Section-A (10*2=20)

Q.1 Differentiate between dynamic loaders and linkers.

Ans.

LINKER	LOADER
(i) The main function of linker is to generate executable files.	(i) Whereas main objective of loader is to load executable files to main memory
(ii) Linker takes input of object code generated by compiler / assembler.	(ii) And the loader takes the input of executable files generated by linker.
(iii) Linking can be defined as the process of combining various pieces of codes and source code to obtain executable code.	(iii) Loading can be defined as process of loading executable codes to main memory for further execution.
(iv) Linkers are of 2 types: Linkage editor and Dynamic linker.	(iv) loaders are of 4 types: Absolute, relocating, direct and bootstrap.
(v) Linker is also responsible for arranging objects in program's address space	(v) Responsible for adjusting references which are used within the program.
(vi) It is used to combine all object modules.	(vi) Helps in allocating the address to executable files.

Q2 Describe languages denoted by the following regular expression $(1+0)^*$.

Ans Any string starting with 1 or 0 or ε.

Q3. What is the role of lookahead component in CLR parser?

Ans. LR(1) item is a collection of LR(0) items and a lookahead symbol. The lookahead is used to determine that where we place the final item. The lookahead always add $ symbol for the argument production.

Q4 Differentiate b/w operator grammar and operator precedence grammar.

Ans Operator grammar has two characteristics.

(i) No two adjacent non terminal on the right side of production.

(ii) ϵ is not in the right side of production.

Operator precedence grammar is an operator grammar in which all three relations $<$, $>$ and $=$ between each pair of terminal is ~~joint~~ disjoint.

Q5 What is backpatching? Explain.

Ans (i) The main difficulty with code generation in one pass is that we may not know the target of a branch when we generate code for flow of control ~~of~~ statements.

(ii) Backpatching is the technique to get around this problem.

~~(iii)~~
- Generate branch instructions with empty targets.
- When the target is known, fill in the label of the branch instructions (backpatching).

(iii) Boolean expressions.

Q6 Write the prefix and postfix expression for A = (20 + (-5) * 6 + 12).

Ans PREFIX NOTATION:

A = (20 + (-5) * 6 + 12)
= (20 + -5 * 6 + 12)
= (20 + * - 5 6 + 12)
= + 20 * - 5 6 + 12
= + + 20 * - 5 6 12

POSTFIX NOTATION:

A = (20 + (-5) * 6 + 12)
= (20 + 5 - * 6 + 12)
= (20 + 5 - 6 * + 12)
= (20 5 - 6 * + + 12)
= 20 5 - 6 * + 12 +

Q7. What is control stack in runtime environment?

Ans. Control stack or runtime stack is used to keep track of the live procedure activations i.e the procedures whose execution have not been completed. A procedure name is pushed on to the stack when it is called and it is popped when it returns.

Q8. What do you mean by dynamic memory allocation?

Ans. Dynamic memory allocation is the process of assigning the memory space during the execution time or the run time. Reasons and advantage of allocating memory dynamically. when we do not know how much amount of memory would be needed for the program beforehand.

Q9. What are advantages of DAG (Directed Acyclic Graph)?

Ans
- DAGs are a type of data structure. It is used to implement transformations on basic blocks.
- DAG provides a good way to determine the common sub-expression.
- It gives a picture representation of how the ~~computed~~ value computed by the statement is used in subsequent statements.

Q10. What is the role of algebraic law in code optimization?

Ans The algebraic identities are used to optimize basic blocks.

$$x + 0 = 0 + x = x$$

$$x - 0 = x$$

$$x * 1 = 1 * x = x$$

$$x / 1 = x$$

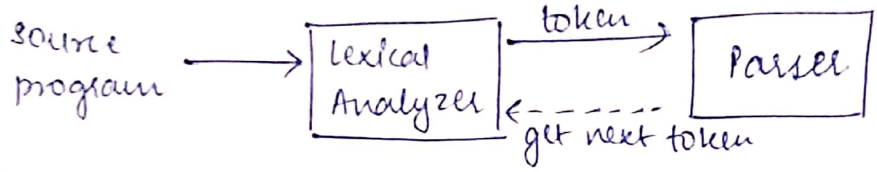
Commutations law : $a + b = b + a$ & $a * b = b * a$

$$x * * 2 = x * x$$

Section-B

Q11. Explain the role of lexical analyzer in compilation process. Enumerate the issues handled by a lexical analyser.

Ans. Lexical analyzer reads the program(source) character by character to produce tokens. Normally a lexical analyzer doesn't return a list of tokens at one shot, it returns a token when the parser asks a token from it.



