# Intermediate Code Generation

su hoi 6e .... machine independent codes

- *Intermediate codes* are machine independent codes, but they are close to machine instructions. but they are very close to the machine codes
- The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.

is retargeting supported? like we can convert to some different target language from here.

- Retargeting is supported and design is simplified
- Machine independent optimizers can be applied

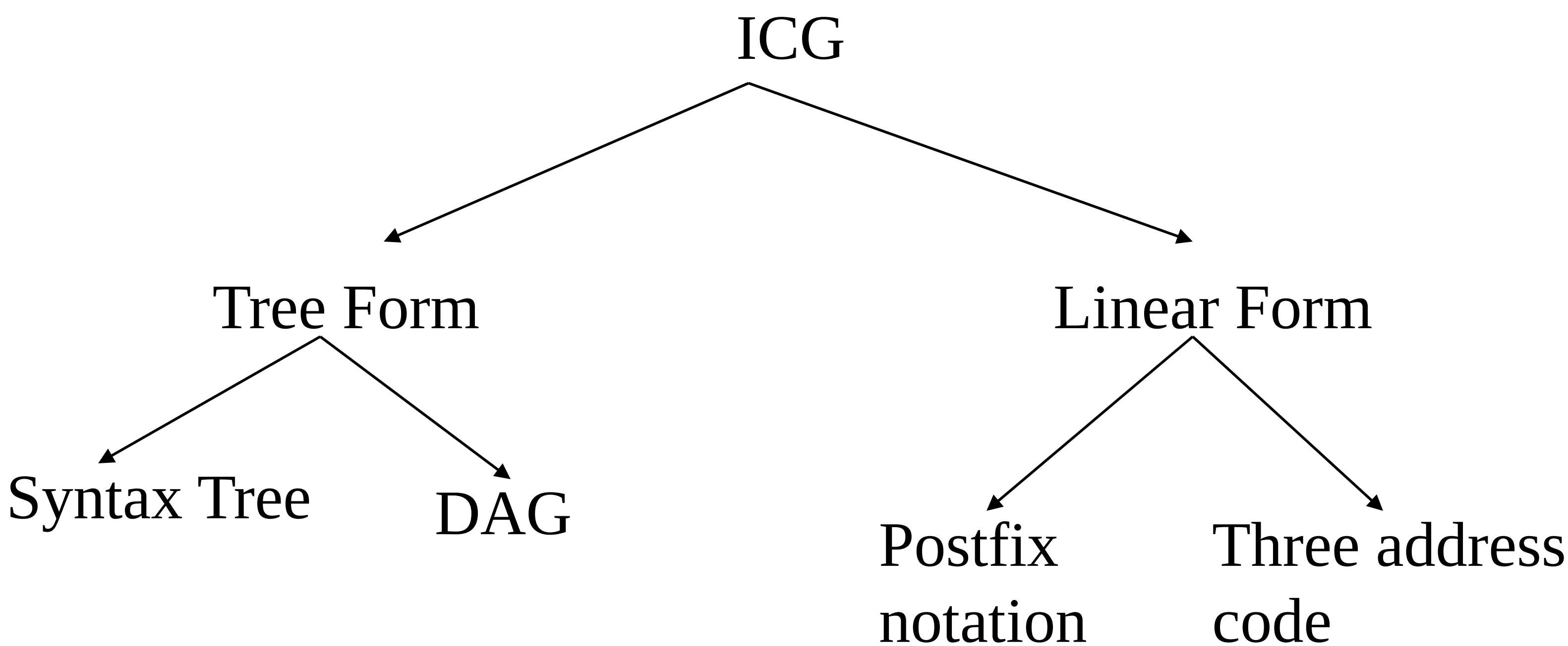
then it would be complex na ..no the design is simplified .only!

- Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language.

bahu j badhi jat ni intermediate languages hoi 6e...and hte design of the compiler decide kare 6e ke kaya type ni intermediate language tamne joise.

