4 Compiler Design

1. Overview

01. Consider the grammar
S→ABC|bc
BA→AB
Bb→bb
Ab→ab
Aa→aa
Which of the following sentences can be derived by this grammar?
(a) abc (b) aabc
(c) abcc (d) abbc
[ISRO - 2008]

02. Relative to the program translated by a complier, the same program when interpreted runs
(a) faster
(b) slower
(c) at the same speed
(d) may be faster or slower
[ISRO - 2008]

03. Which of the following is **NOT** represented in a subroutine's activation record frame for a stack-based programming language?
(a) Values of local variables
(b) Return address
(c) Heap area
(d) Information needed to access non local variables
[ISRO - 2014]

04. A simple two-pass assembler does which of the following in the first pass?
(a) Checks to see if the instructions are legal in the current assembly mode
(b) It allocates space for the literals
(c) It builds the symbol table for the symbols and their values
(d) All of these
[ISRO - 2016]

05. The output of a lexical analyzer is
(a) A parse tree
(b) Intermediate code
(c) Machine code
(d) A stream of tokens
[ISRO - 2017(May

06. Which languages necessarily need heap allocatio in the run time environment?
(a) Those that support recursion
(b) Those that use dynamic scoping
(c) Those that use global variables
(d) Those that allow dynamic data structures
[ISRO - 2017(Dec

07. Consider the following table :

A. Activation record	p. Linking loader
B. Location counter	q. Garbage collection
C. Reference counts	r. Subroutine call
D. Address relocation	s. Assembler

Matching A, B, C, D in the same order gives :
(a) p, q, r, s
(b) q, r, s, p
(c) r, s, q, p
(d) r, s, p, q
[ISRO - 2017(Dec

08. Which of the following comparisons between stati and dynamic type checking is incorrect?
(a) Dynamic type checking slows down the execution
(b) Dynamic type checking, offers more flexibili to the programmers.
(c) In contrast to Static type checking, dynamic type checking may cause failure in runtime du to type errors
(d) Unlike static type checking, dynamic type checking is done during compilation.
[ISRO - 201

09. Incremental-
(a) which is
from the
(b) compiles
object co
(c) compiles
have bee
(d) that runs
code for

10. In a two-pa
calls and in
done during
(a) second
(b) first pas
(c) second
(d) first pa

11. Which on
compilers'
(a) Beta c
(c) Mexica

KE
01. (*
05. (d
09. (c

01. Ans: (*)
Sol: S→ ABC
BA→AB
Bb→bb
Ab→ab
Aa→aa
In the g
S→ AB
from S.

09. Incremental-Compiler is a compiler
(a) which is written in a language that is different from the source language
(b) compiles the whole source code to generate object code afresh
(c) compiles only those portion of source code that have been modified.
(d) that runs on one machine but produces object code for another machine **[ISRO - 2018]**

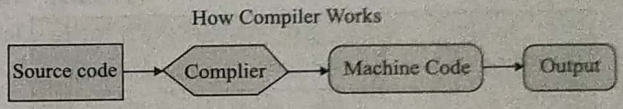
10. In a two-pass assembler, resolution of subroutine calls and inclusion of labels in the symbol table is done during
(a) second pass
(b) first pass and second pass respectively
(c) second pass and first pass respectively
(d) first pass **[ISRO - 2020]**

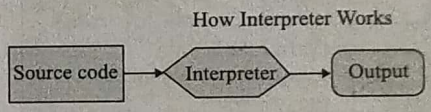
11. Which one indicates a technique of building cross compilers?
(a) Beta cross (b) Canadian cross
(c) Mexican cross (d) X-cross
[ISRO - 2020]

KEY & Detailed Solutions			
01. (*)	02. (b)	03. (c)	04. (d)
05. (d)	06. (d)	07. (c)	08. (d)
09. (c)	10. (c)	11. (b)	

01. Ans: (*)
Sol: S→ ABCc | bc
BA→AB
Bb→bb
Ab→ab
Aa→aa
In the given grammar start symbol is S, Except S→ **ABCc | bc**, no other production is reachable from S.
Hence, **bc** is the only string that can be generated. It is not mentioned in options.