TOKEN:

- Token represents a set of strings described by a pattern.
  - Identifier represents a set of strings which start with a letter continues with letters and digits.
  - The actual string (newval) is called as lexeme.
  - Tokens: identifier, number, addop, delimiter.
- Since a token can represent more than one lexeme, additional information should be held for that specific lexeme. This additional information is called as the attribute of the token.
- For simplicity, a token may have a single attribute which holds the required information for that token.
  - For identifiers, this attribute a pointer to the symbol table, and the symbol table holds the actual attributes for that token.

ISSUES HANDLED BY LEXICAL ANALYZER:

(i) We call the recognizer of the tokens as a finite automaton.

(ii) A finite automata can be DFA or NFA.

(iii) Both deterministic and non-deterministic finite automaton as a lexical analyzer, and recognize regular sets.
  - deterministic - faster recognizer, but it may take more space.
  - non deterministic - slower but may take less space
  - deterministic automatons are widely used lexical analyzers.

- The lexical analyzer has to recognize the longest possible string.
- The end of token is normally not defined.
- If the number of characters in a token is fixed, initial case no problem.
- Skipping comments.
- Symbol table interface
- Positions of the tokens in the file (for the error handling).

Q12. Construct precedence relation table for the following grammar and parse the string 101001.

Ans. $S \to 1S1 / 0S0 / 10$

Q13. Construct CLR parsing table for the following grammar.

$S \to Aa / bAc / dc / bda$

$A \to d$

Ans. Let us first number the productions as below:

1. $S \to Aa$
2. $S \to bAc$
3. $S \to dc$
4. $S \to bda$
5. $A \to d$

I_0 : $S' \to .S, \$$
$S \to .Aa, \$$
$S \to .bAc, \$$
$S \to .dc, \$$
$S \to .bda, \$$
$A \to .d, a$

I_1 : goto (I_0, S)
$S' \to S., \$$

I_2 : goto (I_0, A)
$S \to A.a, \$$

I_3 : goto (I_0, b)
$S \to b.Ac, \$$
$S \to b.da, \$$
$A \to .d, c$

Ans12: Finding leading and trailing

	LEADING()	TRAILING()
S	0,1	0,1

$<\cdot$ $1 <\cdot 1$ $0 <\cdot 1$ $\$ <\cdot 0, \$ <\cdot 1, 1 <\cdot 0, 0 <\cdot 0$

$\cdot>$ $1 \cdot> 1$ $0 \cdot> 1$ $1 \cdot> 0$ $0 \cdot> 0, 0 \cdot> 1, 0 \cdot> \$, 1 \cdot> \$$

$\doteq$ $1 \doteq 1, 0 \doteq 0, 1 \doteq 0$

The table are

	0	1	$
0	$\doteq / <\cdot / \cdot>$	$<\cdot / \cdot>$	$\cdot>$
1	$\doteq / <\cdot / \cdot>$	$\doteq / \cdot> / <\cdot$	$\cdot>$
$	$<\cdot$	$<\cdot$	

The above table has conflicts so it is not operator precedence grammar because due to conflict the parsing action cannot be completed. So no need to parse the string.

I_4 : goto (I_0, d)

$S \to d \cdot c$, \$

$A \to d \cdot$, a

I_5 : goto (I_2, a)

$S \to Aa \cdot$, \$

I_6 : goto (I_3, A)

$S \to bA \cdot c$, \$

I_7 : goto (I_3, d)

$S \to bd \cdot a$, \$

$A \to d \cdot$, c

I_8 : goto (I_4, c)

$S \to dc \cdot$, \$

I_9 : goto (I_6, c)

$S \to bAc \cdot$, \$

I_{10} : goto (I_7, a)

$S \to bda \cdot$, \$

Now, we will construct LALR parsing table.

State	Action					Goto	
	a	b	c	d	\$	S	A
0		s3		s4		1	2
1					Accept		
2	s5						
3				s7			6
4	r5		s8				
5							
6			s9				
7	s10		r5				
8					r3		
9					r2		
10					r4		

Give the syntax directed translation scheme to translate scheme to translate the switch case construct. Also write the three address code for the following code segment.

```
switch (a+b)
{ case 2: { x=y; break; }
  case 5: switch x
          {
            case0 : { a = b+1 ; break; }
            case1 : { a = b+3 ; break; }
            default : { a=2; break; }
          }
  case 9 : { x = y-1 ; break; }
  default : { a=2; break; }
}
```

Ans: Syntax directed translation scheme for switch case:

```
switch(E)
case V1 : S1
          break;
case V2 : S2
          break;
   :
case Vn : Sn
          break;
default : Sd
          break;
   :
end
```

