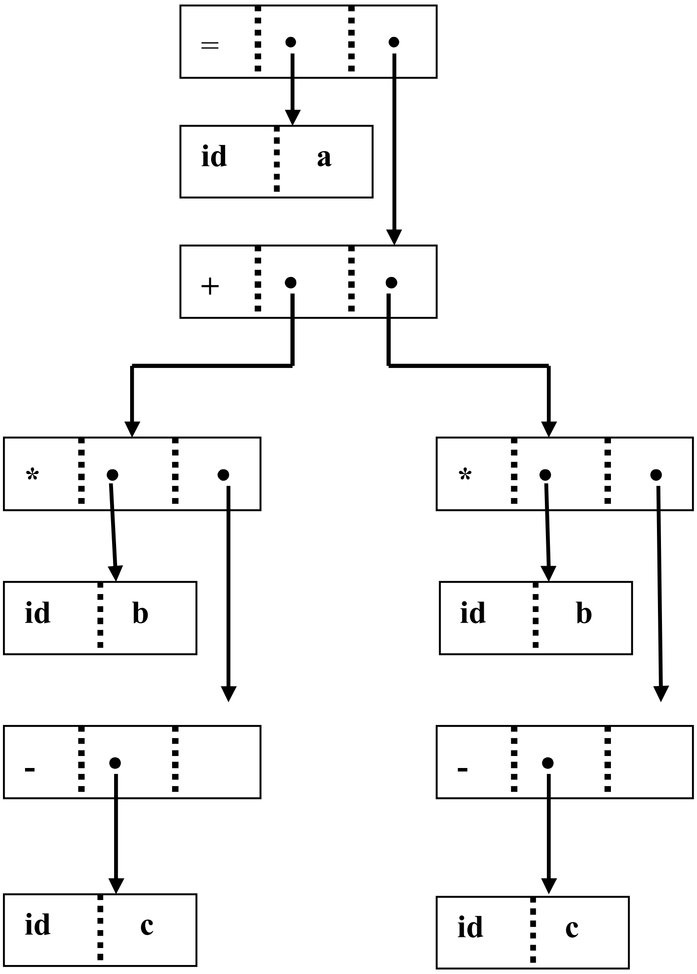


1

0	id	b	
1	id	c	
2	-	1	
3	*	0	2
4	id	b	
5	id	c	
6	-	5	
7	*	4	6
8	+	3	7
9	id	a	
10	=	9	8
	….	…..	…..
	…..	…..	…..

2

2. Three-Address Code is a sequence of statements of the general form :

X=Y op Z // op is binary arithmetic operation

For example : *x + y * z*

*t1 = y * z*
t2 = x + t1

where t1 ,t2 are compiler generated temporary.

Types of three address code statement:-

1. Assignment statements of the form *X=Y op Z* (where op is a binary arithmetic or logical operator).
2. Assignment instructions of the form *X= op Y* (op is a unary operator).
3. Copy statements of the form *X=Y* .
4. Unconditional jump (*Goto L*).
5. Conditional jump (*if X relop Y goto L*).
6. *Param X* & *Call P,N* for procedure call and and return *Y* , for example :

```
Param      x1
Param      x2
……..
Param      xn
Call       P,n
```

7. Index assignments of the form X=Y[i] & X[i]=Y.
8. Address & Pointer Assignments

X= &Y
*X= * Y*
**X= Y*

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

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

Three address code
For syntax tree

```
t1 = - c
t2 = b * t1
t5 = t2 + t2
a = t5
```

Three address code
For DAG

Note: Three-address statements are a kin to assembly code statements can have symbolic labels and there are statements for flow of control.

Implementation of Three Address Code :-

In compiler , three-address code can be implement as records, with fields for operator and operands.

1. **Quadruples :-** It is a record structure with four fields:
   - **OP** // operator
   - **arg1 , arg2** // operands
   - **result**

2. **Triples :-** To avoid entering temporary into *ST* , we might refer to a temporary value by position of the statement that compute it . So three address can be represent by record with only three fields:

   - **OP** // operator
   - **arg1 , arg2** // operands

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

i. By Quadruples

Position	OP	arg1	arg2	result
0	-	c		t1
1	*	b	t1	t2
2	-	c		t3
3	*	b	t3	t4
4	+	t2	t4	t5
5	=	t5		a

ii. By Triples

Position	OP	arg1	arg2
0	-	c	
1	*	b	(0)
2	-	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Code Optimization

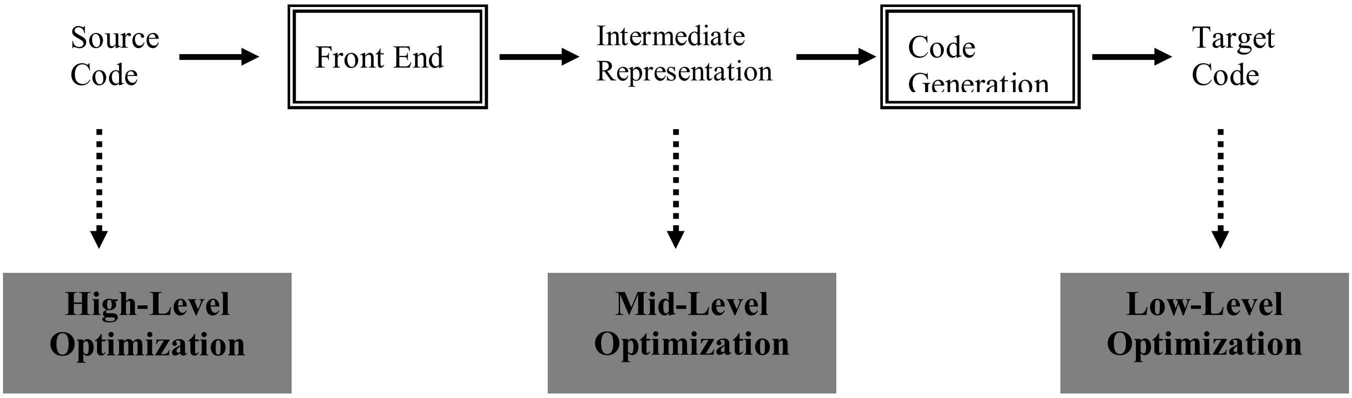
Compilers should produce target code that is as good as can be written by hand. This goal is achieved by program transformations that are called " Optimization " . Compilers that apply code improving transformations are called " Optimizing Compilers ".

Code optimization attempts to increase program efficiency by restructuring code to simplify instruction sequences and take advantage of machine specific features:-

- Run Faster , or
- Less Space , or
- Both (Run Faster & Less Space).

The transformations that are provided by an optimizing compiler should have several properties:-

1. A transformation must preserve the meaning of program. That is , an optimizer must not change the output produce by program for an given input, such as **division by zero.**
2. A transformation must speed up programs by a measurable amount.



Places for Optimization

This lecture concentrates on the transformation of intermediate code (Mid-Optimization or Independent Optimization),this optimization using the following organization:-