02. Ans: (b)
Sol: A compiled program(written in a language like C) is usually translated directly into machine code. When you run the code, it is executed directly by the CPU.
Whereas In case of interpreter, When you execute your code, the CPU executes the interpreter and the interpreter reads and executes your source code, this makes it slower than compiler





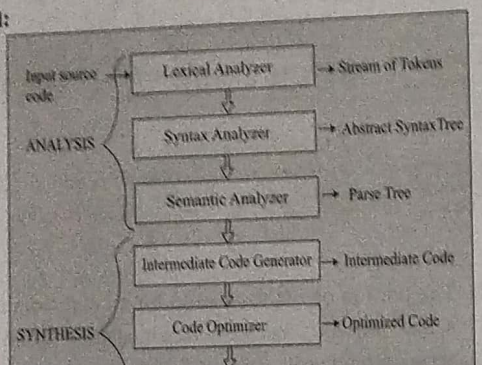
03. Ans: (c)
Sol: The portion of the stack used for an invocation of a function is called the function's stack frame or activation record.
It is generally composed of:
Local data of the callee
Return address to the caller
Parameters of the callee.
Optional links to access non-local variables. It does not include a heap.
Heap area is required in the case of dynamic variables.

04. Ans: (d)
Sol: All the options mentioned are the responsibilities of the Assembler.
Assembler is a program that converts instructions written in low-level assembly code into relocatable machine code and generates information for the loader.
Two pass assemblers divide these tasks in two passes:

Pass-1:
Define symbols and literals and remember them in symbol table and literal table respectively.
Keep track of location counter
Process pseudo-operations
Pass-2:
Generate object code by converting symbolic op-code into respective numeric op-code
Generate data for literals and look for values of symbols

05. Ans: (d)

Sol:



06. Ans: (d)

Sol: Global variables are stored in the data section of the program, not on heap.
Stack memory implementation helps the runtime environment to support Recursion.
Run-time environments that allow dynamic data structure (ex: malloc function) use memory from the heap area.

07. Ans: (c)

Sol: **A. Activation record → r. Subroutine call**
Activation Records are required to keep the information of subroutine calls.
B. Location counter → s. Assembly
Assembler maintains a location counter to assign storage addresses to each instruction of the sour program.
C. Reference counts → q. Garbage collection
For each object, the Reference counter of t
Garbage collection tracks the count of the numb of references made.
D. Address relocation → p. Linking loader
A linking loader generally performs the relocati of code.

08. Ans: (d)

Sol: Dynamic type checking slows down the executio **True.**
Because, we need to wait till the run time for ty checking.
Dynamic type checking offers more flexibility the programmers, **True.**
We can achieve overriding (runtime polymorphism with dynamic type checking.
In contrast to Static type checking, dynamic ty checking may cause failure in runtime due to ty errors, **True.**
Unlike static type checking, dynamic type checki is done during compilation, **False.**
Unlike dynamic type checking, static type checki is done during compilation.

09. Ans: (c)

Sol: Incremental-Compiler is a compiler that compil only those portions of source code that have be modified.

10. Ans: (c)

Sol: Assembler is a program that converts instructio written in low-level assembly code into relocatab machine code and generates information for t loader.
Two pass assemblers divide these tasks in tw passes:

Pass-1:
Define symbols and literals and remember them in symbol table and literal table respectively.
Keep track of location counter
Process pseudo-operations
Pass-2:
Generate object code by converting symbolic op-code into respective numeric op-code
Generate data for literals and look for values of symbols
Hence, resolution of subroutine calls is done during second pass and inclusion of labels in the symbol table is done during the first pass.

11. Ans: (b)

Sol: Native Compiler :
Native compilers are compilers that generate code for the same Platform on which it runs. It converts high language into computer's native language.