Evaluate the E and store it in T

```
1.    T = E goto next
L1  : 3 address code for S1
      goto last
L2  : 3 address code for S2
      goto last
   :
Ln  : 3 address code for Sn
      goto last.
Ld  : 3 address code for Sd
      goto last
Next : if V1 = E goto L1
       if V2 = E goto L2
       if Vn = E goto Ln
       goto Ld.
last : end.
```

Three address code:

```
101  if a+b == 2  goto 107.
102  goto 103
103  if a+b == 5  goto 114.
104  goto 103
105  if a+b == 9 goto 110
106  goto 118 (Ld default)
107  T1 = y
108  x = T1
109  goto 129 (end)
110  T2 = y
111  T3 = y-1
```

```
112 : x = T3
113 : goto 129
114 : if x == 0 then goto 121
115 : goto 116
116 : if x == 1 then goto 124
117 : goto 129
118 : T4 = 2
119 : a = T4
120 : goto 129
121 : T5 = b + 1
122 : a = T5
123 : goto 129
124 : T6 = b + 3
125 : a = T6
126 : goto 129
127 : T7 = y - 1
128 : x = T7
129 : End.
```

} default case of switch 2

Q15 Consider the following grammar for addressing the array elements. Give the syntax directed translation scheme to convert into three address code.

$S \rightarrow L := E$

$E \rightarrow E + E \mid (E) \mid L$

$L \rightarrow id \mid Elist]$

$Elist \rightarrow E] \mid E, Elist$

Generate three address code for the following expression:

$A[I,J] = B[I,J] + C[A[K,L]] + D[I+J]$

Assume bpw = 4

(1) $T_1 = I * d_2$

(2) $T_2 = T_1 + J$

(3) $T_3 = T_2 * 4$

(4) $T_4 = B[T_3]$

(5) $T_5 = k * d_2$

(6) $T_6 = T_5 + L$

(7) $T_7 = T_6 * 4$

(8) $T_8 = A[T_7]$

(9) $T_9 = T_8 * 4$

(10) $T_{10} = C[T_9]$

(11) $T_{11} = T_4 + T_{10}$

(12) $T_{12} = I + J$

(13) $T_{13} = T_{12} * 4$

(14) $T_{14} = D[T_{13}]$

(15) $T_{15} = T_{11} + T_{14}$

(16) $T_{16} = I * d_2$

(17) $T_{17} = T_{16} * J$

(18) $T_{18} = T_{17} * 4$

(19) $A[T_{18}] = T_{15}$

$S \rightarrow L := E$ { if L.offset = null then
emit (L.place ':=' E.place);
else
emit (L.place '[' L.offset ']' ':=' E.place) }

$E \rightarrow E_1 + E_2$ { E.place := new temp ;
emit { E.place ':=' 'E1.place '+' E2.place }

$E \rightarrow (E_1)$ { E.place := E_1.place }

$E \rightarrow L$ { if L.offset = null
E.place := L.place
else begin
E.place := new temp
emit (E.place ':=' L.place ['L.offset'])
end }

$L \rightarrow Elist$ { L.place := newtemp ;
L.offset := new temp;
emit (L.place ':=' 'c' (Elist, anay) };
emit (L.offset ':=' Elist.place *
width (Elist.arry) }

$L \rightarrow id$ { L.place := id.place ;
L.offset := null }

(1) $T_1 = I * d2$
(2) $T_2 = T_1 + J$
(3) $T_3 = T_2 * 4$
(4) $T_4 = B[T_3]$
(5) $T_5 = K * d2$
(6) $T_6 = T_5 + L$
(7) $T_7 = T_6 * 4$
(8) $T_8 = A[T_7]$
(9) $T_9 = T_8 * 4$
(10) $T_{10} = C[T_9]$
(11) $T_{11} = T_4 + T_{10}$
(12) $T_{12} = I + J$
(13) $T_{13} = T_{12} * 4$
(14) $T_{14} = D[T_3]$
(15) $T_{15} = T_{11} + T_{14}$
(16) $T_{16} = I * d2$
(17) $T_{17} = T_{16} * J$
(18) $T_{18} = T_{17} * 4$
(19) $A[T_{18}] = T_{15}$

Section - C

C. Attempt all the parts.

16. Attempt anyone.

a) What are the lexical phase errors, syntactic phase errors and semantic phase errors, explain with suitable examples.

Ans. **LEXICAL PHASE ERRORS:**

- There are not many errors that can be ~~taught~~ caught at the lexical level, they are:

→ Characters that cannot appear in any token in our source language, such as @ or #.

→ Integer constants out of bounds.

→ Identifier names that are too long.

→ Text strings that ~~dee~~ span more than one line.