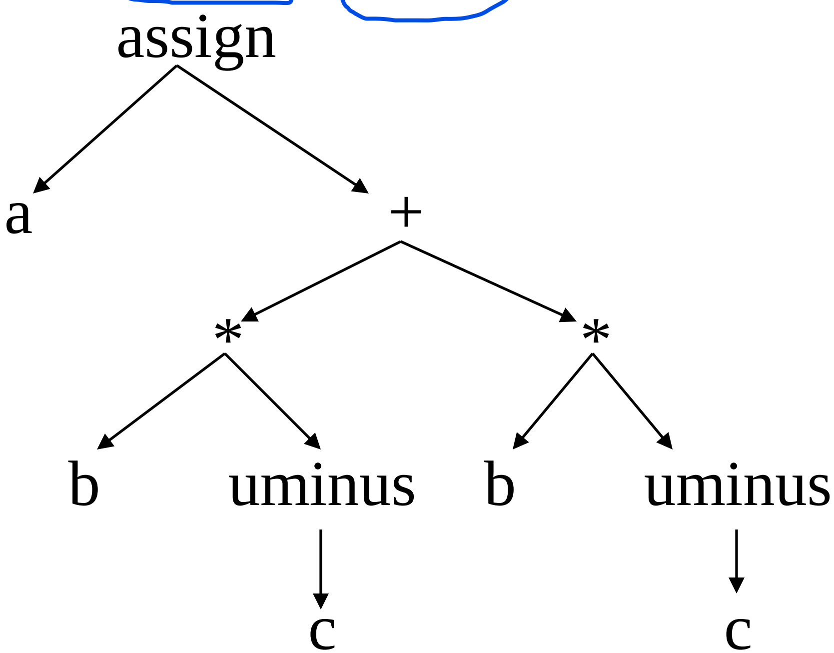
souce language ---------------------------------------> intermediate language
(intermediate code generator)

how ? = we applly the machine independent optimizers

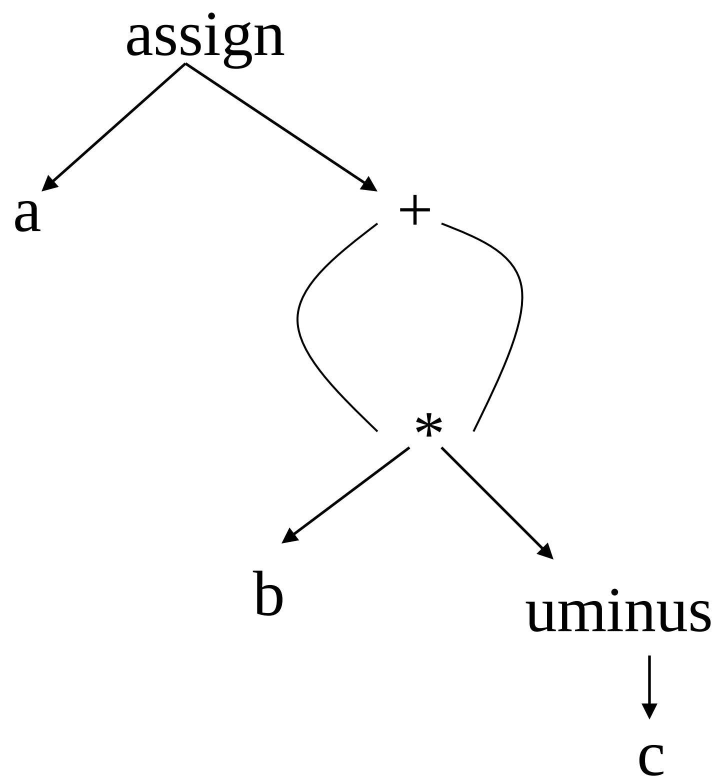

ICG
Tree Form
Linear Form
Syntax Tree
DAG
Postfix notation
Three address code

# Syntax Tree and DAG

- Syntax Tree : it depicts the natural hierarchical structure of a source program
- DAG: it gives same information in compact way because common sub expressions are identified here.
- For a:=b * -c + b * -c




Syntax Tree




DAG

# Syntax tree representation

| | | | |
|---|---|---|---|
| 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 | assign | 9 | 8 |

All nodes in the syntax tree can be visited by following pointers, starting from the root at position 10

how to create the syntax tree and dag?

Syntax tree and DAG can be created using SDT

# Types of Three-Address Statements

***Binary Operator:*** `op y,z,result` or `result := y op z`

where `op` is a binary arithmetic or logical operator. This binary operator is applied to `y` and `z`, and the result of the operation is stored in `result`.

Ex:
```
add  a,b,c
gt   a,b,c
addr a,b,c
addi a,b,c
```

***Unary Operator:*** `op y,,result` or `result := op y`

where `op` is a unary arithmetic or logical operator. This unary operator is applied to `y`, and the result of the operation is stored in `result`.

Ex:
```
uminus    a,,c
not       a,,c
inttoreal a,,c
```

# Three-Address Statements (cont.)

***Move Operator:*** `mov y,,result` or `result := y`

where the content of `y` is copied into `result`.

Ex:
```
mov   a,,c
movi  a,,c
movr  a,,c
```

***Unconditional Jumps:*** `jmp ,,L` or `goto L`

It will jump to the three-address code with the label `L`, and the execution continues from that statement.

Ex:
```
jmp  ,,L1     // jump to L1
jmp  ,,7      // jump to the statement 7
```

# Three-Address Statements (cont.)

***Conditional Jumps:*** `jmp`***`relop`***` y,z,L` or `if y `***`relop`***` z goto L`

It will jump to the three-address code with the label `L` if the result of `y relop z` is true, and the execution continues from that statement. If the result is false, the execution continues from the statement following this conditional jump statement.