Cross compiler :
A Cross compiler is a compiler that generates executable code for a platform other than one on which the compiler is running.
Following are the techniques used for building cross compilers:
1. Canadian cross
2. GCC compiler
3. Manx Aztec C cross
4. Microsoft C cross compiler

2. Lexical Analysis

01. The number of token the following C statement is
printf ("i = %d, &i=%x", i&i) ;
(a) 13 (b) 6 (c) 10 (d) 11
[ISRO - 2015]

KEY & Detailed Solutions
01. (c)

01. Ans: (c)

Sol: printf ("i = %d, &i=%x", i,&i) ;

Token	Printf	(	"i = %d, & i = %x"	,	i	,	&	i	)	;
Count	1	2	3	4	5	6	7	8	9	10

3. Parsing Techniques

01. Which variable does not drive a terminal string in the grammar
S → AB
A → a
B → b
B → c
(a) A (b) B (c) C (d) S
[ISRO - 2011]

02. Which of the following sentences can be generated by
S → aS | bA
A → d | cA
(a) bccdd (b) abbcca
(c) abcabc (d) abcd
[ISRO - 2011]

03. Shift reduce parsing belongs to a class of
(a) bottom up parsing (b) top down parsing
(c) recursive parsing (d) predictive parsing
[ISRO - 2013]

04. Which of the following productions eliminate left recursion in the productions given below:
S → Aa |b
A → Ac |Sd| ε
(a) S→Aa|b, A → bdA^1, A^1→ A^{1}c|A^{1}ba|A|ε
(b) S→Aa|b, A → A^1 |bdA^1, A^1→ cA^1|adA^1|ε
(c) S→Aa|b, A → A^{1}c|A^{1}d, A^1 →bdA^1 |cA|ε
(d) S→Aa|b, A → cA^1 |adA^1|bdA^1 , A^1→ A|ε
[ISRO - 2013]

05. What is the number steps required to derive the string ((() ()) ()) for the following grammar.
S → SS
S → (S)
S → ε
(a) 10 (b) 15 (c) 12 (d) 16
[ISRO - 2014]

06. Given the following expression grammar :
E → E * F | F + E | F
F → F – F | id
Which of the following is true?
(a) *has higher precedence than +
(b) – has higher precedence than *
(c) + and – have same precedence
(d) + has higher precedence than *
[ISRO - 20

07. Which grammar rules violate the requirement the operator grammar? A, B, C are variables and b, c are terminals
(i) A → BC
(ii) A → CcBb
(iii) A → BaC
(iv) A → ε
(a) (i) only (b) (i) and (ii)
(c) (i) and (iii) (d) (i) and (iv)
[ISRO - 20

08. Which one of the following is a top-down parse
(a) Recursive descent parser
(b) Shift left associative parser
(c) SLR (k) parser
(d) LR (k) parser **[ISRO - 20**

09. Yacc stands for
(a) yet accept compiler constructs
(b) yet accept compiler compiler
(c) yet another compiler constructs
(d) yet another compiler compiler
[ISRO - 20

10. Which statement is true?
(a) LALR parser is most powerful and costly as compare to other parsers
(b) All CFG's are LP and not all grammars are uniquely defined
(c) Every SLR grammar is unambiguous but no every unambiguous grammar is SLR
(d) LR (K) is the most general back tracking sh reduce parsing method **[ISRO - 20**

11. Recursive descent parsing is an example of
(a) Top-down parsers (b) Bottom-up parsers
(c) Predictive parsers (d) None of these
[ISRO - 2016]

12. A top-down parser generates
(a) Rightmost derivation
(b) Rightmost derivation in reverse
(c) Leftmost derivation
(d) Leftmost derivation in reverse **[ISRO - 2016]**

13. A given grammar is called ambiguous if
(a) two or more productions have the same non-terminal on the left hand side
(b) a derivation tree has more than one associated sentence
(c) there is a sentence with more than one derivation tree corresponding to it
(d) brackets are not present in the grammar
[ISRO - 2020]

14. Given the grammar
$s \rightarrow T*S \mid T$
$T \rightarrow U + T \mid U$
$U \rightarrow a \mid b$
Which of the following statements is wrong?
(a) Grammar is not ambiguous
(b) Priority of + over * is ensured
(c) Right to left evaluation of * and + happens
(d) None of these **[ISRO - 2020]**

