## UNIT-IV

1. One of the purposes of using intermediate code in compilers is to
(A) make parsing and semantic analysis simpler.
(B) improve error recovery and error reporting.
(C) increase the chances of reusing the machine-independent code optimizer in other compilers.
(D)improve the register allocation.
Answer:(C)

2. Consider the following Syntax Directed Translation Scheme (SDTS), with non-terminals {S, A} and terminals {a,b}.
S→aA{print1} S→a{print2} S→Sb{print3}
Using the above SDTS, the output printed by a bottom-up parser, for the input aab is:
(A) 1 3 2
(B) 2 2 3
(C) 2 3 1
(D) syntax error
Answer: (C)

3. Which languages necessarily need heap allocation in the runtime environment?
(A) Those that support recursion
(B) Those that using dynamic scoping
(C) Those that allow dynamic data structure
(D) Those that use global variables
Answer: (C)

4. In the compiler, the function of using intermediate code is:
(A) to improve the register allocation
(B) to increase the error reporting & recovery.
(C) to make semantic analysis easier.
(D) to increase the chances of re-using the machine-independent code optimizer in other compilers.
Answer: (D)

5. In how many types of optimization can be divided?
(A) two types
(B) three types
(C) four types
(D) five types
Answer: (A)

6. Which algorithm invokes a function GETREG()?
(A) Code motion algorithm
(B) Code optimization algorithm
(C) Intermediate Code
(D) Code generation algorithm
Answer: (D)

7. Which statement is an abstract form of intermediate code?
(A) 3- address
(B) 2-address
(C) address
(D) Intermediate code
Answer: (A)

8. Which mapping is described by the implementation of the syntax-directed translator?
(A) Parse table
(B) Input
(C) Output
(D) Input-Output
Answer: (D)

9. How many descriptors are used for track both the registers (for availability) and addresses (location of values) while generating the code?
(A) 2
(B) 3
(C) 4
(D) 5
Answer: (A)

10. x * 2 can be replaced by x << 1 is an example of?
(A) Algebraic expression simplification
(B) Accessing machine instructions
(C) Strength reduction
(D) Code Generator
Answer: (C)

11. The following code is an example of?
voidadd_ten(int x)
{
return x + 10;
printf(""value of x is %d"", x);
}
(A) Redundant instruction elimination
(B) Unreachable code
(C) Flow of control optimization
(D) Reachable code
Answer: (B)

12. Code generator uses _____ function to determine the status of available registers and the location of name values.
(A) setReg
(B) cinReg
(C) pfReg
(D) getReg
Answer:(D)

13. Which of the following is not a form of Intermediate representation?
(A) Abstract Syntax Tree
(B) 3-address code
(C) Directed cyclic Graph
(D) Reverse Polish Notation
Answer: (C)

15. Some code optimizations are carried out on the intermediate code because _____
(A) They enhance the portability of the complier to other target processors
(B) Program analysis is name accurate on intermediate code than on machine code
(C) The information from data flow analysis cannot otherwise be used for optimization
(D) The information from the front end cannot otherwise be used for optimization
Answer: (B)

16. Consider the following intermediate program in three address code

```
p  =  a  -  b
q  =  p  *  c
p  =  u  *  v
q  =  p  +  q
```

Which one of the following corresponds to a static single assignment form of the above code?

(A)
```
p1 = a - b
q1 = p1 * c
p1 = u * c
q1 = p1 + q1
```

(B)
```
p3 = a - b
q4 = p3 * c
p4 = u * c
q5 = p4 + q4
```

(C)
```
p1 = a - b
q1 = p2 * c
p3 = u * c
q2 = p4 + q3
```

(D)
```
p1 = a - b
q1 = p * c
p2 = u * c
q2 = p + q
```

Answer:(B)

17. Which one of the following is FALSE?

(A) A basic block is a sequence of instructions where control enters the sequence at the beginning and exits at the end.
(B) Available expression analysis can be used for common subexpression elimination.
(C) Live variable analysis can be used for dead code elimination.
(D) $x = 4 * 5 \Rightarrow x = 20$ is an example of common subexpression elimination.
Answer:(D)

## PART-B

1. Open MPI is a Message Passing Interface library project combining technologies and resources from several other projects that is developed and maintained by a consortium of academic, research, and industry partners. In particular case, Intel C Compiler icc is used as the backend compiler but we also have variants that use GCC. The following are the packages that are involved in the versioning of OpenMPI model.

