RAJIV GANDHI PROUDYOGIKI VISHWAVIDYALAYA, BHOPAL New Scheme
Based On AICTE Flexible Curricula Computer Science & Engineering, VI-Semester
CS-603(C): Compiler Design

Topics Covered :-

Unit -IV

Code Generation Intermediate code generation: Declarations, Assignment statements, Boolean expressions, Case statements, Back patching, Procedure calls Code Generation: Issues in the design of code generator, Basic block and flow graphs, Register allocation and assignment, DAG representation of basic blocks, peephole optimization, generating code from DAG.

Overview of Intermediate Code Generation

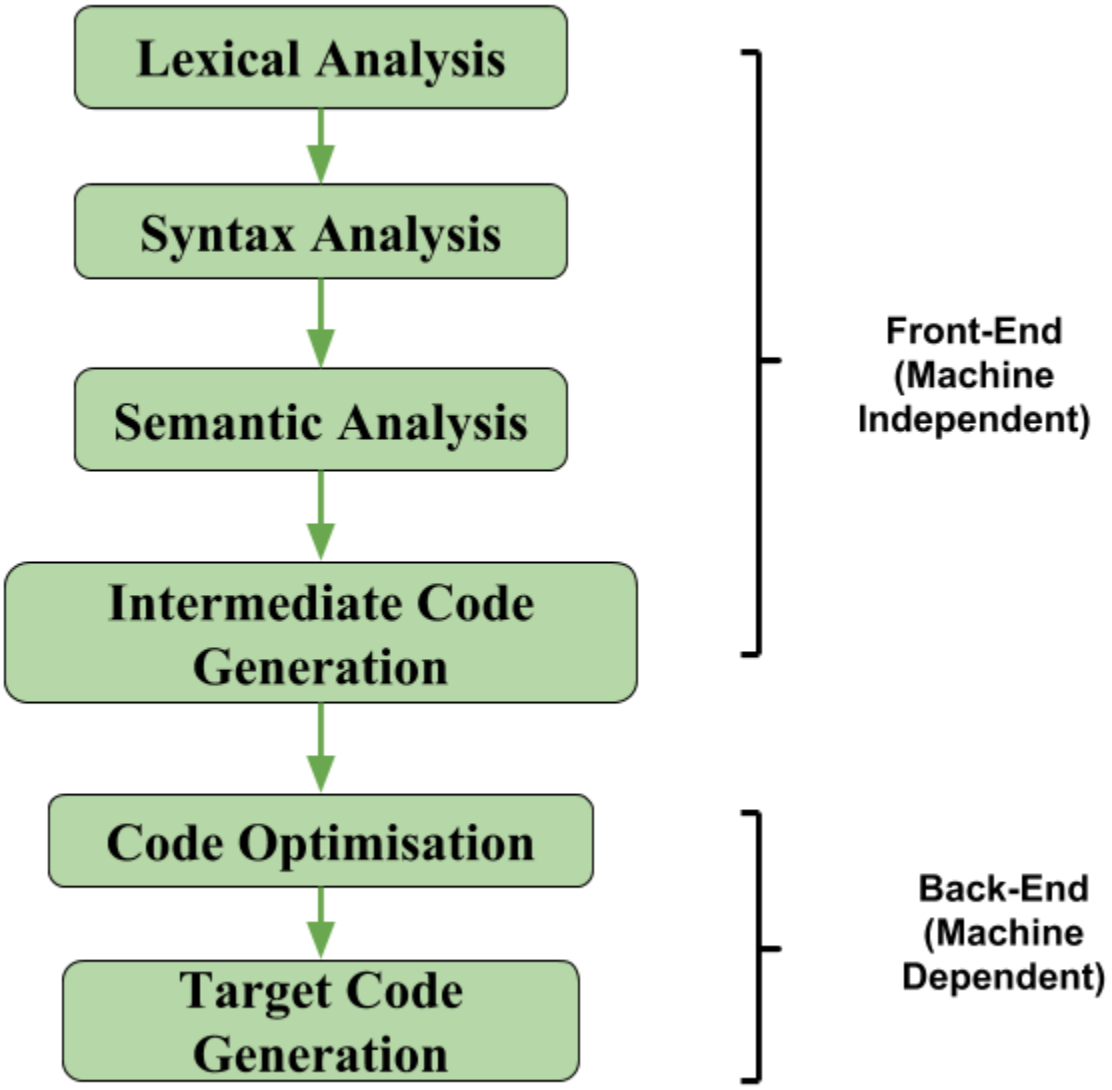
In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine).