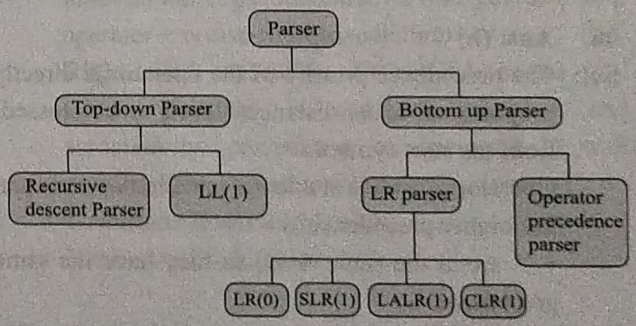
15. A grammar is defined as
$A \rightarrow BC$
$B \rightarrow x \mid Bx$
$C \rightarrow B \mid D$
$D \rightarrow y \mid Ey$
$E \rightarrow z$
The non-terminal alphabet of the grammar is
(a) {A, B, C, D, E}
(b) {B, C, D, E}
(c) {A, B, C, D, E, x, y, z}
(d) {x, y, z} **[ISRO - 2020]**

KEY & Detailed Solutions				
01. (c)	02. (d)	03. (a)	04. (b)	05. (a)
06. (b)	07. (d)	08. (a)	09. (d)	10. (c)
11. (a)	12. (c)	13. (c)	14. (d)	15. (a)

01. Ans: (c)
Sol: C is the useless variable as there is no production rule which replaces C with a terminal. Hence it does not derive any non-terminal.

02. Ans: (d)
Sol: $S \rightarrow aS \mid bA$
$A \rightarrow d \mid cA$
$S \rightarrow aS \rightarrow abA \rightarrow abcA \rightarrow abcd$ (Hence abcd can be derived)
Other strings cannot be derived.

03. Ans: (a)
Sol: Shift reduce is the class of parsers which builds a parse tree in a bottom-up manner and scans it from left to right.
Examples: SLR, CLR, LALR etc.



04. Ans: (b)
Sol: To remove left recursion from the grammar of the form :
$A \rightarrow A\alpha \mid \beta$
We rewrite the production rules as:
$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \varepsilon$

Given Grammar:

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \varepsilon$

indirect left recursion exists in the grammar:

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

here, $\alpha = c$ and $\alpha = ad$

$\beta = bd$

So, Grammar after removing left recursion

$S \rightarrow Aa \mid b$

$A \rightarrow A^1 \mid bdA^1$

$A^1 \rightarrow cA^1 \mid adA^1 \mid \varepsilon$

05. Ans: (a)

Sol: $s \rightarrow (s)$

$\rightarrow (ss)$

$\rightarrow (s(s))$

$\rightarrow (s())$

$\rightarrow ((s)())$

$\rightarrow ((ss)())$

$\rightarrow (((s)s)())$

$\rightarrow (((()s)())$

$\rightarrow (((()(s))())$

$\rightarrow (((()())())$

So 10 steps are required to derive the given string.

06. Ans: (b)

Sol: The precedence/priority of the operator is directly proportional to its distance(#levels to be crossed) from the start symbol.

Operator which is at a lower level in the grammar, has higher precedence.

+, * are at the same level, so they have the same precedence.

− has more precedence compared to +,*.

07. Ans: (d)

Sol: A grammar that is used to define mathematical operators is called an **operator grammar** or **operator precedence grammar**. Such grammars have the restriction that no production has either an empty right-hand side (null productions) or two adjacent non-terminals in its right-hand side.

(i) $A \rightarrow BC$ (**violation**, production has two adjacent non-terminals in its right-hand)

(ii) $A \rightarrow CcBb$ (No violation)

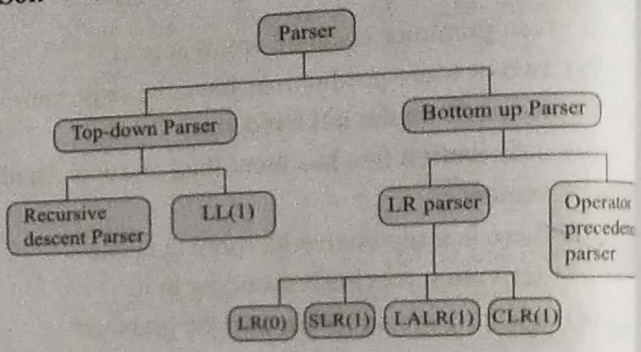
(iii) $A \rightarrow BaC$ (No violation)

(iv) $A \rightarrow \varepsilon$ (**violation**, Null productions are not allowed in operator grammar)

Hence Option (d) is the Answer.

