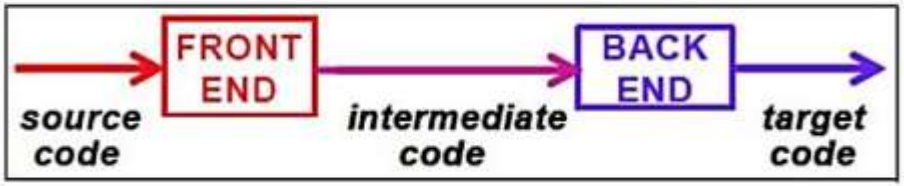
Intermediate Code Generation

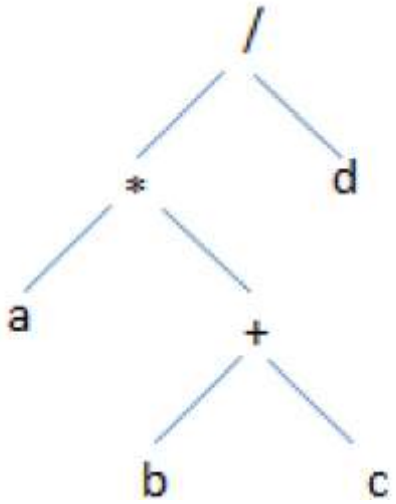
- In compiler, the front-end translates a source program into an intermediate representation from which the back end generates target code.
- Why is intermediate code used ?
  - Source Target code generation n optimizers and n code generators
  - Intermediate code 1 optimizer
- Intermediate Representation
  - ***Syntax Tree (parse tree)***
  - Postfix Notation
  - Three Address Code



Intermediate Representation
1. Syntax Tree/Abstract Syntax Tree (AST)

- 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.

a*(b+c)/d



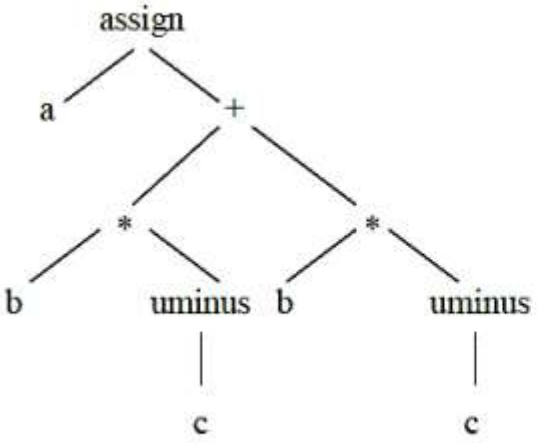
Parse Tree Vs Syntax Tree

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

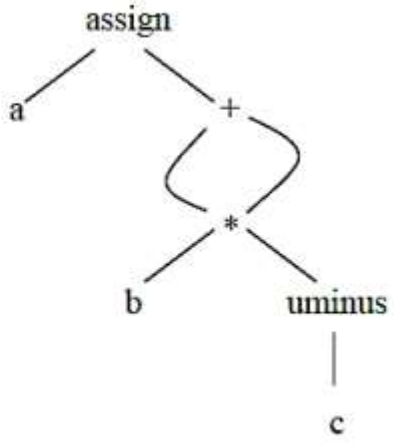
Directed Acyclic Graph (DAG)