The benefits of using machine independent intermediate code are:

- Because of the machine independent intermediate code, portability will be enhanced. For ex, suppose, if a compiler translates the source language to its target machine

language without having the option for generating intermediate code, then for each new machine, a full native compiler is required. Because, obviously, there were some modifications in the compiler itself according to the machine specifications.

- It is easier to apply source code modification to improve the performance of source code by optimising the intermediate code.



Intermediate Code Generation

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

- The designer of the compiler decides the intermediate language.
- Syntax trees can be used as an intermediate language.

- Postfix notations, 3 address code (quadruples) can be used as an intermediate language.

Three address code

The input is received by the predecessor phase, semantic analyzer, in the form of annotated syntax tree by the Intermediate code generator. The syntax tree is then converted into a linear representation. The intermediate code is machine independent code and hence it is assumed by the code generator for having unlimited number of memory storage for generating the code.

Types of Three address code

There are different types of statements in source program to which three address code has to be generated. Along with operands and operators, three address code also use labels to provide flow of control for statements like if-then-else, for and while. The different types of three address code statements are:

1. Assignment statement

a = b op c
In the above case b and c are operands, while op is binary or logical operator. The result of applying op on b and c is stored in a.

2. Unary operation

a = op b This is used for unary minus or logical negation.

Example: a = b * (- c) + d

Three address code for the above example will be
t1 = -c
t2 = t1 * b
t3 = t2 + d
a = t3

3. Copy Statement

a = b
The value of b is stored in variable a.

4. Unconditional jump

goto L
Creates label L and generates three-address code ‘goto L’

Example of three address code -

Generate three address code for the following code-

while (A < C and B > D) do

if A = 1 then C = C + 1

else

while A <= D

do A = A + B

Solution-

Three address code for the given code is-

1. if (A < C) goto (3)
2. goto (15)
3. if (B > D) goto (5)
4. goto (15)
5. if (A = 1) goto (7)
6. goto (10)
7. T1 = c + 1
8. c = T1
9. goto (1)
10. if (A <= D) goto (12)
11. goto (1)
12. T2 = A + B
13. A = T2
14. goto (10)
15.

Implementation of Three Address Code –

There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

1. QUADRUPLES

The instruction of the quadruple presentation is divided into four fields - operator, arg1, arg2, and result.
One field to store operator op, two fields to store operands or arguments arg1and arg2 and one field to store result res. res = arg1 op arg2

Example:

a = b + c
b is represented as arg1, c is represented as arg2, + as op and a as res.
Unary operators like '-'do not use agr2. Operators like param do not use agr2 nor result.
For conditional and unconditional statements res is label. Arg1, arg2 and res are pointers to symbol table or literal table for the names.
Example: a = -b * d + c + (-b) * d
Three address code for the above statement is as follows
t1 = - b
t2 = t1 * d
t3 = t2 + c
t4 = - b
t5 = t4 * d
t6 = t3 + t5

a = t6

Quadruples for the above example is as follows

Op	Arg1	Arg2	Res
-	B		t1
*	t1	d	t2
+	t2	c	t3
-	B		t4
*	t4	d	t5
+	t3	t5	t6
=	t6		a

2. TRIPLES

The instruction of the Triples presentation is divided into three fields - op, arg1, and arg2. The position of the expression denotes the results of the respective sub-expressions. The similarity with DAG and syntax tree is represented by triples. When expressions are represented, they are equivalent to DAG.

Triples uses only three fields in the record structure. One field for operator, two fields for operands named as arg1 and arg2. Value of temporary variable can be accessed by the position of the statement the computes it and not by location as in quadruples.

Example: a = -b * d + c + (-b) * d

Triples for the above example is as follows

Stmt no	Op	Arg1	Arg2
(0)	-	b	
(1)	*	d	(0)
(2)	+	c	(1)
(3)	-	b	
(4)	*	d	(3)
(5)	+	(2)	(4)
(6)	=	a	(5)

Arg1 and arg2 may be pointers to symbol table for program variables or literal table for constant or pointers into triple structure for intermediate results.