Fft= Numpy + Scipy + plot
Gsl = Scipy + plot
Python = Scikit + Matlib + Seaborn + plot + numpy
Gcc = Boost + atlas + hdf5 + plot
Torque = Fft + Gsl
OpenMPI = Torque + Python + Gcc

a) For the above hierarchy of the OpenMPI versioning, interpret the instruction and generate Three Address Code for the same.
b) Create Directed Acyclic Graph for the Three Address Code generated from the above Q.No (1a)

2. . For the code given below, determine the following : Constant propagation, live-variable analysis.

```
(a)  N = 10;
(b)  k = 1;
(c)  prod = 1;
(d)  MAX = 99;
(e)  while (k > N) {
(f)  read(num);
(g)  if (MAX/num < prod) {
(h)  print("cannot compute prod");
(i)  break;
(j)  }
```

(k) prod = prod * num;

(l) k++;

(m) }

print(prod);

3. Construct Three address Code for the following Code and Explain different kinds of Three address code representation.

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

Note: Consider the type of array is integer.

3. Translate the conditional statement if a<b then 1 else 0 into three address code

4. Explain the following with example: i) Quadruples ii) Triples iii) Indirect triple

5. **What are different intermediate code forms? Discuss different Three Address code types and implementations of Three Address statements**.
Ans: Everything related to three address code.

6.Explain the translation scheme to produce three address code for assignment statements.

7.Explain the Backpatching process for Boolean expression.

Backpatching is mainly used for two purposes:

**1. Boolean expression:**

Boolean expressions are statements whose results can be either true or false. A boolean expression which is named for mathematician George Boole is an expression that evaluates to either true or false. Let's look at some common language examples:

- My favorite color is blue. → true
- I am afraid of mathematics. → false

- 2 is greater than 5. → false

8.Explain the following terms: i) Register Descriptor ii) Address Descriptor iii) Instruction Cost.

9.Explain various issues in the design of the code generation.

## UNIT-V

1. Two standard storage-allocation strategies are

A. Stack Allocation and Queue Allocation

B. Stack Allocation and Static Allocation

C. Static Allocation and Queue Allocation

D. Simple Allocation and Static Allocation

ANSWER: B

2. Which field is not present in activation record

A. saved machine status

B. register allocation

C. optional control link

D. temporaries

ANSWER: B

3. which statement is correct about passing by value parameter?

A. it cannot change the actual parameter value

B. it can change the actual parameter value

C. parameter is always in read only mode

D. parameter is always in write only mode

ANSWER: A

4. Identify the synthesized and inherited attribute from the attributes: gen, kill, in, out

A. synthesized, inherited, inherited, synthesized

B. inherited, synthesized, synthesized, synthesized

C. synthesized, synthesized, inherited, inherited

D. synthesized, synthesized, inherited, synthesized

ANSWER: D

5. Which one of the following is FALSE?

A. A basic block is a sequence of instructions where control enters the sequence at the beginning and exits at the end.

B. Available expression analysis can be used for common subexpression elimination.

C. Live variable analysis can be used for dead code elimination.

D. x = 4 ∗ 5 => x = 20 is an example of common subexpression elimination.

ANSWER: D

6. The effect of the instruction LD R2, a(R1) is

A. R2= contents(a) + contents(R1)

B. R2= contents(a+ contents(R1))

C. R2= a + contents(R1)

D. R2= contents (contents(a+contents(R2)))

ANSWER: B

7. In the context of compiler design, reduction in strength refers to

A. code optimization obtained by the use of cheaper machine instructions.

B. reduction in accuracy of the output.

C. reduction in the range of values of input variables.

D. reduction in efficiency of the program.

ANSWER: A

8. In static allocation, names are bound to storage at _____ time

A. Compile

B. Runtime

C. Debugging

D. both (a) and (b)

ANSWER: A

9, Issues in code generation (i) Addressing Modes (ii) Choice of Evaluation Order (iii) Microprocessor Mnemonics (iv) Linking and Loading

