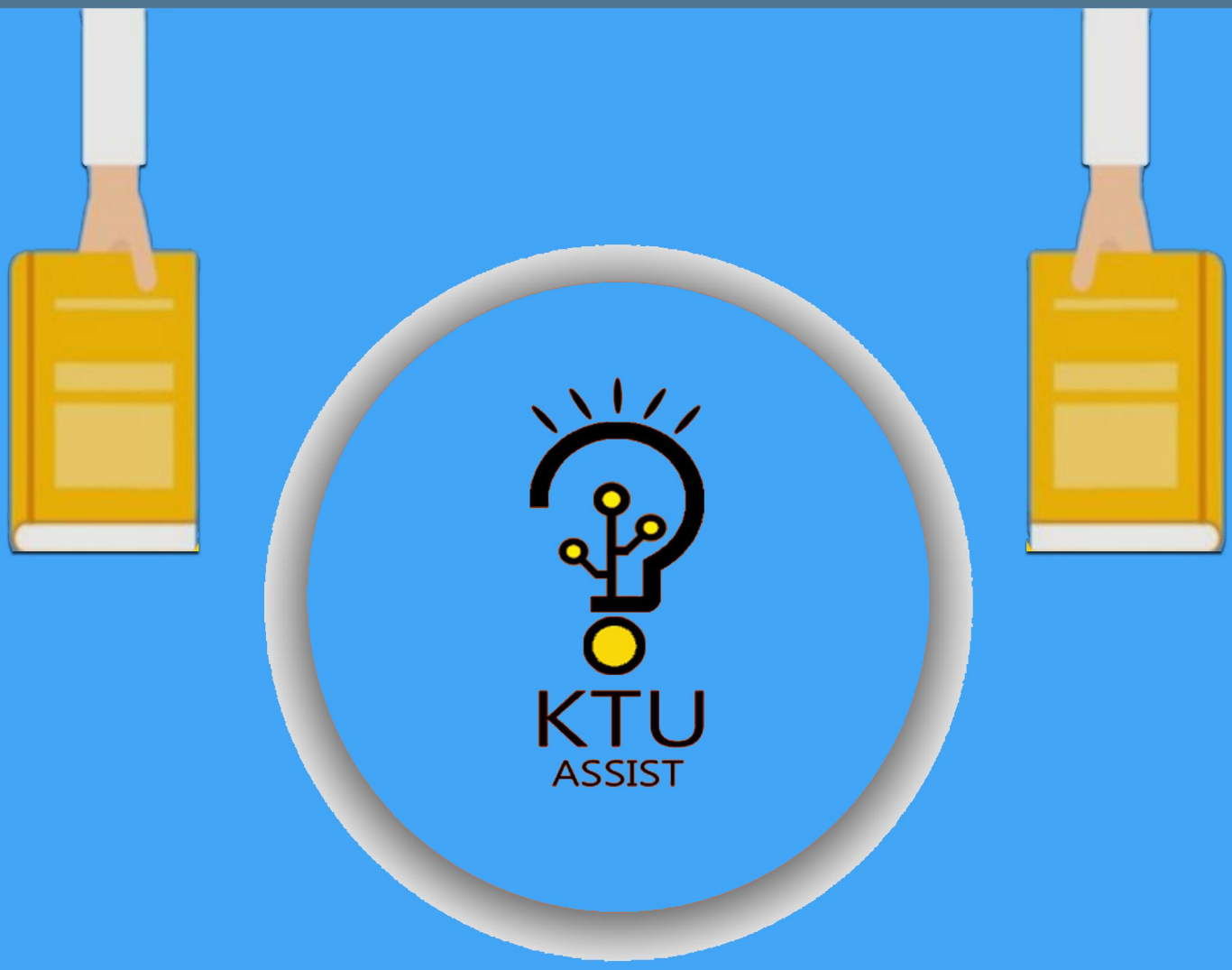
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

STUDY MATERIALS



Reg. No:_____ Name:_____

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY
SIXTH SEMESTER B.TECH DEGREE MODEL EXAMINATION, MARCH 2018
CS304 COMPILER DESIGN

Max. Marks: 100 **Duration: 3 hours**

PART A
(Answer all questions.Each carries 3 marks.)

1.What advantages are there to a language processing system in which the compiler produces assembly language rather than machine language. (3)
2.Describe the languages denoted by the following regular expressions. (3)

1.a(a|b)*a 2.((ε|a)b*)* 3.(a|b)*a(a|b)(a|b)

3.Consider the context-free grammar: S -> S S + | S S * | a and the string aa + a*. (3)

1.Give a leftmost derivation for the string.
2.Give a rightmost derivation for the string.
3. Give a parse tree for the string.

4.Design grammars for the following languages: (3)
1. The set of all strings of 0s and 1s that are palindromes; that is, the string reads the same backward as forward.
2.The set of all strings of 0s and 1s with an equal number of 0s and 1s.

PART B
(Answer any two questions. Each carries 9 marks.)

5.a) Explain the phases of a compiler. (6)
b)Calculate the number of tokens in the following C statement
printf("i = %d, &i = %x", i, &i); (2)

6.Construct NFA for the regular expression a(a|b)*a . And convert it to DFA. (9)
7.a) Consider the grammar, remove left recursion. (3)
S →A a | b
A → A c | S d | ε

b) Compute FIRST and FOLLOW
S -> ACB|Cbb|Ba A -> da|BC B-> g|Є C-> h| Є (4)

c) S--> a | ab | abc | abcd | e | f Perform left factoring. (2)

PART C
(Answer all questions.Each carries 3 marks.)

8.Explain Handle pruning. (3)
9.What is operator precedence grammar?.Give examples. (3)
10.Explain the applications of syntax directed translation (3)
11.Define synthesized and inherited translation. (3)

PART D
(Answer any two questions. Each carries 9 marks.)

12.Show that the following is LR(1) but not LALR(1) (9)

S -> A a | b A c | B c | b B a

A -> d

B -> d

13.a) Write the syntax directed definition of a simple desk calculator. (6)

b)Describe all the viable prefixes for the grammar S->0S1|01 (3)

14. a) Describe SDT for infix-to-prefix translation (3)

b) Describe the conflicts in shift reduce parsing (3)

c) Compute leading and trailing for the following grammar (3)

S---> a | ^ | (T)

T----> T, S | S

PART E
(Answer any four questions. Each carries 10 marks.)