Example:

Triples for statement x[i] = y which generates two records is as follows

Stmt no	Op	Arg1	Arg2
(0)	[]=	x	i
(1)	=	(0)	y

Triples for statement x = y[i] which generates two records is as follows

Stmt no	Op	Arg1	Arg2
(0)	=[]	y	i
(1)	=	x	(0)

3. Indirect Triples

Indirect triples are used to achieve indirection in listing of pointers. That is, it uses pointers to triples than listing of triples themselves.

Example:

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

	Stmt no
(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)
(6)	(16)

Stmt no	Op	Arg1	Arg2
(10)	-	b	
(11)	*	d	(0)
(12)	+	c	(1)
(13)	-	b	
(14)	*	d	(3)
(15)	+	(2)	(4)
(16)	=	a	(5)

BASIC BLOCKS AND FLOW GRAPHS

Basic Blocks

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.

The following sequence of three-address statements forms a basic block:

- t1 : = a * a

t2 : = a * b

t3 : = 2 * t2

t4 : = t1 + t3

t5 : = b * b

t6 : = t4 + t5

Basic Block Construction:

Algorithm: Partition into basic blocks

Input:

A sequence of three-address statements

Output:

A list of basic blocks with each three-address statement in exactly one block

Method:

1. We first determine the set of leaders, the first statements of basic blocks.

The rules we use are of the following:

a. The first statement is a leader.

b. Any statement that is the target of a conditional or unconditional goto is a leader.

c. Any statement that immediately follows a goto or conditional goto statement is a leader.

2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

- Consider the following source code for dot product of two vectors a and b of length 20

```
begin
        prod :=0;
       i:=1;
  do begin
          prod :=prod+ a[i] * b[i];
          i :=i+1;
    end
      while i <= 20
end
```

- The three-address code for the above source program is given as :

(1) prod := 0
(2) i := 1
(3) t1 := 4* i
(4) t2 := a[t1] /*compute a[i] */
(5) t3 := 4* i
(6) t4 := b[t3] /*compute b[i] */
(7) t5 := t2*t4

(8) t6 := prod+t5
(9) prod := t6
(10) t7 := i+1
(11) i := t7
(12) if i<=20 goto (3)

Basic block 1: Statement (1) to (2)
Basic block 2: Statement (3) to (12)

Transformations on Basic Blocks:

A number of transformations can be applied to a basic block without changing the set of expressions computed by the block. Two important classes of transformation are :

- Structure-preserving transformations
- Algebraic transformations

1. Structure preserving transformations:

a) Common subexpression elimination:

a : = b + c		a : = b + c
b : = a – d	⟶	b : = a - d
c : = b + c		c : = b + c
d : = a – d		d : = b

Since the second and fourth expressions compute the same expression, the basic block can be transformed as above.

b) Dead-code elimination:

Suppose x is dead, that is, never subsequently used, at the point where the statement x : = y + z appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

c) Renaming temporary variables:

A statement t : = b + c (t is a temporary) can be changed to u : = b + c (u is a new temporary) and all uses of this instance of t can be changed to u without changing the value of the basic block.
Such a block is called a normal-form block.

d) Interchange of statements:

Suppose a block has the following two adjacent statements:

t1 : = b + c
t2 : = x + y

We can interchange the two statements without affecting the value of the block if and only if neither x nor y is t1 and neither b nor c is t2.

2. Algebraic transformations:

Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

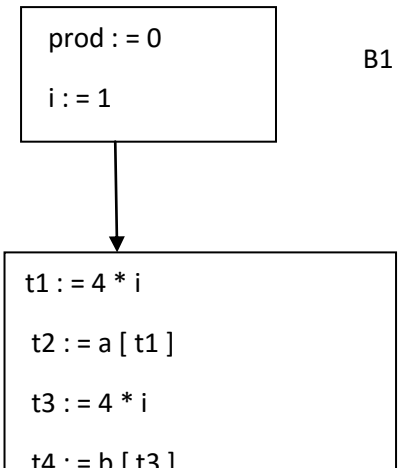
Examples:

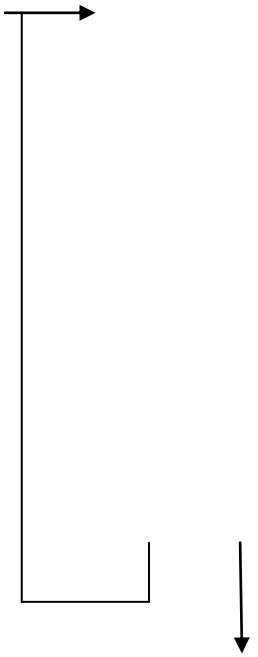
i) x : = x + 0 or x : = x * 1 can be eliminated from a basic block without changing the set of expressions it computes.
ii) The exponential statement x : = y * * 2 can be replaced by x : = y * y.