Lexical error recovery actions:

These errors can be detected during lexical analysis phase. Typical lexical errors (phase) are

- exceeding length of identifier or numeric constants.
- appearance of illegal characters.
- unmatched string

Error Recovery:

→ Panic mode error recovery - In this recovery mechanism successive characters from the remaining input are detected until the well formed token is formed.

→ If any unwanted character occurs then delete that character to recover from error.

→ If any unmatched string occurs then insert appropriate string / character in order to match the string

SYNTAX ERRORS:

- It occurs when stream of tokens is an invalid string.
- In LL(k) or LR(k) parsing tables, blank entries refer to syntax error.

Semantic error : The semantic errors are logical errors which are not syntactic form but the program will not run. The main semantic feature is type checking.

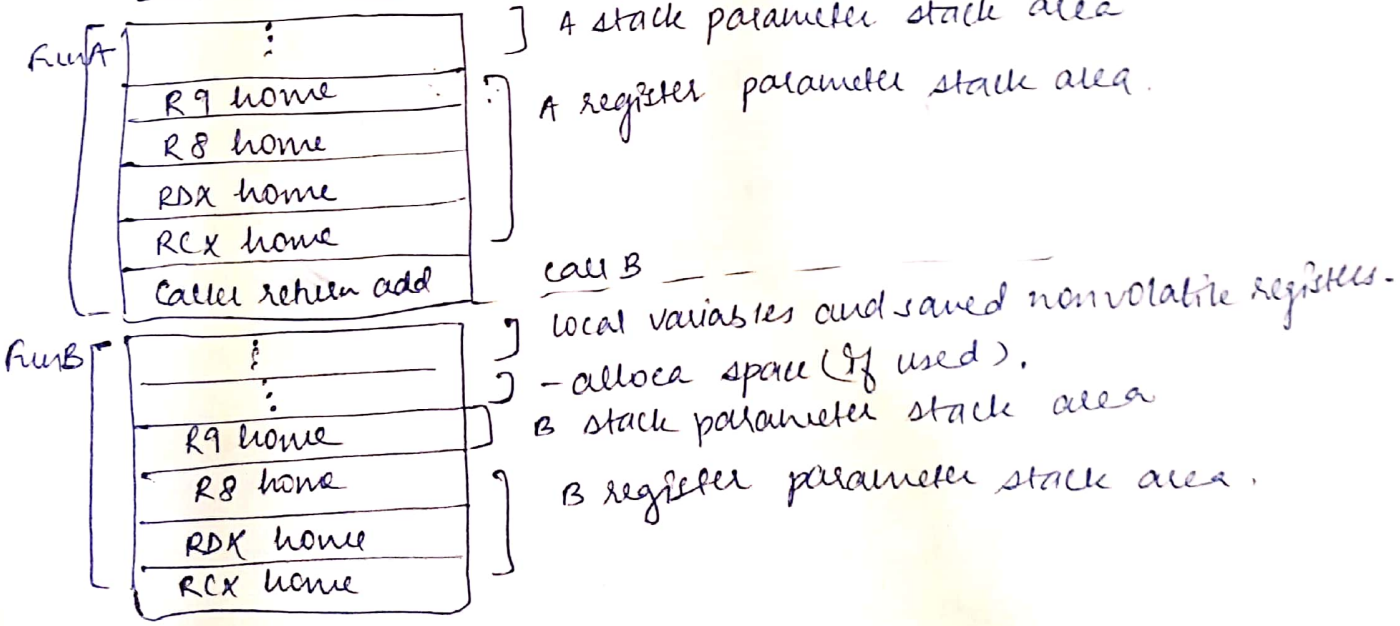
Type checking:
- Types :
  - Simple types
  - Structured types: array, struct, union, pointer
- Type equivalence
- Type checking
- Type conversion
- Overloading.

(b) Why runtime storage management is required ? How simple stack implementation is implemented ?

Runtime Environments :

- Places for some of data objects will be allocated at runtime.
- Data objects that can be determined at compile time will be allocated statistically.
- Allocation of de-allocation of the data objects is managed by the runtime support package.
- Each execution of a procedure is called as activation of that procedure.

Stack Allocation :



Q17. Attempt anyone.

a) What are the different data structures used for symbol table? How is the scope information is represented in symbol table.

Ans Data structures for symbol tables.

- Linear lists
- Trees
- Hash Tables

* LINEAR LISTS:

→ [Name | Info] → [Name | Info] → [Name | Info]

→ Search: search linearly from beginning to end. stop if found.