15.a) Explain with example how three address code is partitioned to basic blocks (4)

b)Explain peephole optimization (6)

16.What are the issues in the design of a code generator. (10)

17.a) Construct the DAG for the expression. (4)

((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))

b) Translate the arithmetic expression a+-(b+c) into (6)

a)Syntax tree b)Quadruples c)Triples d)Indirect triples

18.Write a note on the translation of boolean expression. (10)

19.Explain optimization of basic blocks . (10)

20.Discuss the different storage allocation strategies. (10)

Reg. No:_____ Name:_____

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY
SIXTH SEMESTER B.TECH DEGREE MODEL EXAMINATION, MARCH 2018
CS304 COMPILER DESIGN

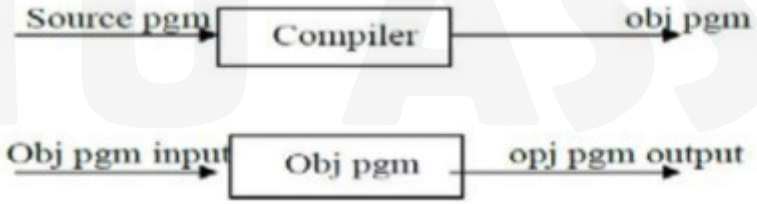
Max. Marks: 100 Duration: 3 hours

Answer key

PART A
(Answer all questions.Each carries 3 marks.)

1.What advantages are there to a language processing system in which the compiler produces assembly language rather than machine language. (3)

Compiler is a translator program that translates a program written in (HLL) the source program andtranslate it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.Executing a program written n HLL programming language is basically of two parts.The source program must first be compiled translated into a object program. Then the results object program is loaded into a memory executed.



The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug.Programmers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

2. Describe the languages denoted by the following regular expressions. (3)

1.a(a|b)*a 2.((ε|a)b*)* 3.(a|b)*a(a|b)(a|b)

1.String of a's and b's that start and end with a.
2.String of a's and b's.

3.String of a's and b's that the character third from the last is a.

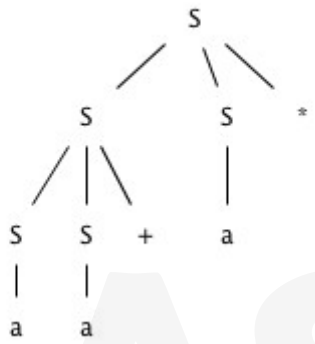
3.Consider the context-free grammar: S -> S S + | S S * | a and the string aa + a*. (3)

1.Give a leftmost derivation for the string.
2.Give a rightmost derivation for the string.
3. Give a parse tree for the string.

1.S =lm=> SS* => SS+S* => aS+S* => aa+S* => aa+a*

2.S =rm=> SS* => Sa* => SS+a* => Sa+a* => aa+a*

3.



4.Design grammars for the following languages: (3)

1. The set of all strings of 0s and 1s that are palindromes; that is, the string reads the same backward as forward.

2.The set of all strings of 0s and 1s with an equal number of 0s and 1s.

1.S -> 0S0 | 1S1 | 0 | 1 | ε

2.S -> 0S1S | 1S0S | ε

PART B

(Answer any two questions. Each carries 9 marks.)

5.a) Explain the phases of a compiler. (6)

Phases of a compiler: A compiler operates in phases. A phase is a logically interrelated operation
that takes source program in one representation and produces output in another representation. Thephases of a compiler are shown in below

There are two phases of compilation.

a. Analysis (Machine Independent/Language Dependent)
b. Synthesis(Machine Dependent/Language independent)

Compilation process is partitioned into no-of-sub processes called ‘phases’.

Lexical Analysis:-

LA or Scanners reads the source program one character at a time, carving the source program into a sequence of automic units called tokens.

Syntax Analysis:-

The second stage of translation is called Syntax analysis or parsing. In this phase expressions,statements, declarations etc... are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language.

Intermediate Code Generations:-

An intermediate representation of the final machine language code is produced. This phase bridgesthe analysis and synthesis phases of translation.

Code Optimization :-

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

Code Generation:-

The last phase of translation is code generation. A number of optimizations to reduce the length of machine language program are carried out during this phase. The output of the code generator is the machine language program of the specified computer.

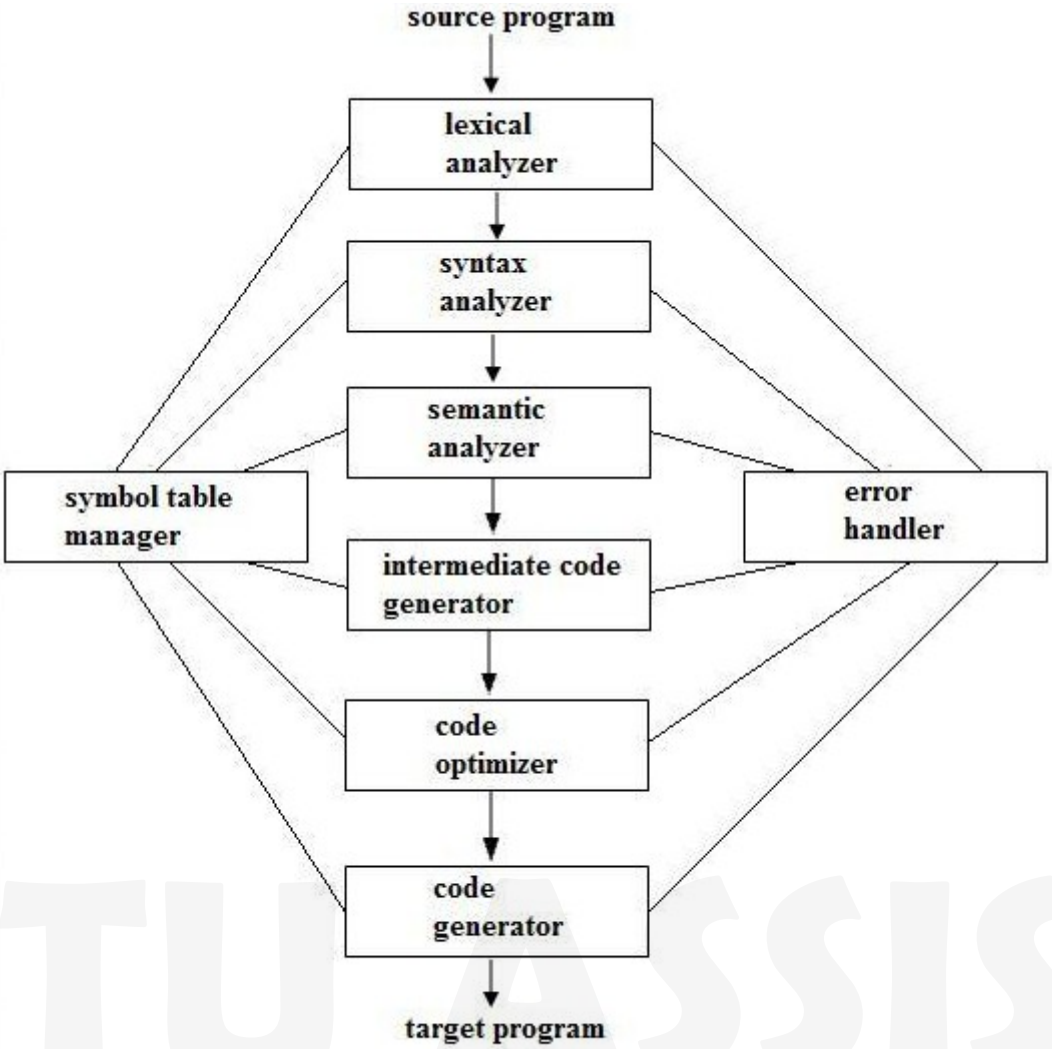
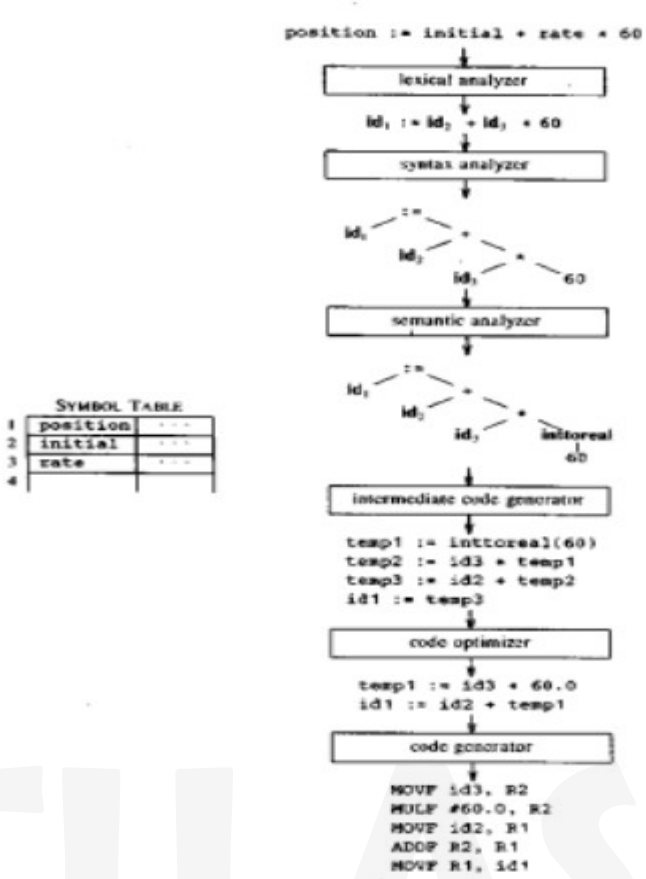