Flow Graphs

- Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program.
- The nodes of the flow graph are basic blocks. It has a distinguished initial node.

- E.g.: Flow graph for the vector dot product is given as follows:





- B1 is the initial node. B2 immediately follows B1, so there is an edge from B1 to B2. The target of jump from last statement of B1 is the first statement B2, so there is an edge from B1 (last statement) to B2 (first statement).
- B1 is the predecessor of B2, and B2 is a successor of B1

Directed Acyclic Graph-

Directed Acyclic Graph (DAG) is a special kind of **Abstract Syntax Tree**.

- Each node of it contains a unique value.
- It does not contain any cycles in it, hence called **Acyclic**.

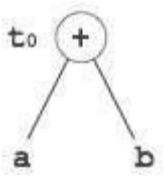
Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.

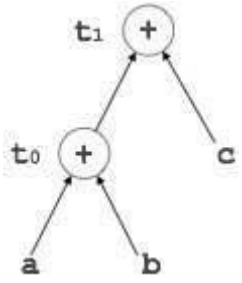
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:

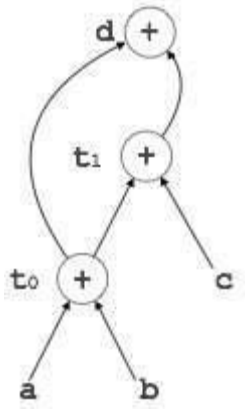
```
t₀ = a + b
t₁ = t₀ + c
d = t₀ + t₁
```



[$t_0 = a + b$]



[$t_1 = t_0 + c$]



[$d = t_0 + t_1$]

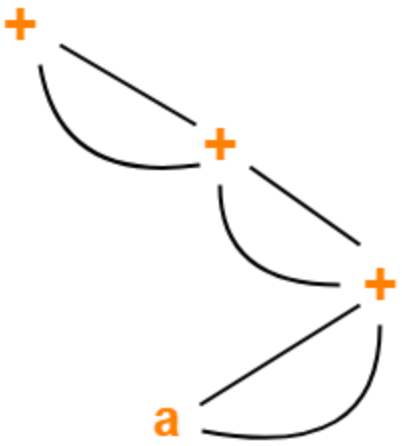
Example

Consider the following expression and construct a DAG for it-

(((a + a) + (a + a)) + ((a + a) + (a + a)))

Solution-

Directed Acyclic Graph for the given expression is-



Directed Acyclic Graph

Peephole Optimization

Peephole optimization is a type of Code Optimization performed on a small part of the code.

It is performed on the very small set of instructions in a segment of code.

It basically works on the theory of replacement in which a part of code is replaced by shorter and faster code without change in output.

Peephole is the machine dependent optimization.

Objectives of Peephole Optimization:
The objective of peephole optimization is:

1. To improve performance
2. To reduce memory footprint
3. To reduce code size

Peephole Optimization Techniques:

1. **Redundant load and store elimination:**
   In this technique the redundancy is eliminated.

```
Initial code:
y = x + 5;
i = y;
z = i;
w = z * 3;

Optimized code:
y = x + 5;
i = y;
```

```
w = y * 3;
```

2. **Constant folding:**
   The code that can be simplified by user itself, is simplified.

```
Initial code:
x = 2 * 3;

Optimized code:
x = 6;
```

3. **Strength Reduction:**
   The operators that consume higher execution time are replaced by the operators consuming less execution time.

```
Initial code:
y = x * 2;

Optimized code:
y = x + x;    or     y = x << 1;

Initial code:
y = x / 2;

Optimized code:
y = x >> 1;
```

4. **Null sequences:**
   Useless operations are deleted.

5. **Combine operations:**
   Several operations are replaced by a single equivalent operation.

This optimization technique works locally on the source code to transform it into an optimized code. By locally, we mean a small portion of the code block at hand. These methods can be applied on intermediate codes as well as on target codes.