Total Pages:2

Reg. No._____ Name:_____

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

SIXTH SEMESTER B.TECH MODEL EXAMINATION, APRIL 2018
Course Code: CS304
Course Name: COMPILER DESIGN

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

PART A
Answer all questions, each carries 3 marks.

1. Briefly explain the role of a lexical analyzer.
2. What are regular expressions? Give examples.
3. Explain any two error recovery strategies in a parser.
4. What is an ambiguous grammar? Give an example.

PART B
Answer any two full questions, each carries 9 marks

5. Explain the different phases of a compiler. (9)
6. a) What is the role of transition diagrams in the construction of a lexical analyzer?(4.5)
   b) Show that the following grammar is not LL(1) . (4.5)

   S →iEtSS' | a
   S' →eS | ε

   E→ b

7. a) Explain the working of Nonrecursive predictive parser. (5)

   b) Find FIRST and FOLLOW for the given grammar (4)

   D→ type list ;

   list → id tlist ;

   tlist → , id tlist | ε

   type → int | float

PART C
Answer all questions, each carries 3 marks.

8. What do you mean by configuration of an LR Parser?
9. What are the different types of conflicts that may occur in a shift reduce parser?

10. What is an L-attributed definition?
11. Draw a DAG for the expression a+a*(b-c)+(b-c)*d

PART D

Answer any two full questions, each carries 9 marks.

12. Explain operator precedence parsing with example . **(9)**

13. Construct the SLR parsing table for the given grammar **(9)**

S → AA

A → aA | b

14. a) What is the difference between synthesized attributes and inherited attributes? (5)

b) Explain bottom up evaluation of S- attributed definition. (4)

PART E

Answer any four full questions, each carries 10 marks.

15. Explain the different intermediate representations. . **(10)**
16. Explain the different storage allocation strategies. **(10)**
17. Explain the different methods of translating Boolean expressions. **(10)**
18. Explain the code generation algorithm . **(10)**
19. Explain the principal sources of optimization . **(10)**
20. Explain the issues in the design of a code generator **(10)**

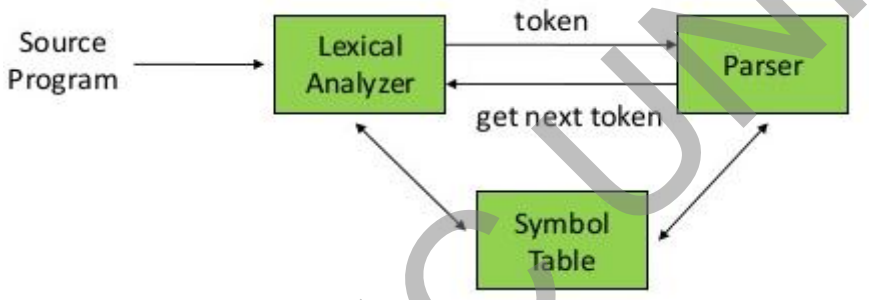
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

SIXTH SEMESTER B.TECH MODEL EXAMINATION, APRIL 2018
Course Code: CS304
Course Name: COMPILER DESIGN

ANSWER KEY

1)

- Task is to read the input characters and produce a sequence of tokens
- Gives output to parser for syntax analysis



- Performs other secondary tasks also
    - Removal of blank spaces
    - Correlating error messages
- Can be divided into two phases
    - Scanning- performing simple jobs
    - Lexical analysis- performing complex jobs (Any 3 points- 3 marks)

2) A **Regular Expression** can be recursively defined as follows −

- **ε** is a Regular Expression indicates the language containing an empty string. **(L (ε) = {ε})**
- **φ** is a Regular Expression denoting an empty language. **(L (φ) = { })**
- **x** is a Regular Expression where **L = {x}**
- If **X** is a Regular Expression denoting the language **L(X)** and **Y** is a Regular Expression denoting the language **L(Y)**, then
    - **X + Y** is a Regular Expression corresponding to the language **L(X) ∪ L(Y)** where **L(X+Y) = L(X) ∪ L(Y)**.
    - **X . Y** is a Regular Expression corresponding to the language **L(X) . L(Y)** where **L(X.Y) = L(X) . L(Y)**