Ex:

```
jmpgt   y,z,L1    // jump to L1 if y>z
jmpgte  y,z,L1    // jump to L1 if y>=z
jmpe    y,z,L1    // jump to L1 if y==z
jmpne   y,z,L1    // jump to L1 if y!=z
```

Our relational operator can also be a unary operator.

```
jmpnz   y,,L1     // jump to L1 if y is not zero
jmpz    y,,L1     // jump to L1 if y is zero
jmpt    y,,L1     // jump to L1 if y is true
jmpf    y,,L1     // jump to L1 if y is false
```

# Three-Address Statements (cont.)

***Procedure Parameters:*** `param x,,` or `param x`

***Procedure Calls:*** `call p,n,` or `call p,n`

where `x` is an actual parameter, we invoke the procedure `p` with `n` parameters.

Ex:

param $x_1$,,
param $x_2$,,
→ $p(x_1, \ldots, x_n)$
param $x_n$,,
call p,n,

f(x+1,y) →

```
add   x,1,t1
param t1,,
param y,,
call  f,2,
```

# Three-Address Statements (cont.)

***Indexed Assignments:***

`move y[i],,x` or `x := y[i]`

`move x,,y[i]` or `y[i] := x`

***Address and Pointer Assignments:***

`moveaddr y,,x` or `x := &y`

`movecont y,,x` or `x := *y`

# Syntax-Directed Translation into Three-Address Code

| | |
|---|---|
| $S \rightarrow \textbf{id} := E$ | S.code = E.code \|\| gen('mov' E.place ',,' id.place) |
| $E \rightarrow E_1 + E_2$ | E.place = newtemp(); |
| | E.code = $E_1$.code \|\| $E_2$.code \|\| gen('add' $E_1$.place ',' $E_2$.place ',' E.place) |
| $E \rightarrow E_1 * E_2$ | E.place = newtemp(); |
| | E.code = $E_1$.code \|\| $E_2$.code \|\| gen('mult' $E_1$.place ',' $E_2$.place ',' E.place) |
| $E \rightarrow - E_1$ | E.place = newtemp(); |
| | E.code = $E_1$.code \|\| gen('uminus' $E_1$.place ',,' E.place) |
| $E \rightarrow ( E_1 )$ | E.place = $E_1$.place; |
| | E.code = $E_1$.code |
| $E \rightarrow \textbf{id}$ | E.place = **id**.place; |
| | E.code = null |

- E.place is the name that will hold the value of E
- E.code is the sequence of three-address statements evaluating E
- || - merging/concatenation

# Syntax-Directed Translation (cont.)

$S \rightarrow$ while E do $S_1$

S.begin = newlabel();
S.after = newlabel();
S.code = gen(S.begin ":") || E.code ||
gen('jmpf' E.place ',,' S.after) || $S_1$.code ||
gen('jmp' ',,' S.begin) ||
gen(S.after ':")

$S \rightarrow$ if E then $S_1$ else $S_2$

S.else = newlabel();
S.after = newlabel();
S.code = E.code ||
gen('jmpf' E.place ',,' S.else) || $S_1$.code ||
gen('jmp' ',,' S.after) ||
gen(S.else ':") || $S_2$.code ||
gen(S.after ':")

# Implementation of Three-Address Statements

- Each line of code contains one operator and up to three operands, represented as addresses
- Closer to the machine/targeted code than parse tree representation
- No of variants…quadraples, triples, indirect triples,….

# Quadraples

- Consist of an operation, up to two operands and a result
- A+B would be translated into quads as:

+ A B T1

operation operands result

| | Operator | Op1 | Op2 | Result |
|---|---|---|---|---|
| (1) | + | a | b | t1 |
| (2) | - | c | | t2 |
| (3) | * | t1 | t2 | t3 |
| (4) | / | t3 | d | t4 |
| (5) | = | t4 | | x |

Adv: statements can be moved around

Dis: too much space wasted for temp

Quadraples representation of x = (a+b)* (-c)/d

# Triples

•Don't use an extra temporary variable like quadraples rather pointer is used to reference

| | Operator | Op1 | Op2 |
|---|---|---|---|
| (1) | + | a | b |
| (2) | - | c | |
| (3) | * | (1) | (2) |
| (4) | / | (3) | d |
| (5) | = | x | (4) |

Adv: No wastage of space

Dis: statements can't be moved around

# Indirect Triple

- Uses an addition array to list the pointers to the triples in the desired order

je order ma joita hoi te order ma karine apis.

| |
|---|
| (1) |
| (2) |
| (3) |
| (4) |
| (5) |

| | Operator | Op1 | Op2 |
|---|---|---|---|
| (1) | + | a | b |
| (2) | - | c | |
| (3) | * | (1) | (2) |
| (4) | / | (3) | d |
| (5) | = | x | (4) |