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

a=b*-c + b*-c



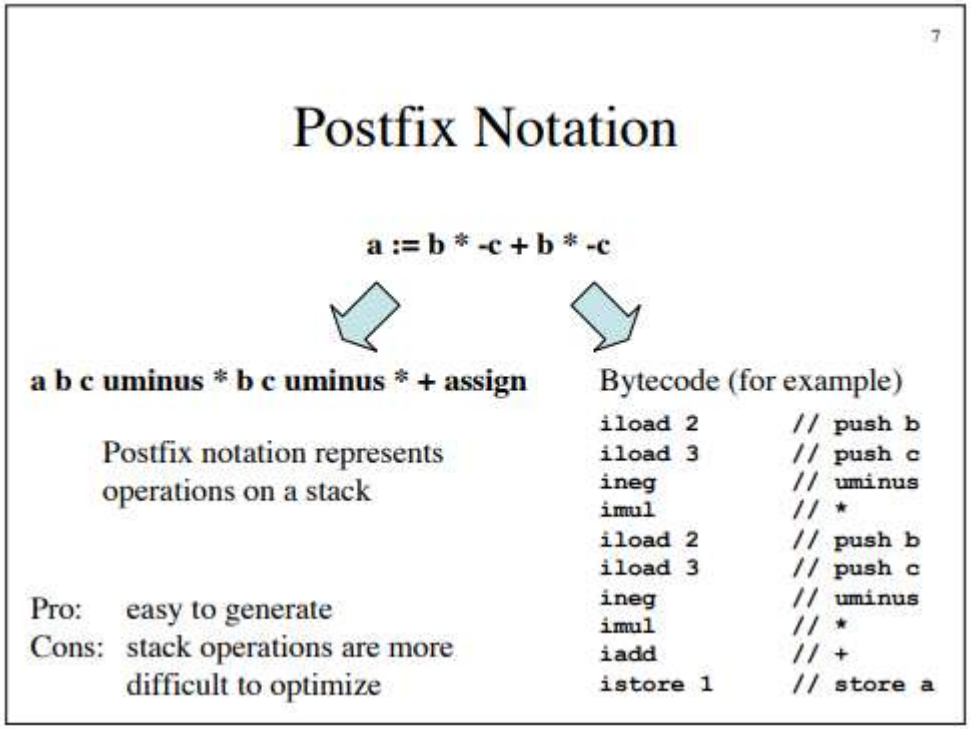
(a) Syntax tree



(b) Dag

Intermediate Representation
2. *Postfix Notation*

- Infix Notation a+b
- Postfix Notation ab+
- Ex: (a+b)*(c+d)+(a-b) ab+cd+*ab-+
- Ex2: a=b*-c + b*-c
- Postfix notation: abc-*bc-*=



Intermediate Representation
3. *Three Address Code*

- <3 references – 3 Address Statement
- ***Example1: a+b*c+d***
- t1=b*c
- t2=a+t1
- t3=t2+d
- ***Example2: a*-(b+c)***
- t1=b+c
- t2=uminus t1
- t3=a*t2

Intermediate Representation ***Three Address Code***

- 3 representation of Three Address Code
  - Quadruple
    - 4 fileds (op,arg1,arg2,res)
  - Triple
    - 3 fields (op,arg1,arg2)
  - Indirect Triples

a = b * – c + b * – c

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

Indirect Triples – pointer (references)

a = b * – c + b * – c

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

List of pointers to table

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

Indirect Triples representation

Declaration and Assignment

- Assignment statements: x `:=` y *op* z, x `:=` *op* y
- Indexed assignments: x `:=` $y[i]$, $x[i]$ `:=` y
- Pointer assignments: x `:= &`y, x `:= *`y, `*`x `:=` y
- Copy statements: x `:=` y
- Unconditional jumps: **`goto`** *lab*
- Conditional jumps: **`if`** x *relop* y **`goto`** *lab*
- Function calls: **`param`** x... **`call`** p, n
  `return` y