Fig 1.5 Phases of a compiler



5 b)Calculate the number of tokens in the following C statement

printf("i = %d, &i = %x", i, &i); (2)

No of Tokens-10

1.printf

2.(

3.“i=%d,&i=%x”

4.,

5.i

6.,

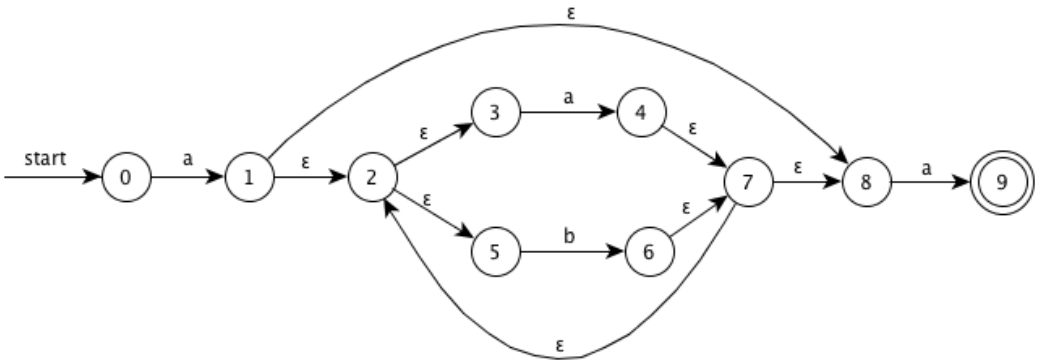
7.&

8.i

9.)

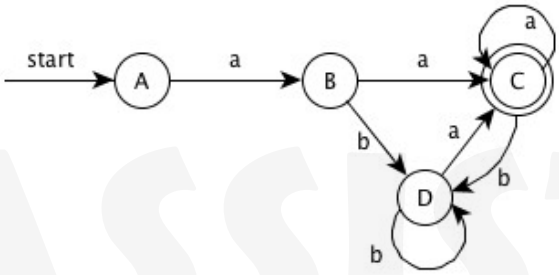
10.;

6.Construct NFA for the regular expression a(a|b)*a . And convert it to DFA. (9)



DFA:

NFA	DFA	a	b
{0}	A	B	
{1,2,3,5,8}	B	C	D
{2,3,4,5,7,8,**9**}	**C**	C	D
{2,3,5,6,7,8}	D	C	D



7.a) Consider the grammar, remove left recursion. (3)

S → A a | b

A → A c | S d | ε

S → A a | b

A → b d A' | A'

A' → c A' | a d A' | ε

b) Compute FIRST and FOLLOW

S -> ACB|Cbb|Ba A -> da|BC B-> g|Є C-> h| Є (4)

FIRST(S) = FIRST(A) U FIRST(B) U FIRST(C) = { d, g, h, Є, b, a}

FIRST(A) = { d } U FIRST(B) = { d, g, h, Є }
FIRST(B) = { g, Є }
FIRST(C) = { h, Є }

FOLLOW Set
FOLLOW(S) = { $ }
FOLLOW(A) = { h, g, $ }
FOLLOW(B) = { a, $, h, g }
FOLLOW(C) = { b, g, $, h }

c) S--> a | ab | abc | abcd | e | f Perform left factoring. (2)

$S \longrightarrow a\,S' / e / f$

$S' \longrightarrow b\,S'' / \varepsilon$ - For single **a**

$S'' \longrightarrow c\,S''' / \varepsilon$ - For **ab**

$S''' \longrightarrow d / \varepsilon$ - For **abc**

PART C
(Answer all questions.Each carries 3 marks.)

8.Explain Handle pruning. (3)

Bottom-up parsing during a left-to -right scan of the input constructs a rightmost derivation in reverse.A handle is a substring that matches the body of a production and whose reduction represents one step along the reverse of a rightmost derivation. A rightmost derivation in reverse can beobtained by handle pruning.That is ,we start with a string of terminals w to be parsedIf w is a sentence of the grammar at hand,then finally reach to the start symbol.