08. Ans: (a)

Sol: A Recursive Descent Parser is a Top Down Parser



09. Ans: (d)

Sol: Yacc (Yet Another Compiler-Compiler) is computer program for the Unix operating system. It is a compiler-compiler means a compiler which creates a compiler.

10. Ans: (c)

Sol: LALR parser is the most powerful and costly compared to other parsers. **False.**

CLR(1) parser is the most powerful and costly compared to other parsers

LR(0) < SLR(1) < LALR(1) < CLR(1)

All CFG's are LP and not all grammars are uniquely defined. **False.**

The grammar generated by LP is CFG, but it does not imply All CFG are LP.

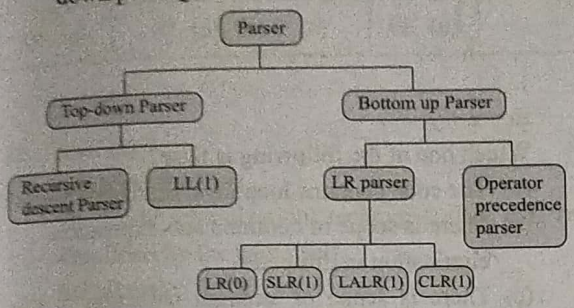
Every SLR grammar is unambiguous but not every unambiguous grammar is SLR. **True.**

LR(K) is the most general backtracking shift reduce parsing method. **False**

LR(k) is the most general non backtracking shift reduce parsing method.

11. Ans: (a)

Sol: Recursive descent parsing is an example of Top-down parsing.



12. Ans: (c)

Sol: The top-down parser is LL parser.

First L means, Scan the input from Left to Right, second L means Left most derivation(LL).

Bottom up parser is LR parser.

First L means, Scan the input from Left to Right, second R means Right most derivation in reverse order.(LR).

13. Ans: (c)

Sol: **If two or more productions have the same nonterminal on the left hand side, Grammar may or may not be ambiguous.**

$S \rightarrow BA$

$A \rightarrow e$

$A \rightarrow d$

$B \rightarrow f$

In the above grammar, $A \rightarrow d$, $A \rightarrow e$ have the same nonterminal on the left hand side, still the grammar is not ambiguous.

Because, {fe, fd} is the language generated by the grammar and no string in the language has two or more derivation trees.

If a derivation tree has more than one associated sentence

It is not possible.

A sentence can have more than one derivation tree in case of ambiguous grammar, but irrespective of the ambiguity of the grammar, A derivation tree always yields a unique sentence.

If there is a sentence with more than one derivation tree corresponding to it, then the grammar is ambiguous,True.

If brackets are not present in the grammar, the grammar may or may not be ambiguous.

$S \rightarrow (S) \mid ( )$

The above grammar involves brackets, still it is unambiguous.

14. Ans: (d)

Sol: Given the grammar

$S \rightarrow T*S \mid T$

$T \rightarrow U + T \mid U$

$U \rightarrow a \mid b$

The precedence/priority of the operator is directly proportional to its distance(#levels to be crossed) from the start symbol.

Operator which is at a lower level in the grammar, has higher precedence.

+ has more precedence compared to *.

Priority of + over * is ensured

$S \rightarrow T*S \mid T$

Since the above production involves right recursion, operator * is right associative.

$T \rightarrow U + T \mid U$

Since the above production involves right recursion, operator + is also right-associative.

Right to left evaluation of * and + happens

Given grammar is operator grammar and associativity, precedence of operators is well defined, Hence Grammar is unambiguous.

Grammar is not ambiguous

Hence (a), (b) & (c) are True.