A. Only (i), (ii)

B. Only (i), (ii), (iv)

C. Only (ii), (iii), (iv)

D. All of them

ANSWER: D

10. Activation Record stores: (i)Parameters (ii) Local Variables (iii) Parameters and Local Variables (iv) Parameters, Local Variables and code for procedures

A. (i)

B. (ii)

C. (iii)

D. (iv)

ANSWER: C

11. Function call actions are divided into sequences

A. Calling and composition

B. Return and composition

C. Calling and return

D. None of the other options

ANSWER: C

12. Evaluation of actual parameters is done by

A. Callee

B. Caller

C. Both Caller and Callee

D. None of the other options

ANSWER: B

13. If an activation of procedure ‘A’ calls procedure ‘B’ then which one is TRUE?

A. Activation of B must end before the activation of A can end.

B. Activation of A must end before the activation of B can end.

C. Activation of A must end before the activation of B can start.

D. Activation of B must start after the activation of A can end.

ANSWER: A

14. Which of the following field of an activation record will point to the activation record of the caller?

A. Retrned Values

B. Access Link

C. Temporaries

D. Control Link

ANSWER: D

15. Consider the following three address code. Identify the CORRECT collection of different optimization can be performed? m = 3

j = n

v = 2 * n

limit = integer n / 2

L1: j = j – 1

t4 = 4 * j

t5 = a[t4]

if t5 > limit – v goto L1

A. Code Motion, Constant Folding, Induction Variable Elimination, Reduction in Strength

B. Copy Propagation , Code Motion, Deadcode Elimination, Reduction in Strength

C. Constant Folding, Copy Propagation, Deadcode Elimination, Reduction in Strength

D. Code Motion, Constant Folding, Copy Propagation, Induction Variable Elimination

ANSWER: A

16. Which of the following tasks is managed by a runtime stack?

A. Static data and functions

B. Garbage collection

C. Procedure calls and returns

D. All of the above

ANSWER: C

17. Is code-optimization could be performed over the basic-block? x = t3

a[t2] = t5

a[t4] = x

goto B1

A. Not Possible

B. Yes Possible

C. Not enough information

D. Don’t know

ANSWER: B

18. The sequence of procedure calls of a program corresponds to which traversal of the activation tree?

A. In order traversal

B. Pre order traversal

C. Post order traversal

D. Level-order travel

ANSWER: B

19. The _____ code must not, in any way, change the meaning of the program.

A. input

B. output

C. print

D. execute

ANSWER: A

20.Optimization should increase the _____ of the program and if possible, the program should demand less number of resources.

A. length

B. count

C. speed

D. cost

ANSWER: C

PART-B

1) Write the code sequence for the d:=(a-b)+(a-c)+(a-c).
Apply code generation algorithm to generate a code sequence for the three address statement.

2) Consider the following block and construct a DAG for it-

(1) a = b x c

(2) d = b

(3) e = d x c

(4) b = e

(5) f = b + c

(6) g = f + d

3) Explain different storage allocation strategies.

Three main storage allocation mechanisms:

1. Static allocation.
   - Objects retain absolute address throughout.
   - Examples: global variables, literal constants: `"abc"`, `3`.
2. Stack-based allocation:
   - Object addresses relative to a stack (segment) base, usually in conjunction with fcn/proc calls.
3. Heap-based allocation. Objects allocated and deallocated at programmer's discretion.

4) Discuss briefly about DAG representation of basic blocks.

5) Explain in detail about primary structure-preserving transformations on basic blocks.

6) Discuss in detail about global data analysis.