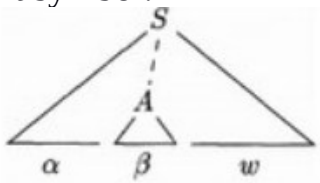


Figure 4.27: A handle $A \to \beta$ in the parse tree for $\alpha\beta w$

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w.$$

9.What is operator precedence grammar?.Give examples. (3)

An operator precedence grammar is a context-free grammar that has the property (among others) that no production has either an empty right-hand side or two adjacent nonterminals in its right-hand side. These properties allow precedence relations to be defined between the terminals of the grammar.
The grammar
E->EAE | (E) | -E | id
A-> + | - | * | / | ↑
is not an operator grammar, because the right side EAE has two (in fact three) consecutive nonterminals;
However, if we substitute for A each of its alternatives, we obtain the following operator grammar:
E->E+E | E-E | E*E | E/E | E↑E | (E) | -E | id

10.Explain the applications of syntax directed translation (3)

Syntax-directed translation (SDT) refers to a method of compiler implementation where the source language translation is completely driven by the parser, i.e., based on the syntax of the language The parsing process and parse trees are used to direct semantic analysis and the translation of the source program. Almost all modern compilers are syntax-directed. SDT can be a separate phase of a compiler or we can augment our conventional grammar with information to control the semantic analysis and translation. Such grammars are called attribute grammars.
Applications of Syntax-Directed Translation

1 Construction of Syntax Trees

2 The Structure of a Type

11.Define synthesized and inherited translation. (3)

Synthesized Attributes

A SDD defines zero or more *attributes* for each nonterminal and terminal. A *synthesized attribute* has its value defined in terms of attributes at its children.

Example:

A production $A \rightarrow BC$ and a rule like

$$A.att := f(B.att1,\ B.att2,\ C.att3)$$

makes $A.att$ a synthesized attribute.

- It is conventional to call the attributes of terminals, which are generally lexical values returned by the lexical analyzer, "synthesized."
- A SDD with only synthesized attributes is called an *S-attributed definition*. All the examples seen so far are S-attributed.

Implementing S-attributed Definitions

It is easy to implement an S-attributed definition on an LR grammar by a postfix SDT.

- Values of attributes for symbol X are stored along with any occurrence of X on the parsing stack.
- When a reduction occurs, the values of attributes for the nonterminal on the left are computed from the attributes for the symbols on the right (which are all at the top of the stack), before the stack is popped and the left side pushed onto the stack, along with its attributes.

Infix to postfix conversion

$$E.post := E_1.post \mid T.post \mid \text{'+'}$$

which can be turned into SDT

$$E \rightarrow E + T \{ \textbf{print } \text{'+'} \}$$

In principle, if we wanted to translate to prefix, we could blithely write the SDD with rules like:

$$E.pre = \text{'+'} \mid E_1.pre \mid T.pre$$

The corresponding SDT, with rules like

$$E \rightarrow \{ \textbf{print } \text{'+'} \}\ E + T$$

is legal, but *cannot be executed as written*, because the grammar will not let the parser know when to execute the print actions. Rather, it must be implemented as discussed previously: build the parse tree and then traverse it in preorder to execute the actions.

Inherited Attributes

Any attribute that is not synthesized is called *inherited*.

- The typical inherited attribute is computed at a child node as a function of attributes of its parent.
- It is also possible that attributes at sibling nodes (including the node itself) will be used.

Example:

Consider the grammar with nonterminals

1. D = type definition.
2. T = type (integer or real).
3. L = list of identifiers.

and SDD

$D \rightarrow T\ L$	
	$L.type := T.type$
$T \rightarrow int$	
	$T.type :=$ INT
$T \rightarrow real$	
	$T.type :=$ REAL
$L \rightarrow L_1\ ,\ id$	
	$L_1.type := L.type;$
	addtype($id.entry,\ L.type$)
$L \rightarrow id$	
	addtype($id.entry,\ L.type$)

Notes

- The call to *addtype* is a side-effect of the SDD. The timing of the call is not clear, but it must take place at least once for every occurrence of the last two productions in the parse tree.
- We shall discuss "L-attributed definitions," where there is a natural order of evaluation, making ambiguities like this one disappear.
- We assume that terminal *id* has an attribute *entry*, which is a pointer to the symbol table entry for that identifier. The effect of *addtype* is to enter the declared type for that identifier.

PART D
(Answer any two questions. Each carries 9 marks.)

12.Show that the following is LR(1) but not LALR(1) (9)

S -> A a | b A c | B c | b B a

A -> d

B -> d

S -> Aa | bAc | Bc | bBa

A -> d

B -> d

~~~~~~

LR(1) Parser

State 0: [S -> *Aa , $]
[S -> *bAc , $]
[S -> *Bc , $]
[S -> *bBa , $]
[A -> *d , a]
[B -> *d , c]

State 1: Goto(State 0, d) = [A -> d* , a]
[B -> d* , c]

State 2: Goto(State 0, b) = [S -> b*Ac , $]
[S -> b*Ba , $]
[A -> *d , c]
[B -> *d , a]

State 3: Goto(State 2, d) = [A -> d* , c]
[B -> d* , a]

~~~~~~

LALR(1) Parser

Merge lookaheads for State 1 and State 3 in LR(1) parser to

create new state:

State = Merge (State 1, State 3) = [A -> d* , a]
[A -> d* , c]
[B -> d* , c]
[B -> d* , a]

Reduce/reduce conflict for lookahead "a" and "c"
(i.e., can't decide whether to reduce "d" to A or B)

13.a) Write the syntax directed definition of a simple desk calculator. (6)

A Syntax Directed Definition(SDD) is a context-free grammar together with attributes and rules.Attributes are associated with grammar symbols and rules are associated with productions.If X is a symbol and a is one of its attributes,then we write X.a to denote the value of a at a particular parse-tree node labelled X.If we implement the nodes of the parse tree by records or objects, then the attributes of X can be implemented by data fields in the records that represent the nodes for X. The attributes are evaluated by the semantic rules attached to the productions.

Example:

PRODUCTION	SEMANTIC RULE
E → E1 + T	E.code = E1.code || T.code || '+'

SDDs are highly readable and give high-level speif ications for translations. But they hide many implementation details.

Production	Semantic Rules
$L \rightarrow E$ **n**	*print* (*E.val*)
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val + F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$ **digit**	$F.val :=$ **digit**.*lexval*

Fig. 5.2. Syntax-directed definition of a simple desk calculator

b) Describe all the viable prefixes for the grammar S->0S1|01 (3)
The prefixes of right sentential forms that can appear on the stack of a shift - reduce parser are called viable prefixes.By definition, a viable prefix is a prefix of a right sentential form that does
not continue past the right end of the rightmost handle of that sentential form.