15. Ans: (a)

Sol: $A \rightarrow BC$

$B \rightarrow x \mid Bx$

$C \rightarrow B \mid D$

$D \rightarrow y \mid Ey$

$E \rightarrow z$

Non-Terminals: {A,B,C,D,E}

Terminals: {x, y, z}

4. Code Optimization

01. Substitution of values for names (whose values are constants) is done in
(a) Local optimization (b) Loop optimization
(c) Constant folding (d) Strength reduction
[ISRO - 2009]

02. In compiler terminology, reduction in strength means
(a) Replacing run time computation by compile time computation
(b) Removing loop invariant computation
(c) Removing common subexpressions
(d) Replacing a costly operation by a relatively cheaper one
[ISRO - 2011]

03. Which of the following statements about peephole optimizations is False?
(a) It is applied to a small part of the code
(b) It can be used to optimize intermediate code
(c) To get the best out of this, it has to be applied repeatedly
(d) It can be applied to a portion of the code that is not contiguous
[ISRO - 2011]

04. Peephole optimization is a form of
(a) Loop optimization (b) Local optimization
(c) Constant folding (d) Data flow analysis
[ISRO - 2016]

05. Consider the code segment

```
int, i, j, x, y, m, n;
n = 20;
for (i = 0, i < n; i++)
{
  for (j = 0; j < n; j++)
  {
    if (i % 2)
    {
      x += ((4*j) + 5*i);
      y += (7 + 4*j);
    }
  }
}
m = x + y;
```

Which one of the following is false?
(a) The code contains loop invariant computation
(b) There is scope of common sub-expression elimination in this code
(c) There is scope of strength reduction in the code
(d) There is scope of dead code elimination in the code
[ISRO - 2017(Dec)]

06. DU-chains(Definition-Use) in compiler design
(a) consist of a definition of a variable and all its uses, reachable from that definition
(b) are created using a form of static code analysis
(c) are prerequisite for many compile optimization including constant propagation and common sub-expression elimination
(d) All of the above **[ISRO - 2018]**

07. Which of the following comment about peep-hole optimization is true?
(a) It is applied to small part of the code and applied repeatedly
(b) It can be used to optimize intermediate code
(c) It can be applied to a portion of the code that is not contiguous
(d) It is applied in symbol table to optimize the memory requirements. **[ISRO - 2018]**

08. Which of the following is a type of a out-of-order execution, with the reordering done by a compiler
(a) loop unrolling
(b) dead code elimination
(c) strength reduction
(d) software pipelining
[ISRO - 2020]

KEY & Detailed Solutions				
01. (c)	02. (d)	03. (*)	04. (b)	05. (d)
06. (d)	07. (a), (b) & (c)		08. (d)	

01. Ans: (c)

Sol: Constant folding:

The code optimization technique that can be simplified by the user itself.

Initial code:

```
int fun()
{
    return(2*6);
}
```

Optimized code:

```
int fun()
{
    return(12);
}
```

02. Ans: (d)

Sol: Strength Reduction:

The operators that consume higher execution time are replaced by the operators consuming less execution time.

Initial code:

```
a= b * 2;
```

Optimized code:

```
a = b + b;   or   a = b << 1;
```

03. None of the options

Sol: All options are TRUE about peephole optimization.

It is applied to a small part of the code

It can be used to optimize intermediate code

To get the best out of this, it has to be applied repeatedly

It can be applied to a portion of the code that is not contiguous

Peephole is machine dependent optimization. Its a local optimization technique.

Objectives of Peephole Optimization:

To improve performance

To reduce memory footprint

To reduce code size

Peephole Optimization Techniques:

Redundant load and store elimination

Constant folding

Strength Reduction

Null sequences

Combine operations

04. Ans: (b)

Sol: Peephole is machine dependent optimization. Its a local optimization technique.

Objectives of Peephole Optimization:

To improve performance

To reduce memory footprint

To reduce code size

Peephole Optimization Techniques:

Redundant load and store elimination

Constant folding

Strength Reduction

Null sequences

Combine operations

05. Ans: (d)

Sol: There is scope for common subexpression elimination in this code,True.

```
for (i = 0, i<n; i++)
{
  for (j=0; j<n; j++)
  {
    if (i%2)
    {
      x += (4*j + 5*i);
      y += (7 + 4*j);
    }
  }
}
```

We are computing 4*j twice per iteration,we can avoid this.