**Unit V Important Topics:**

## Loop Optimization:

A loop is a collection of nodes in a flow graph such that:
1. All nodes in the collection are strongly connected, i.e, from any node in the loop to any other, there is a path of length one or more, wholly within the loop, and
2. The collection of nodes has a unique entry, i.e, a node in the loop such that the only
way to reach a node of the loop from a node outside the loop is through the entry point.
A loop that contains no other loops is called an inner loop.
Most programs run as a loop in the system. It becomes necessary to optimize the loops in
order to save CPU cycles and memory. Loops can be optimized by the following techniques
(a) Invariant code :
• If a computation produces the same value in every loop iteration, move it out of the loop.
• An expression can be moved out of the loop if all its operands are invariant in the loop
• Constants are loop invariants.
(b) Induction analysis:
A variable is called an induction variable if its value is altered within the loop by a loopinvariant value.
Eg:

```
extern int sum;
int foo(int n)
9
{
int i, j; j = 5;
for (i = 0; i < n; ++i)
{
j += 2; sum += j;
}
return sum;
}
```

This can be re-written as,

```
extern int sum;
int foo(int n)
{
int i; sum = 5;
for (i = 0; i < n; ++i)
{
sum += 2 * (i + 1);
}
return sum;
```

}
(c) Reduction in Strength:
There are expressions that consume more CPU cycles, time, and memory. These expressions
should be replaced with cheaper expressions without compromising the output of expression.
For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and
yields the same result.

```
c = 7;
for (i = 0; i < N; i++)
{
y[i] = c * i;
}
```

can be replaced with successive weaker additions

```
c = 7;
k = 0;
for (i = 0; i < N; i++)
{
10
y[i] = k;
k = k + c;
}
```

(d) Constant folding and constant propagation
Constant folding:
It is the process of recognizing and evaluating statements with constant expressions (i=22+222+2222 to i=2466), string concatenation (“abc”+”def” to “abcdef”) and expressions
with arithmetic identities (x=0; z=x*y[i]+x*2 to x=0; z=0;) at compile time rather than at
execution time.

## Code Optimization:

**Principles of source optimization:**
Optimization is a program transformation technique, which tries to improve the code by making it consume less resource (i.e. CPU, Memory) and deliver high speed. In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes.
A code optimizing process must follow the three rules given below:
• The output code must not, in any way, change the meaning of the program.
• Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
• Optimization should itself be fast and should not delay the overall compiling process.
Efforts for an optimized code can be made at various levels of compiling the process.
• At the beginning, users can change/rearrange the code or use better algorithms to write the code.
• After generating intermediate code, the compiler can modify the intermediate code by

address calculations and improving loops.
• While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

**Types of optimization**:

Optimization can be categorized broadly into two types: machine independent and machine dependent.

**Machine-independent Optimization:**

In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations. For example:

```
do
{
item = 10;
value = value + item;
} while(value<100);
```

This code involves repeated assignment of the identifier item, which if we put this way:

```
item = 10;
do
{
value = value + item;
} while(value<100);
```

should not only save the CPU cycles, but can be used on any processor.

**Machine-dependent Optimization**

Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine dependent optimizers put efforts to take maximum advantage of memory hierarchy.

**Organization of the code optimizer:**

The techniques used are a combination of Control-Flow and Data-Flow analysis as shown in Fig 4.1.

**Control-Flow Analysis**: Identifies loops in the flow graph of a program since such loops are usually good candidates for improvement.