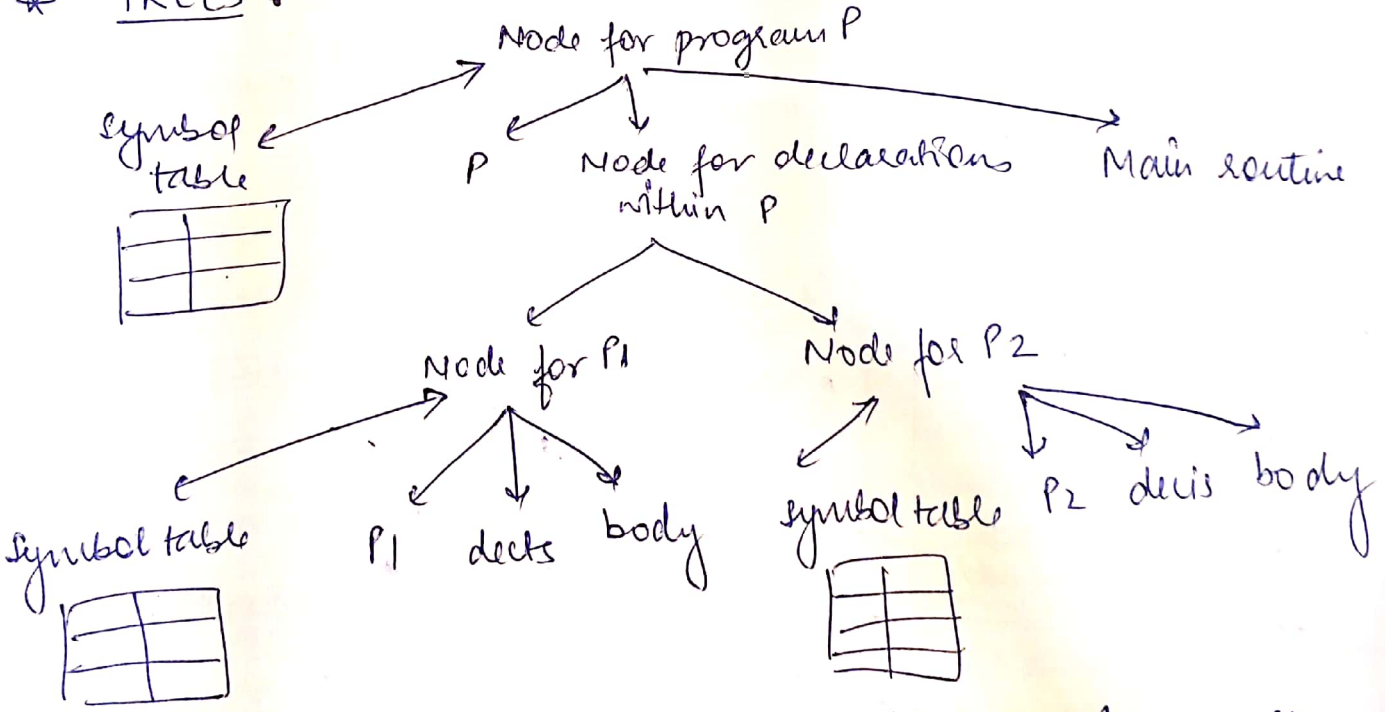
→ Adding: search. Add at beginning if not found.

→ Effectively: To insert n names and search for m names the cost will be $cn(n+m)$ comparisons. Inefficient.

→ Positive: Easy to implement, use little space

→ Negative: Slow for large n and m.

* TREES:



- Each subprogram has a symbol table associated to its node in the abstract syntax tree.
- Quicker than linear lists.
- Easy to represent scoping.

HASH TABLES (WITH CHAINING)

- Search
Hash the name in a hash function,
$h(\text{symbol}) \in [0, k-1]$
where k = table size
If the entry is occupied, follow the link field.