- **R*** is a Regular Expression corresponding to the language **L(R*)**where **L(R*) = (L(R))***

Eg: (a+b)* : Set of strings of a's and b's of any length including the null string. So L = { ε, a, b, aa , ab , bb , ba, aaa…….}

3) Panic mode

When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement mode

When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error productions

Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction

The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

(Any 2 strategies- 3marks)

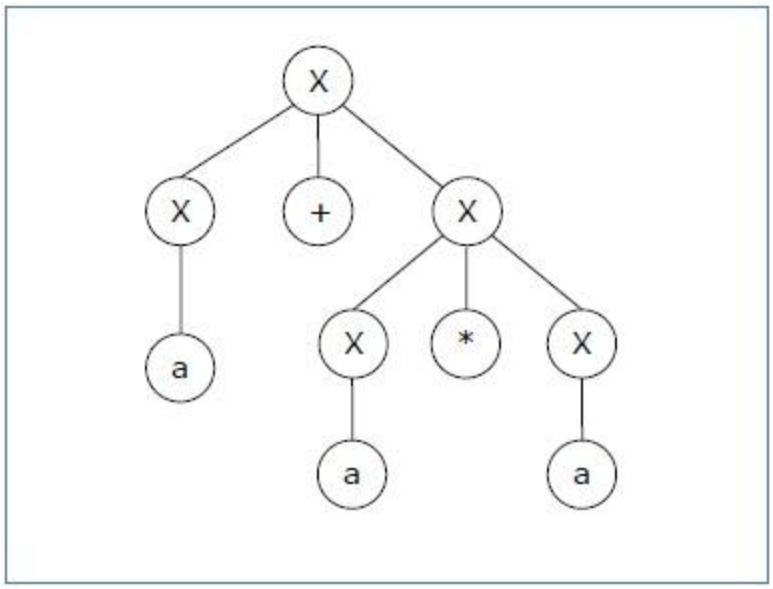
4) An **ambiguous grammar** is a context-free grammar for which there exists a string that can have more than one leftmost derivation or parse tree, while an **unambiguous grammar** is a context-free grammar for which every valid string has a unique leftmost derivation or parse tree.

Eg: X → X+X | X*X |X| a

Let's find out the derivation tree for the string "a+a*a". It has two leftmost derivations.

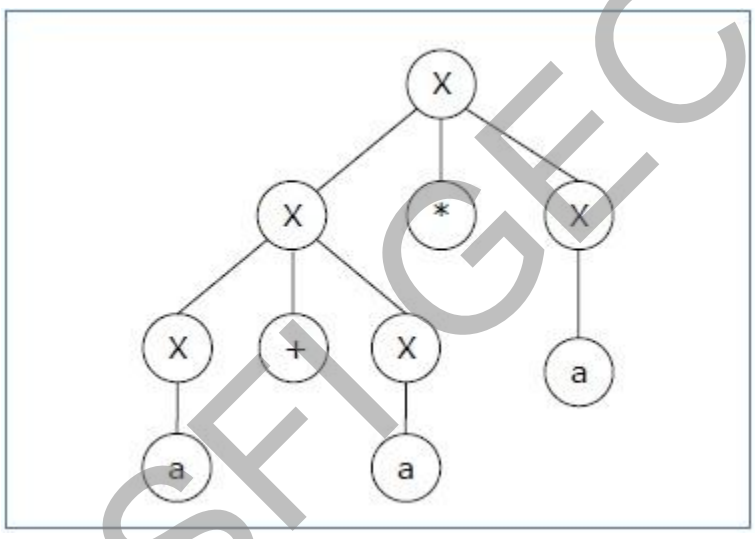
Derivation 1 − X → X+X → a +X → a+ X*X → a+a*X → a+a*a