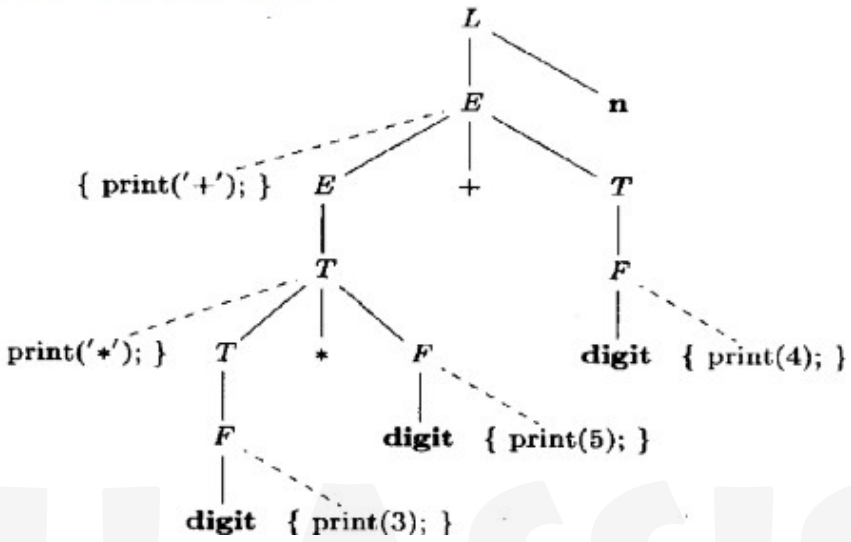
Stack	Input	Action
$	000111$	Shift
$0	00111$	Shift
$00	0111$	Shift
$000	111$	Shift
$0001	11$	Reduce S->01
$00S	11$	Shift
$00S1	1$	Reduce S->0S1
$0S	1$	Shift
$0S1	$	Reduce S->0S1
$S	$	Accept

Stack always contains viable prefixes.

14. a) Describe SDT for infix-to-prefix translation (3)

1) $L \rightarrow E$ **n**
2) $E \rightarrow$ { print('+'); } $E_1 + T$
3) $E \rightarrow T$
4) $T \rightarrow T_1 * F$ { print('*'); }
5) $T \rightarrow F$
6) $F \rightarrow (E)$
7) $F \rightarrow$ **digit** { print(**digit**.*lexval*); }

Parse Tree with Actions Embedded



b) Describe the conflicts in shift reduce parsing (3)

There are context -free grammars for which shift-reduce parsing cannot be used.Every shift -reduce parser for such a grammar can reach configuration in which the parser,knowing the entire stack and also the next k input symbols ,cannot decide whether to shift or to reduce (a shift/reduce conflict),or cannot decide which of several reductions to make (a reduce/reduce conflict).

c) Compute leading and trailing for the following grammar (3)

S---> a | ^ | (T)
T----> T, S | S

The algorithm for finding LEADING(A) where A is a non-terminal is given below
LEADING(A)
{
1. 'a' is in Leading(A) if A $\prec$ γaδ where γ is ε or any Non-Terminal
2. If' 'a' is in Leading(B) and A $\prec$ Bα, then a in Leading(A)
}
Step 1 of algorithm indicates how to add the first terminal occurring in the RHS of everyproduction directly. Step 2 of the algorithm indicates to add the first terminal, through another non-terminal B to be included indirectly to the LEADING() of every non-terminal.Similarly the algorithm to find TRAILING (A) is given below.
TRAILING (A)

{
1. a is in Trailing(A) if A $\succ$ γaδ where δ is ε or any Non-Terminal
2. If a is in Trailing(B) and A $\succ$ αB, then a in Trailing(A)
}
Leading(S)={a,^,(, ,)
Leading(T)={,}
Trialing(S)={a,^,),,}
Trailing(T)={,}

PART E

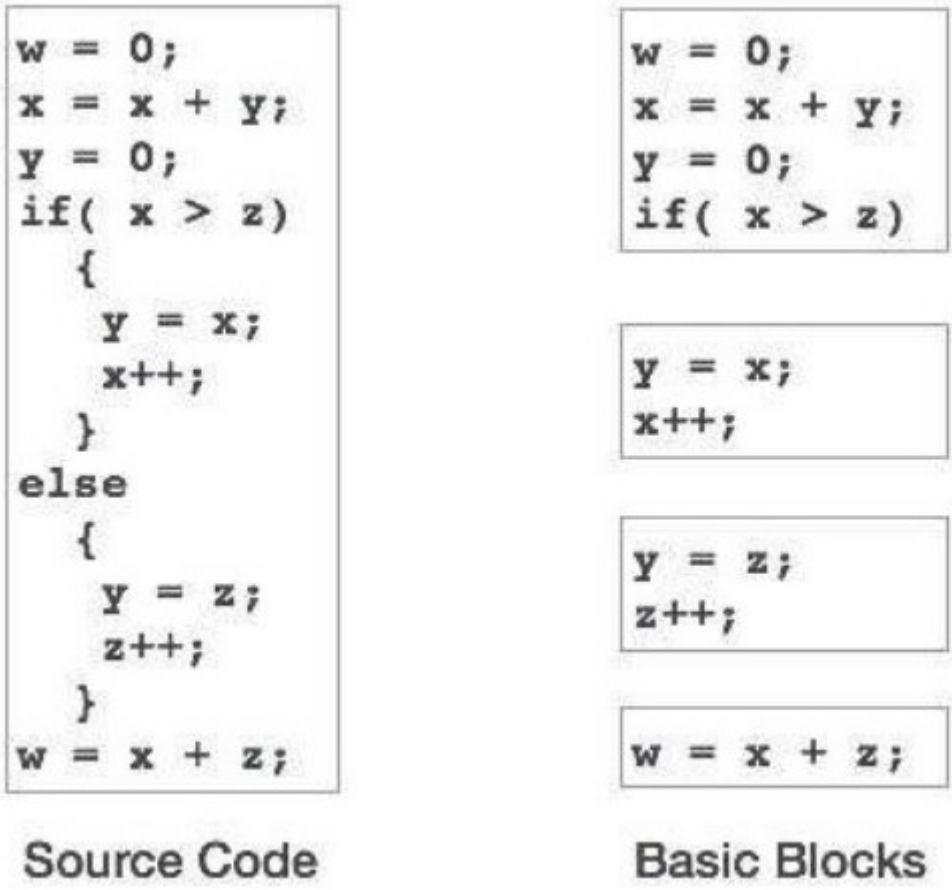
(Answer any four questions. Each carries 10 marks.)