**Data-Flow Analysis:** Collects information about the way variables are used in a program.

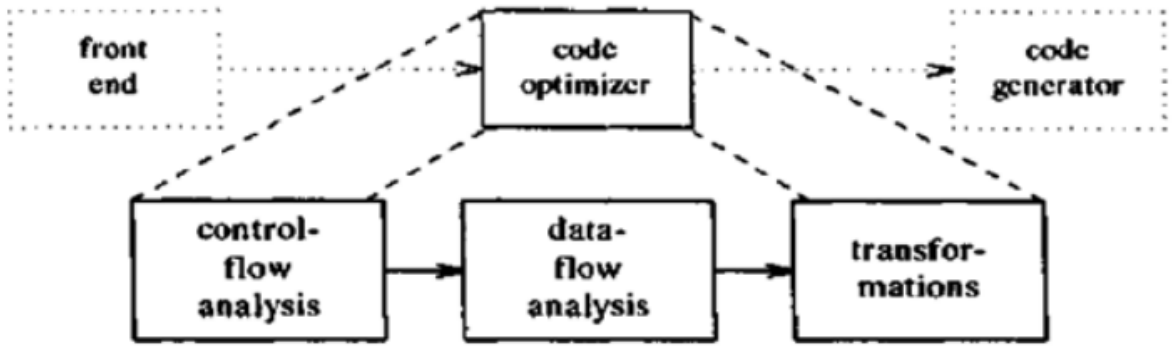


Fig 4.1. Organization of the code optimizer

**Steps before optimization:**

1) Source program should be converted to Intermediate code
2) Basic blocks construction
3) Generating flow graph
4) Apply optimization

## Optimization of basic blocks:

## Peep hole Optimization:

• Optimizing a small portion of the code.
• These methods can be applied on intermediate codes as well as on target codes.
• A bunch of statements is analyzed and are checked for the following possible optimization

**(1) Redundant instruction elimination**

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed.

For example:

```
MOV x, R0
MOV R0, R1
```

First instruction can be rewritten as

```
MOV x,R1
```

**(2) Unreachable Code**

It is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1. In C, the source code might look like:

```
#define debug 0
….
If ( debug )
{
Print debugging information
}
```

In the intermediate representations the if-statement may be translated as:

```
If debug =1 goto L1 goto L2
L1: print debugging information L2: …….....…… (a)
```

One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of debug; (a) can be replaced by:

```
If debug ≠1 goto L2
```

Print debugging information L2: ……..... (b)
If debug ≠0 goto L2 Print debugging information L2: ……..…… (c)
As the argument of the statement of (c) evaluates to a constant true it can be replaced
By goto L2. Then all the statement that print debugging aids are manifestly unreachable and
can be eliminated one at a time.

**(3) Flow of control optimization**

The unnecessary jumps can be eliminated in either the intermediate code or the target code by
the following types of peephole optimizations.
We can replace the jump sequence
goto L1
….
L1: gotoL2 (d)
by the sequence
goto L2
….
L1: goto L2
If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto
L2 provided it is preceded by an unconditional jump .Similarly, the sequence
if a < b goto L1
….
L1: goto L2 (e)
can be replaced by
If a < b goto L2
….
L1: goto L2
Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto.
Then the sequence
goto L1
L1: if a < b goto L2 (f)
L3:
may be replaced by
If a < b goto L2 goto L3
…….
L3:
While the number of instructions in (e) and (f) is the same, we sometimes skip the unconditional jump in (f), but never in (e).Thus (f) is superior to (e) in execution time

**(4) Algebraic Simplification**

There is no end to the amount of algebraic simplification that can be attempted through
peephole optimization. Only a few algebraic identities occur frequently enough that it is

worth considering implementing them. For example, statements such as
x := x+0
or
x := x * 1
are often produced by straightforward intermediate code-generation algorithms, and they can
be eliminated easily through peephole optimization.

**(5) Reduction in Strength**

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target
machine. Certain machine instructions are considerably cheaper than others and can often be
used as special cases of more expensive operators.
For example, $x^2$ is invariably cheaper to implement as x*x than as a call to an exponentiation
routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a
shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.
X2 → X*X

**(6) Use of Machine Idioms**

The target machine may have hardware instructions to implement certain specific operations
efficiently. For example, some machines have auto-increment and auto-decrement addressing
modes. These add or subtract one from an operand before or after using its value. The use of
these modes greatly improves the quality of code when pushing or popping a stack, as in
parameter passing. These modes can also be used in code for statements like i : =i+1.
i:=i+1 → i++
i:=i-1 → i-

## DAG:

One of the important techniques for local optimization begin by transforming a basic block into a DAG (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:

```
a = b + c
b = a - d
c = b + c
d = a - d
```

Computation of Gen & Kill/in and out (2 marks type)