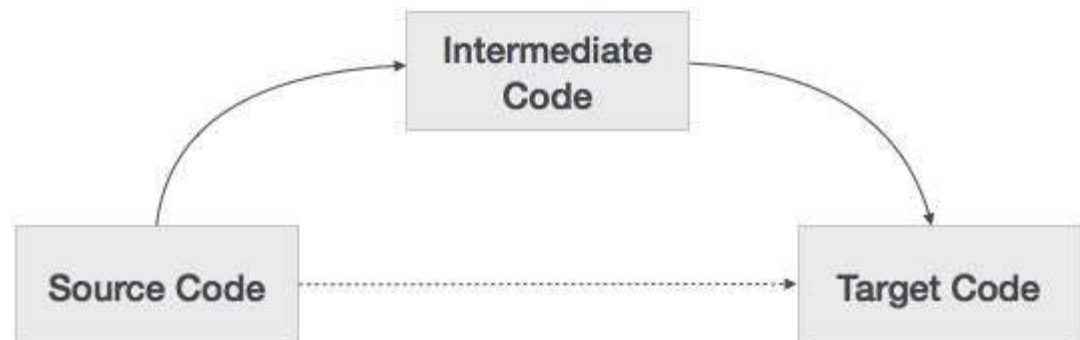




# Intermediate Code Generation

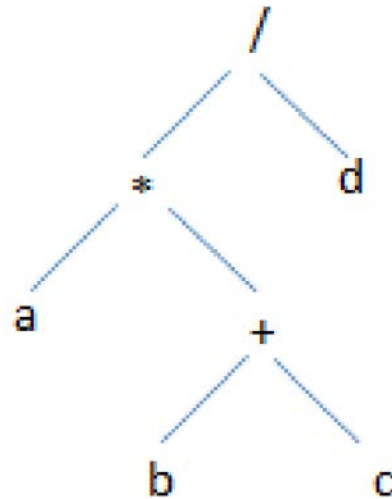
- Why is intermediate code used ?
  - Source  $\rightarrow$  Target code generation  $\rightarrow$  n optimizers and n code generators
  - Intermediate code  $\rightarrow$  1 optimizer
- Intermediate Representation
  - ***Syntax Tree (parse tree)***
  - Postfix Notation
  - Three Address Code





# Intermediate Representation Syntax Tree

$a*(b+c)/d$





# Intermediate Representation


## *Postfix Notation*

- Infix Notation  $\square a+b$
- Postfix Notation  $\square ab+$
- Ex:  $(a+b)*(c+d)+(a-b) \square ab+cd+*ab-+$

7

### Postfix Notation

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



<p><b>a b c uminus * b c uminus * + assign</b></p> <p>Postfix notation represents operations on a stack</p> <p>Pro: easy to generate</p> <p>Cons: stack operations are more difficult to optimize</p>	<p>Bytecode (for example)</p> <pre>iload 2 // push b iload 3 // push c ineg // uminus imul // * iload 2 // push b iload 3 // push c ineg // uminus imul // * iadd // + istore 1 // store a</pre>
---	--



# Intermediate Representation

## *Three Address Code*

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



# Intermediate Representation *Three Address Code*



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

#	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)

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



# Indirect Triples – pointer (references)

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

List of pointers to table

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

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

## Indirect Triples representation



# Declaration and Assignment

- Assignment statements:  $x := y \text{ op } z$ ,  $x := \text{op } y$
- 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
  - $E \rightarrow E \text{ or } E$
  - $E \rightarrow E \text{ and } E$
  - $E \rightarrow \text{not } E$
  - $E \rightarrow (E)$
  - $E \rightarrow \text{id relop id}$
  - $E \rightarrow \text{TRUE } E \rightarrow \text{id}$
  - $E \rightarrow \text{FALSE}$





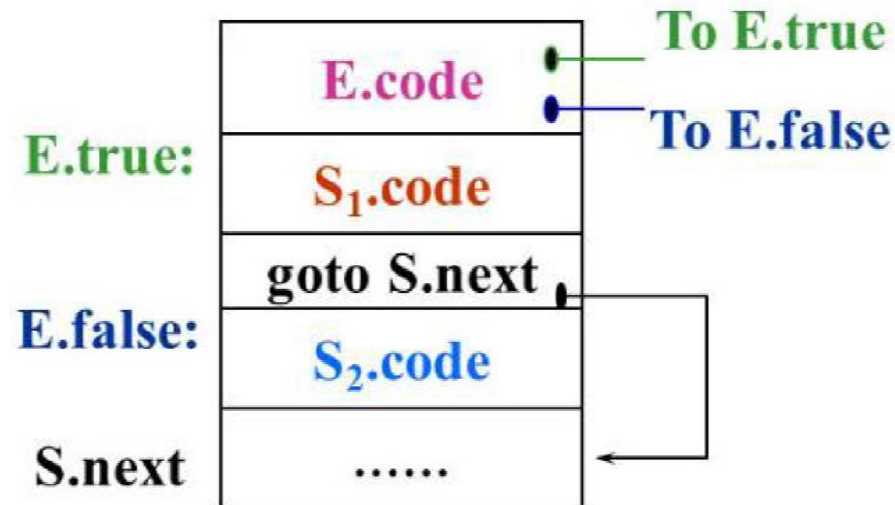
# Intermediate Code generation for *Boolean Expressions*



- 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

- switch  $E$
- begin
- case  $V_1$ :  $S_1$
- case  $V_2$ :  $S_2$
- ...
- case  $V_{n-1}$ :  $S_{n-1}$
- default:  $S_n$
- 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

```
code to evaluate  $E$  into  $t$ 
goto test
L1: code for  $S_1$ 
      goto next
L2: code for  $S_2$ 
      goto next
...
L $n-1$ : code for  $S_{n-1}$ 
         goto next
L $n$ : code for  $S_n$ 
        goto next
test:  if  $t = V_1$  goto L1
        if  $t = V_2$  goto L2
        ...
        if  $t = V_{n-1}$  goto L $n-1$ 
        goto L $n$ 
next:
```

Translation of a switch-statement



# 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)  $S \rightarrow \text{call id}(\text{Elist})$
    - (2)  $\text{Elist} \rightarrow \text{Elist}, \text{E}$
    - (3)  $\text{Elist} \rightarrow \text{E}$



# 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 quadruple
  - 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
    - $E \rightarrow E1$  and M  $E2$
    - Incomplete jumps with unfilled labels  $\rightarrow E.true\text{list}$  and  $E.false\text{list}$
    - $E1$  – false ,  $E$  is also false  $\rightarrow E1.false\text{list}$  becomes a part of  $E.false\text{list}$
    - $E1$  – true  $\rightarrow E2$  test  $\rightarrow E1.true\text{list}$  becomes the beginning code for  $E2$   $\rightarrow$  marker non-terminal M