15.a) Explain with example how three address code is partitioned to basic blocks (4)

A program flow graph is also necessary for compilation. the nodes are the basic blocks. There is an arc from block B1 to block B2 if B2 can follow B1 in some execution sequence.

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

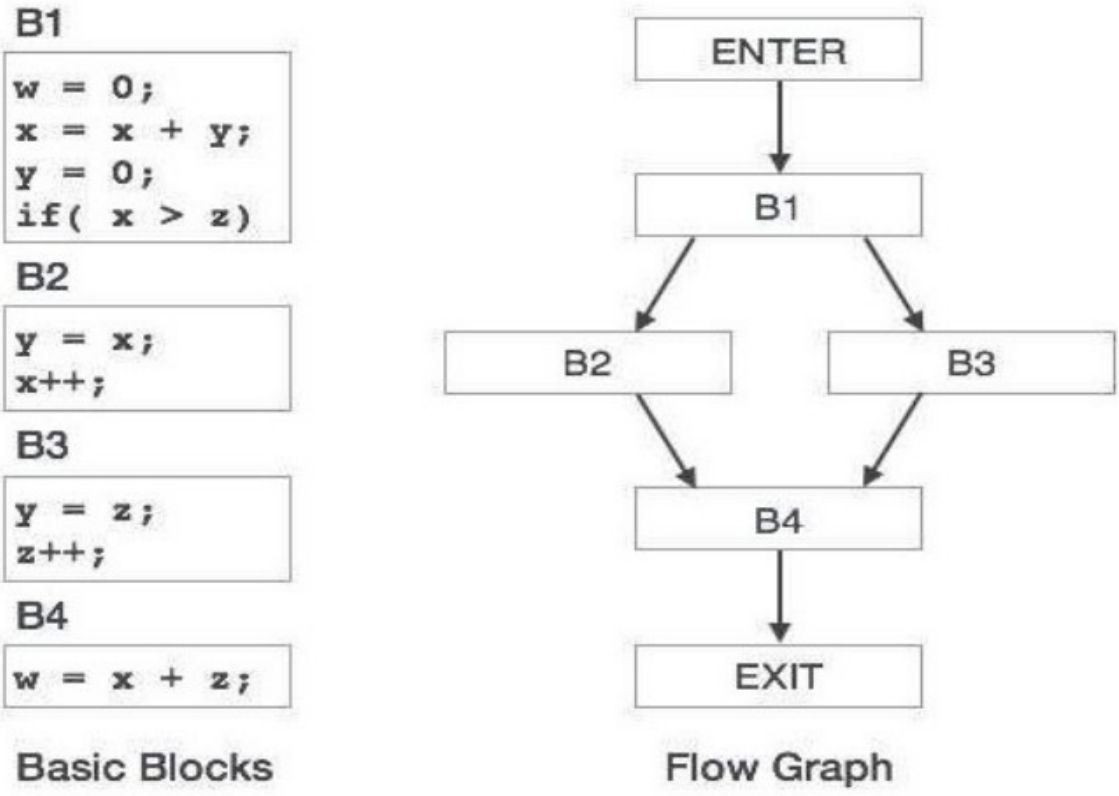
Basic blocks are important concepts from both code generation and optimization point of view.



Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block. If any variable is being used more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

Control Flow Graph

Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization by help locating any unwanted loops in the program.



b)Explain peephole optimization (6)

A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying "optimizing" transformations to the target program.

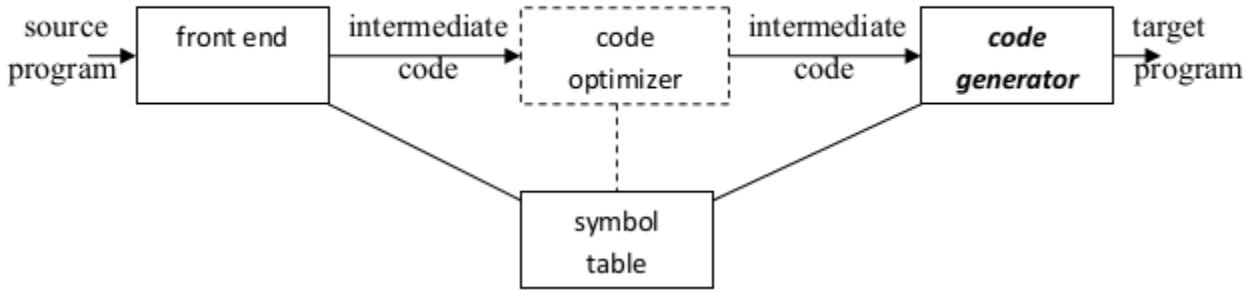
- A simple but effective technique for improving the target code is peephole optimization,a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.
- The peephole is a small, moving window on the target program. The code in the peepholeneed not contiguous, although some implementations do require this.it is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

- Characteristic of peephole optimizations:
  - Redundant-instructions elimination
  - Flow-of-control optimizations
  - Algebraic simplifications
  - Use of machine idioms
  - Unreachable Code.

16.What are the issues in the design of a code generator. (10)

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

Position of code generator



ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase :
1.Input to code generator
2.Target program
3.Memory management
4.Instruction selection
5.Register allocation
6.Evaluation order

. Input to code generator:

- The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.
- Intermediate representation can be :

a. Linear representation such as postfix notation
b. Three address representation such as quadruples
c. Virtual machine representation such as stack machine code
d. Graphical representations such as syntax trees and dags.

- Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. Target program:

- The output of the code generator is the target program. The output may be :

a. Absolute machine language
- It can be placed in a fixed memory location and can be executed immediately.
b. Relocatable machine language
- It allows subprograms to be compiled separately.
c. Assembly language
- Code generation is made easier.

3. Memory management:

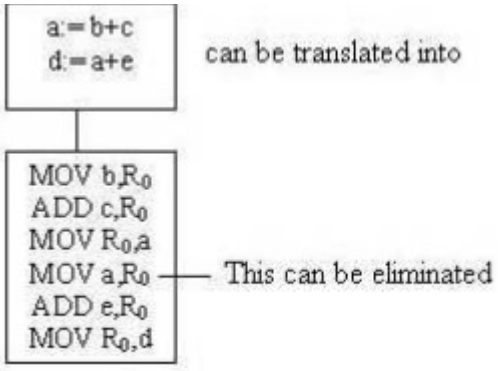
- Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.
- It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.
- Labels in three-address statements have to be converted to addresses of instructions.

For example,
j : goto i generates jump instruction as follows :

- if i < j, a backward jump instruction with target address equal to location of code for quadruple i is generated.
- if i > j, the jump is forward. We must store on a list for quadruple i the location of the first machine instruction generated for quadruple j. When i is processed, the machine locations for all instructions that forward jumps to i are filled.