Positives: very quick search

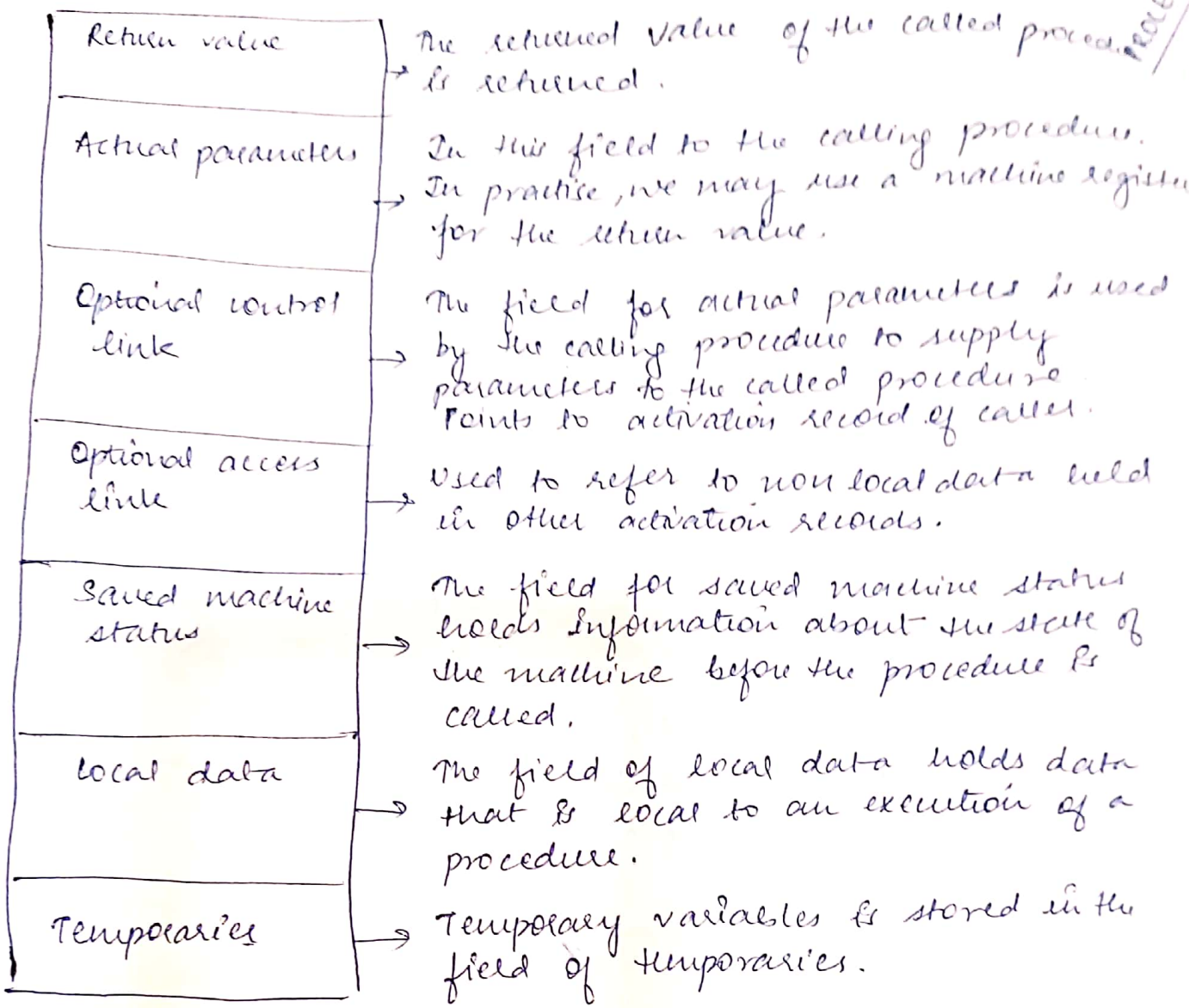
Negative:
- Relatively complicated
- Extra space required, k words for hash table.
- More difficult to introduce scoping.

ex:
```
program prog;
var a, b, c, integer;
procedure p1;
var b, c, real;
  procedure p2;
  var c, real;
  begin;
    c := b + a;
  end;
begin
    c := b + a;
end;
begin
...
end.
```

(b) What is activation record? Explain its organization. Discuss how access links are used to access non-local names.

Ans ACTIVATION RECORDS:

Information needed by a single execution of a procedure is managed using a contiguous block of storage called activation record. An activation record is allocated when a procedure is entered, and it is deallocated when that procedure exited.

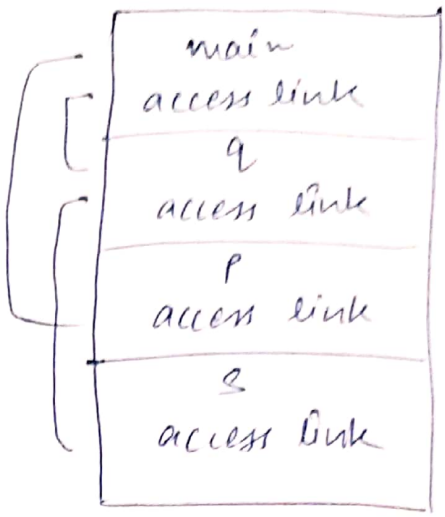



```
Program main;
 var a: int;
 procedure p;
  var d : int;
  begin a:=1; end;
procedure q(i:int);
  var b: int;
  procedure s;
  var c : int;
  begin p; end;
 begin
 if i<>0 then q(i-1)
   else s;
end; begin q(1); end;
```

ACCESS LINKS

PROCEDURE PARAMETERS:

```
program main;
procedure p (procedure a);
 begin a; end;
procedure q;
procedure s;
begin ... end;
begin p(s) end;
begin q; end;
```



Q18. Attempt anyone.

a) What is DAG? Give algorithm for DAG construction. Draw DAG for the following expression.