Parse tree 1 −



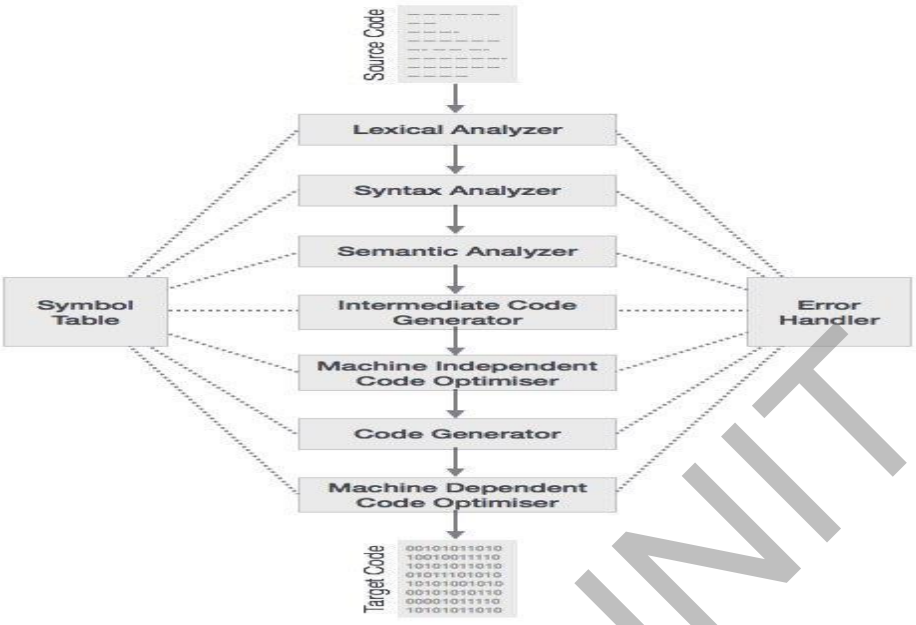
Derivation 2 − X → X*X → X+X*X → a+ X*X → a+a*X → a+a*a

Parse tree 2 −



Since there are two parse trees for a single string "a+a*a", the grammar **G** is ambiguous.

5) The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.



Lexical Analysis
The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:
<token-name, attribute-value>

Syntax Analysis
The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e., the parser checks if the expression made by the tokens is syntactically correct.

Semantic Analysis
Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not, etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generation
After semantic analysis, the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Code Optimization
The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Code Generation
In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of

(generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifiers' names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

6)

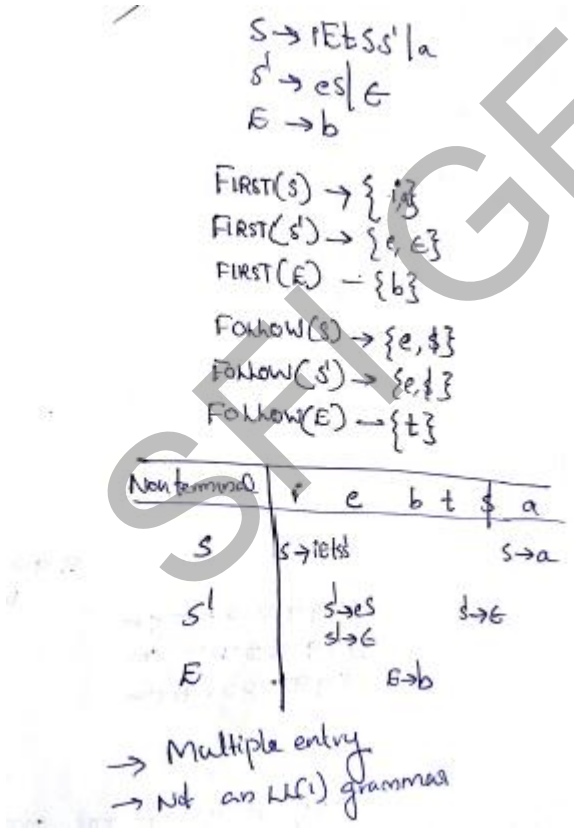
a) To specify a lexical analyzer we need a state machine, sometimes called Transition Diagram(TD), which is similar to a FSA.Transition Diagrams depict the actions that take place when the lexer is called by the parser to get the next token . FSA accepts or rejects a string. TD reads characters until finding a token,returns the read token and prepare the input buffer for the next call. In a TD, there is no out-transition from accepting states Transition labeled other (or not labeled) should be taken on any character except those labeling transitions out of a given state.