4. Instruction selection:

- The instructions of target machine should be complete and uniform.
- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
- The quality of the generated code is determined by its speed and size.
- The former statement can be translated into the latter statement as shown below:



5. Register allocation

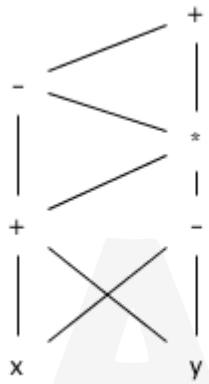
- Instructions involving register operands are shorter and faster than those involving operands in memory.
-

The use of registers is subdivided into two subproblems :

- Register allocation – the set of variables that will reside in registers at a point in the program is selected.

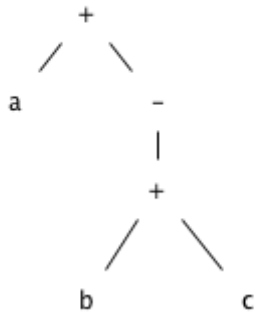
Register assignment – the specific register that a variable will reside in is picked.
Certain machine requires even-odd register pairs for some operands and results.
For example , consider the division instruction of the form :
D x, y where, x – dividend even register in even/odd register pair y – divisor even register holds the remainder odd register holds the quotient
6. Evaluation order
- The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

17.a) Construct the DAG for the expression. (4)
((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))



b) Translate the arithmetic expression a+-(b+c) into (6)
a)Syntax tree b)Quadruples c)Triples d)Indirect triples

a)Syntax tree



b)Quadruples.

	op	arg1	arg2	result
0	+	b	c	t1
1	minus	t1		t2
2	+	a	t2	t3

c) Triples

	op	arg1	arg2
0	+	b	c
1	minus	(0)	
2	+	a	(1)

d)Indirect triples

	op	arg1	arg2
0	+	b	c
1	minus	(0)	
2	+	a	(1)

	instruction
0	(0)
1	(1)
2	(2)

18.Write a note on the translation of boolean expression. (10)

Boolean Expressions

compute logical values
change the flow of control
boolean operators are:
and or not

E → E or E
| E and E
| not E
| (E)

| id relop id
| true
| false

Methods of translation

Evaluate similar to arithmetic expressions

Normally use 1 for true and 0 for false

implement by flow of control

given expression E1or E2

if E1evaluates to true

then E1or E2evaluates to true

without evaluating E2

Numerical representation

a or b and not c

t1= not c

t2= b and t1

t3= a or t2

relational expression a < b is equivalent to

if a < b then 1 else 0

1. if a < b goto 4.
2. t = 0
3. goto 5
4. t = 1

Syntax directed translation of boolean expressions

$E \rightarrow E_1$ or E_2

E.place := newtmp

emit(E.place ':=' E_1.place 'or' E_2.place)

$E \rightarrow E_1$ and E_2

E.place:= newtmp

emit(E.place ':=' E_1.place 'and' E_2.place)

$E \rightarrow$ not E_1

E.place := newtmp

emit(E.place ':=' 'not' E_1.place)

$E \rightarrow (E_1)$ $E.place = E_1.place$

E → id1 relop id2

E.place := newtmp

emit(if id1.place relop id2.place goto nextstat+3)

emit(E.place = 0)

emit(goto nextstat+2)

emit(E.place = 1)

E → true

E.place := newtmp

emit(E.place = '1')

E → false

E.place := newtmp

emit(E.place = '0')

Control flow translation of boolean expression

$E \rightarrow E_1$ or E_2 E_1.true := E.true

E_1.false := newlabel

E_2.true := E.true

E_2.false := E.false

E.code := E_1.code || gen(E_1.false) || E_2.code

$E \rightarrow E_1$ and E_2 E_1.true := new label

E_1 false := E.false

E_2.true := E.true

E_2 false := E.false

E.code := E_1.code || gen(E_1.true) || E_2.code

Control flow translation of boolean expression ...

$E \rightarrow \text{not } E_1$ $E_1.\text{true} := E.\text{false}$

$E_1.\text{false} := E.\text{true}$

$E.\text{code} := E_1.\text{code}$

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

$E_1.\text{false} := E.\text{false}$

$E.\text{code} := E_1.\text{code}$

19.Explain optimization of basic blocks. (10)

OPTIMIZATION OF BASIC BLOCKS

There are two types of basic block optimizations. They are :

- Structure-Preserving Transformations
- Algebraic Transformations

Structure-Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

Common sub-expression elimination
Dead code elimination
Renaming of temporary variables
Interchange of two independent adjacent statements.

Common sub-expression elimination:

Common sub expressions need not be computed over and over again. Instead

they can be computed once and kept in store from where it's referenced when encountered again of course providing the variable values in the expression still remain constant.

Example:
a: =b+c
b: =a-d
c: =b+c
d: =a-d
The 2 nd and 4 th statements compute the same expression: b+c and a-d
Basic block can be transformed to
a: = b+c
b: = a-d
c: = a
d: = b

Dead code elimination:

It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

Renaming of temporary variables:

A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is another temporary name, and change all uses of t to u.
In this we can transform a basic block to its equivalent block called normal-form block.

Interchange of two independent adjacent statements:

Two statements
t 1 :=b+c
t 2 :=x+y
can be interchanged or reordered in its computation in the basic block when value of t 1
does not affect the value of t 2 .

Algebraic Transformations:

Algebraic identities represent another important class of optimizations on basic blocks.This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength. Another class of related optimizations is constant folding. Here we evaluate constantexpressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28.
The relational operators <=, >=, <, >, + and = sometimes generate unexpected common sub expressions.
Associative laws may also be applied to expose common sub expressions. For example, if
the source code has the assignments

a :=b+c
e :=c+d+b
the following intermediate code may be generated:
a :=b+c
t :=c+d
e :=t+b
ಠ

Example:
x:=x+0 can be removed
x:=y**2 can be replaced by a cheaper statement x:=y*y

The compiler writer should examine the language carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate x*y-x*z as x*(y-z) but it may not evaluate a+(b-c) as (a+b)-c.