(a+b) - (e-(c+d)).

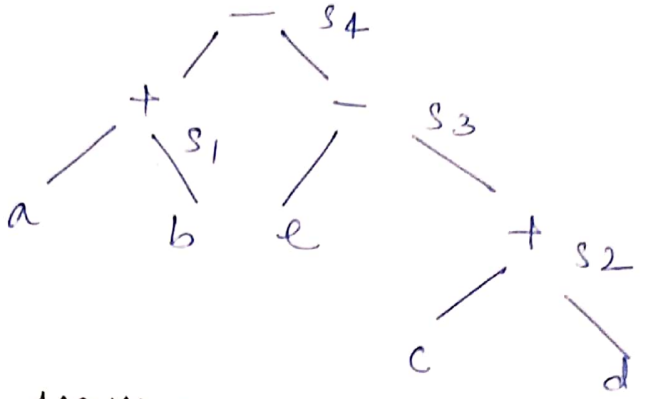
A DAG is a directed graph with no cycles which gives a picture how the value computed by each statement in a basic block is used in subsequent statements in the block.

CONSTRUCT A DAG FOR A BASIC BLOCK:

- There is a node in the DAG for each of the initial values of the variables appearing in the basic block.
- Node N is associated with each statement within the block. Children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.
- Node N is labelled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block.
- Certain nodes are designated output nodes. These are the nodes whose variables are live on exit from the block; that is, their values may be used later, in another block of the flow graph.

The triples of the given expression is:

S1 = a+b
S2 = c+d
S3 = e - S2
S4 = S1 - S3



(b) Consider the following sequence of three address code.

1. Prod := 0
2. I := 1
3. T1 : 4 * I
4. T2 : addr(A) - 4
5. T3 : T2[T1]
6. T4 : addr(B) - 4
7. T5 : T4[T1]
8. T6 : T3 * T5
9. Prod := Prod + T6
10. I = I+1
11. If I <= 20 goto(3)

Perform loop optimization.

Initially we find the basic blocks for the above code on the basis of the leader statement. The leader statements are 1 and 3. So the two basic blocks are generated for the above code fragment.

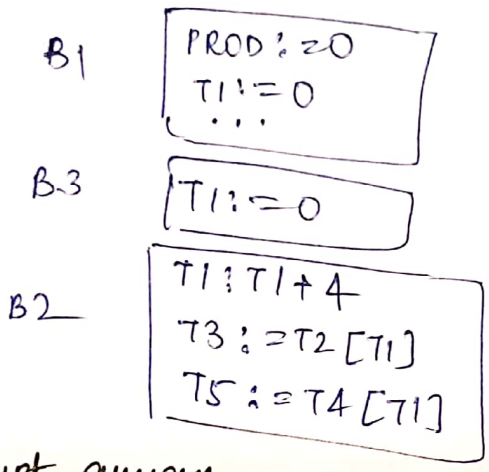
B1
```
PROD := 0
I := 1
```

B2
```
T1 := I*4
T3 := T2[T1]
T5 := T4[T1]
T6 := T3 * T6
```

The above two basic blocks are found after analysing the leaders from the above code. The induction variables are found in the above block B2.

We eliminate I from the block B2 and convert the T1 = 4*I into T1 := T1 + 4.

2 Advantages are achieved by doing this change.
One variable I is reduced and the * means costly operation is replaced by cheaper operation that is +. That is called the reduction in strength. So the resulting optimized blocks are:



Q19. Attempt anyone.

(a) How the global data flow analysis is helpful in code optimization.

DATA FLOW ANALYSIS:

It is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's CFG is used to determine those parts of a program to which a particular value assigned to a variable might propagate.
It is the process of collecting information about the way the variables are used, defined in the program.
Data flow analysis attempts to obtain particular information at each point in a procedure.
The entry state of a block is a function of the exit states of its predecessors. This yields a set of dataflow equations;

For each block b:

$$Out_b = trans_b\ (in_b)$$

$$in_b = join_{p \in pred_p}\ (Out_p)$$

In this, $trans_b$ is the transfer function of the block b. It works on the entry state in b, yielding the exit state out. The join operation join combines the exit states of the predecessors $p \in pred_b$ of b, yielding the entry state of b.

