, "**Three Address Code Examples**", August 1, 2012