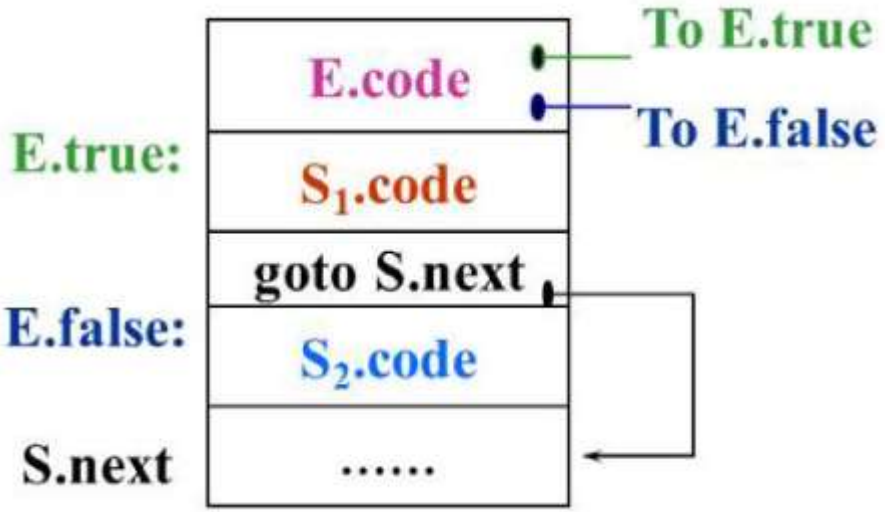
Intermediate Code generation for *Boolean Expressions*

- Boolean Expression
  - Logical values
  - Conditional Expression – change the flow of program (if-else, do-while)
- Boolean operator
  - And
  - Or (lowest precedence)
  - Not
- Example
  - $E \rightarrow E$ or E
  - $E \rightarrow E$ and E
  - $E \rightarrow$ not E
  - $E \rightarrow (E)$
  - $E \rightarrow$ id relop id
  - $E \rightarrow$ TRUE $E \rightarrow$ id
  - $E \rightarrow$ FALSE

Intermediate Code generation for ***Boolean Expressions***

- Numerical representation of Boolean Expression
  - Example1: A or B and C
    - Three Address Sequence:
    - T1=B and C
    - T2=A or T1
  - Example2: A<B if A<B then 1 else 0
    - Three Address Sequence:
    - 1. If A<B goto (4)
    - 2. T=0
    - 3. goto (5)
    - 4. t=1
    - 5. ---

Attributes used for "if E then S1 else S2"



Intermediate Code Generation for Switch Statements

```
        code to evaluate E into t
        goto test
L1:     code for S1
        goto next
L2:     code for S2
        goto next
        ...
Ln-1:   code for Sn-1
        goto next
Ln:     code for Sn
        goto next
test:   if t = V1 goto L1
        if t = V2 goto L2
        ...
        if t = Vn-1 goto Ln-1
        goto Ln
next:
```

Translation of a switch-statement

- switch E
- begin
- case V1: S1
- case V2: S2
- …
- case Vn-1: Sn-1
- default: Sn
- end
- Switch Statement □ temporary t , two new labels test and next are generated
- Each case statement □ new label is created and entered into Symbol Table

Intermediate Code Generation for Procedure Call

- Actions taken during Calling Sequence
  - Procedure call – Activation record - space allocation
  - Evaluate the argument of called procedure
  - Save the State of Calling procedure
  - Save the return address
  - Generate Jump to the beginning of code
  - *Example:*
    - (1) S→ call id(Elist)
    - (2) Elist → Elist,E
    - (3) Elist → E

BACKPATCHING

- Easy way to implement syntax-directed definition of Boolean Expression
- Boolean Expression – Single pass – cannot predict the labels where the control will jump
- Backpatching – address instead of label is used
- Three operations:
  - Makelist(i) – list with I which points to qudraple
  - Merge(i,j) – concatenate i list with j
  - Backpatch(p,i) – inserts i as target label for each of the statement pointed by p

BACKPATCHING

- Process of backpatching
  - A marker Non-terminal M – next instruction to be executed
  - Example
    - E□ E1 and M E2
    - Incomplete jumps with unfilled labels □ E.truelist and E.falselist
    - E1 – false , E is also false □ E1.falselist becomes a part of E.flaselist
    - E1 – true □ E2 test □ E1.truelist becomes the beginning code for E2 □ marker non-terminal M