(b) Write short notes (any two):

(i) Constant folding:

~~loop~~ constant folding is an optimization technique that eliminates expressions that calculate a value that can already be determined before code execution. These are typically calculations that only reference constant values or expressions that reference variables whose values are constant.

ex: i <- 320 * 200 * 32

Most compilers would not actually generate two multiply instructions. Instead, they identify constructs such as these and substitute the computed values. The code will be replaced by:

i <- 2048500

(ii) Loop unrolling:

Loop unrolling involves the programmer analyzing the loop and iterations into a sequence of instructions which will reduce the loop overhead. This is in contrast to dynamic unrolling which is accomplished by the compiler.

ex:

Normal loop	After loop unrolling

int x;
for(x=0; x<4; x++)
{
delete(x);
}
//.
//.
//.
//.
	int x;
	for(x=0; x<100; x+=5)
	{
	delete(x);
	delete(x+1);
	delete(x+2);
	delete(x+3);
	delete(x+4);
	}

(iii) **Loop Jamming:**

Loop fusion also called loop jamming, is a compiler optimization, a loop transformation, which replaces multiple loops with a single one.

example in C

```
int i, a[100], b[100];
for(i=0; i<100; i++)
  a[i] = 1;
for(i=0; i<100; i++)
  b[i]=2;
```

is equivalent to:

```
int i, a[100], b[100];
for (i=0; i<100; i++)
{
  a[i] = 1;
  b[i] = 2;
}
```

Q20. Attempt anyone.

a) Give the syntax directed translation scheme to translate the while control construct. Write the three address code for the following code segment:

```
while a<c and b<d do
    if a=1 then c=c+1
    else while a<=d do a=a+2.
```

SDT scheme for while construct:

while E do S1 : S.begin = newlabel();
S.after = newlabel();
S.code = gen(S.begin ':') || E.code ||
gen('jmpf' E.place ',' S.after)
|| S1.code || gen('jmp' ',' S.begin) || gen(S.after ':')

if E then S1 else S2 S.else = newlabel();
S.after = newlabel();
S.code = E.code || gen('jmpf' E.place ',' S.else)
|| S1.code || gen('jmp' ',' S.after) ||
gen(S.else ':') || S2.code || gen(S.after ':')

Three address code:

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

b). What is 3-address code? Explain types of 3-address code. Convert following expressions into quadruple, triple and indirect triples.

S = (a+b)/(c-d) * (e+f)

Three address code will be the intermediate representation used in our decaf compiler. It is essentially a generic assembly language that falls in the lower end of the mid-level IRs. Some variant of 2, 3 or 4 address code is fairly common used as an IR.

ex: A TAC is:

x := y op z

where x, y and z are names, constants or compiler-generated temporaries, op is any operator.

The implementations are of three types:

1) Quadruples
2) Triples
3) Indirect triples

consider the example:

$-(a+b)+(c+d)-(a+b+c)$

The TAC is:

$T1 := a+b$

$T2 := T1$

$T3 := c+d$

$T4 := T2+T3$

$T5 := T1+c$

$T6 := T4-T5$

Quadruples: Quadruples contain OP, ARG1, ARG2 and RESULT. The above given TAC is represented into Quadruples in following table.

	OP	ARG1	ARG2	RESULT
(0)	+	a	b	T1
(1)	Unimus	T1		T2
(2)	+	c	d	T3
(3)	+	T2	T3	T4
(4)	+	T1	c	T5
(5)	-	T4	T5	T6

Triples:

	OP	ARG1	ARG2	RESULT
(0)	+	a	b	T1
(1)	Unimus	T1		T2
(2)	+	c	d	T3
(3)	+	T2	T3	T4
(4)	+	T1	c	T5
(5)	-	T4	T5	T6

Indirect Triples: This representation contains triples and one pointer which contain the address of the triples. That's why it is called indirect triples.

	OP	ARG1	ARG2	RESULT
(0)	+	a	b	T1
(1)	Uminus	T1		T2
(2)	+	c	d	T3
(3)	+	T2	T3	T4
(4)	+	T1	c	T5
(5)	−	T4	T5	T6

The given expression is:

$S = (a+b)/(c-d) * (e+f)$

The 3 address code is

$T1 = a + b$

$T2 = c - d$

$T3 = T1 / T2$

$T4 = e + f$

$T5 = T3 * T4$

$S = T5$

Quadruple:

	OP	ARG1	ARG2	RESULT
1	+	a	b	T1
2	−	c	d	T2
3	/	T1	T2	T3
4	+	e	f	T4
5	*	T3	T4	T5
6	=	T5		S

Triple:

	OP	ARG1	ARG2
1	+	a	b
2	−	c	d
3	/	1	2
4	+	e	f
5	*	3	4
6	=	5	

~~RESULT~~

Indirect Triple:

	OP	ARG1	ARG2
1	+	a	b
2	−	c	d
3	/	101	102
4	+	e	b
5	*	103	104
6	=	105	

	Statement
1	101
2	102
3	103
4	104
5	105
6	106