20.Discuss the different storage allocation strategies. (10)

Static allocation

Compiler makes the decision regarding storage allocation by looking only at the program text

Dynamic allocation

Storage allocation decisions are made only while the program is running

Stack allocation

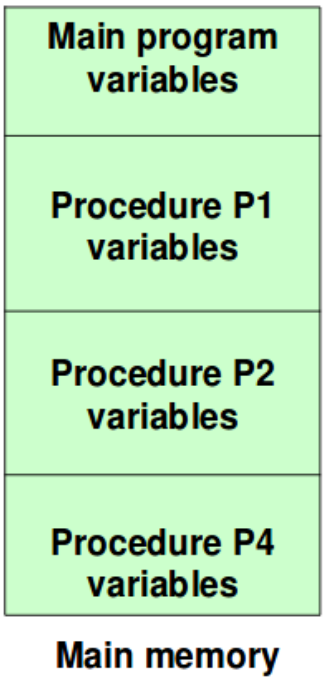
Names local to a procedure are allocated space on a stack

Heap allocation

Used for data that may live even after a procedure call returns
Ex: dynamic data structures such as symbol tables
Requires memory manager with garbage collection

Static Data Storage Allocation

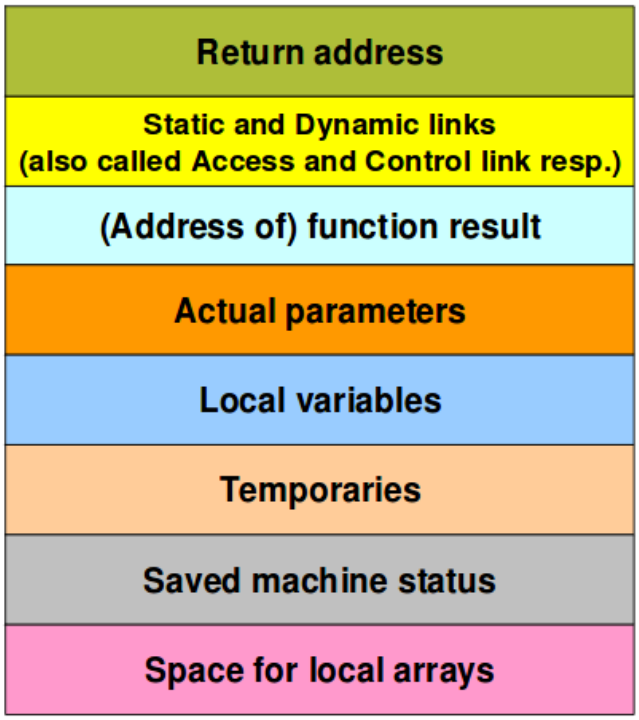
- Compiler allocates space for all variables (local and global) of all procedures at compile time
  - No stack/heap allocation; no overheads
  - Ex: Fortran IV and Fortran 77
  - Variable access is fast since addresses are known at compile time
  - No recursion



Dynamic Data Storage Allocation

- Compiler allocates space only for golbal variables at compile time
- Space for variables of procedures will be allocated at run-time
  - Stack/heap allocation
  - Ex: C, C++, Java, Fortran 8/9
  - Variable access is slow (compared to static allocation) since addresses are accessed through the stack/heap pointer
  - Recursion can be implemened

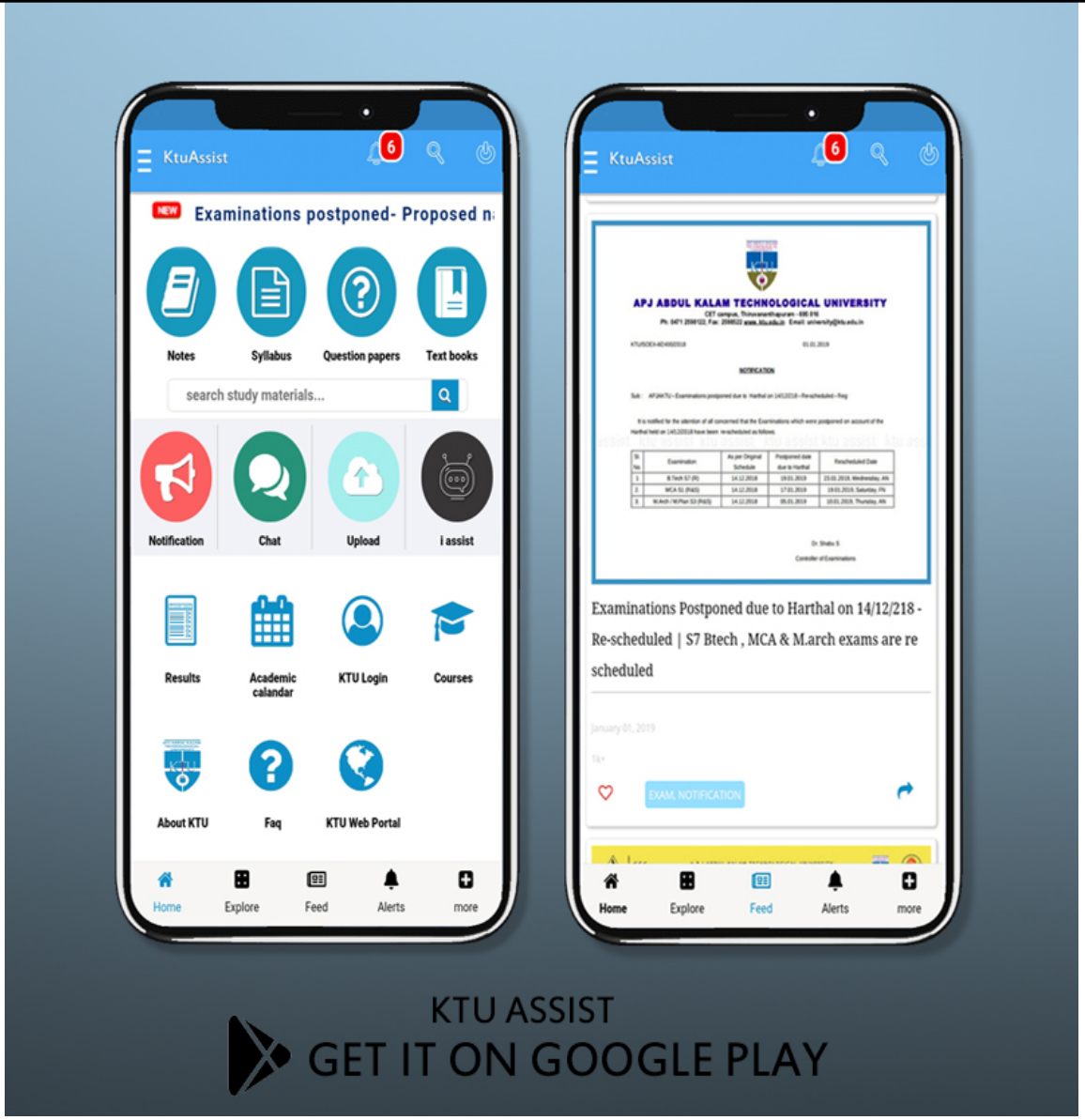
Activation Record Structure



Note:

The position of the fields of the act. record as shown are only notional.

Implementations can choose different orders; e.g., function result could be at the top of the act. record.



END

facebook.com/ktuassist

instagram.com/ktu_assist