After Common subexpression elimination,

```
for (i = 0, i<n; i++)
{
  for (j=0; j<n; j++)
  {
    if (i%2)
{
  z=4*j
  x += (z + 5*i);
  y += (7 + z);
  }
 }
}
```

The code contains loop invariant computation, **True**.
With respect to the inner loop, i value is invariant for each iteration.

After removing loop invariant computation,

```
for (i = 0, i<n; i++)
{  y=5*i;
   c=i%2;
   for (j=0; j<n; j++)
   {
     if (c)
     {
        z=4*j
        x += (z +y );
        y += (7 + z);
     }
   }
  }
```

There is scope of strength reduction in this code, **True.**
Right shift is a cheaper operation compared to Multiplication.
k right shifts are equivalent to multiplying with 2^k.
After applying strength reduction optimization.

```
for (i = 0, i<n; i++)
{  y = (i>>2)+1;
   c = i%2
   for (j = 0; j<n; j++)
   {
     if (c)
     {
       z = j>>2
       x += (z +y );
       y += (7 + z);
     }
   }
  }
```

There is scope for dead code elimination in t code, False.
There is no dead code in the given code snippet

06. Ans: (d)

Sol: All the statements are TRUE.

A definition use chain (DU-Chain) is data structu It consists of a definition of a variable and all uses that are reachable from that definition variable.

The main purpose of DU-chains is livene analysis.

DU-chains (Definition-Use) in compiler design a created using a form of static code analysis.

07. (a), (b) & (c)

Sol: All the options (a), (b) & (c) are TRUE abo peephole optimization.

It is applied to a small part of the code and appli repeatedly

It can be used to optimize intermediate code

It can be applied to a portion of the code that is n contiguous

Peephole is machine dependent optimization. Its a local optimization technique.

Objectives of Peephole Optimization:

To improve performance

To reduce memory footprint

To reduce code size

Peephole Optimization Techniques:

Redundant load and store elimination

Constant folding

Strength Reduction

Null sequences

Combine operations

08. Ans: (d)

Sol: loop unrolling

This optimizatio needed.

Code before loo

```
int countbit(unsi
{
  int bits = 0;
  while (n != 0)
  {
    if (n & 1) b
    n >>= 1;
  }
  return bits;
}
```

Code after loo

```
int countbit(uns
{
   int bits = 0
   while (n !=
   {
     if (n & 1
     if (n & 2
     if (n & 4
     if (n & 8
     n >>= 4;
   }
   return bits;
}
```

dead code eli

Code that is u program (e.g.

Code before

```
int global;
void f ()
{
   int i;
   i = 1;
   global =
   global =
   return;
   global =
}
```

08. **Ans: (d)**

Sol: **loop unrolling**

This optimization reduces the number of iterations needed.

Code before loop unrolling:

```
int countbit(unsigned int n)
{
  int bits = 0;
  while (n != 0)
  {
    if (n & 1) bits++;
    n >>= 1;
  }
  return bits;
}
```

Code after loop unrolling:

```
int countbit(unsigned int n)
{
   int bits = 0;
   while (n != 0)
   {
     if (n & 1) bits++;
     if (n & 2) bits++;
     if (n & 4) bits++;
     if (n & 8) bits++;
     n >>= 4;
   }
   return bits;
}
```

dead code elimination

Code that is unreachable or that does not affect the program (e.g. dead stores) can be eliminated.

Code before dead code elimination:

```
int global;
void f ()
{
   int i;
   i = 1;        /* dead store */
   global = 1;   /* dead store */
   global = 2;
   return;
   global = 3;   /* unreachable */
}
```

Code after dead code elimination:

```
int global;
void f ()
{
   global = 2;
   return;
}
```

strength reduction

The operators that consume higher execution time are replaced by the operators consuming less execution time.

Code before strength reduction:

```
a = b * 2;
```

Code after strength reduction:

```
a = b + b;   or   a = b << 1;
```

Software pipelining

In computer science, software pipelining is a technique used to optimize loops, in a manner that parallels hardware pipelining. Software pipelining is a type of out-of-order execution, except that the reordering is done by a compiler (or in the case of hand written assembly code, by the programmer) instead of the processor. Some computer architectures have explicit support for software pipelining, notably Intel's IA-64 architecture.