States can be marked with a $*$

: This indicates states on which a input retraction must take place.

To consider different kinds of lexeme, we usually build separate DFAs (or TD) corresponding to the regular expressions for each kind of lexeme then merge them into a single combined DFA (or TD).

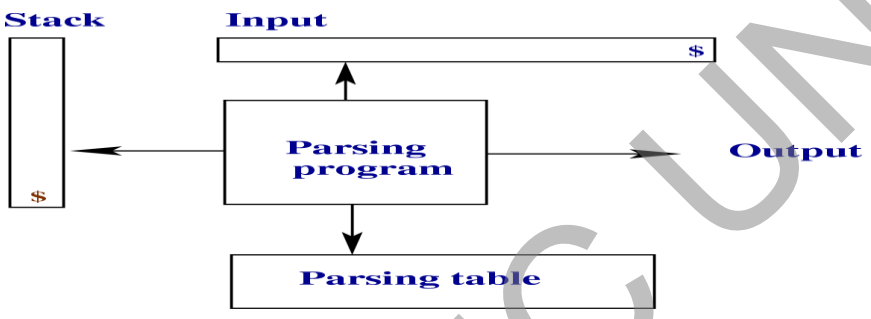
b)



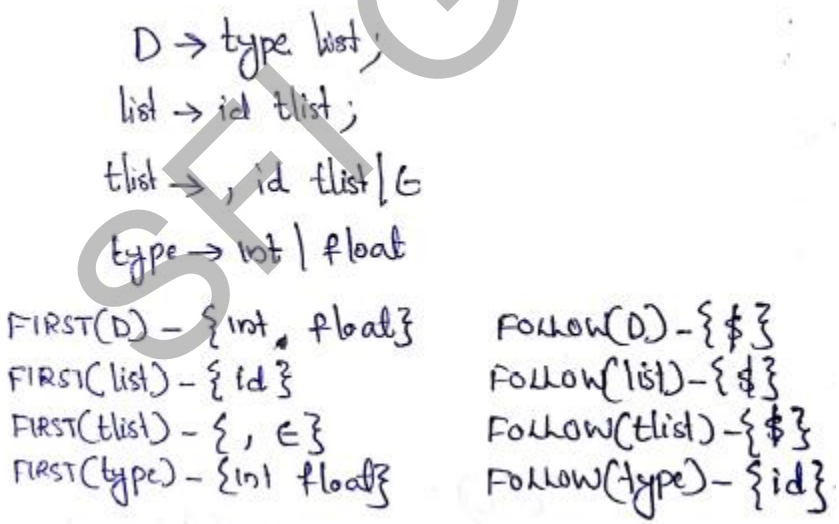
7**)**

a) Predictive parsing can be performed using a pushdown stack, avoiding recursive calls.

- Initially the stack holds just the start symbol of the grammar.
- At each step a symbol *X* is popped from the stack:
  - if *X* is a terminal symbol then it is matched with *lookahead* and *lookahead* is advanced,
  - if *X* is a nonterminal, then using *lookahead* and a *parsing table* (implementing the FIRST sets) a production is chosen and its right hand side is pushed onto the stack.
- This process goes on until the stack and the input string become empty. It is useful to have an *end_of_stack* and an *end_of_input* symbols. We denote them both by $.

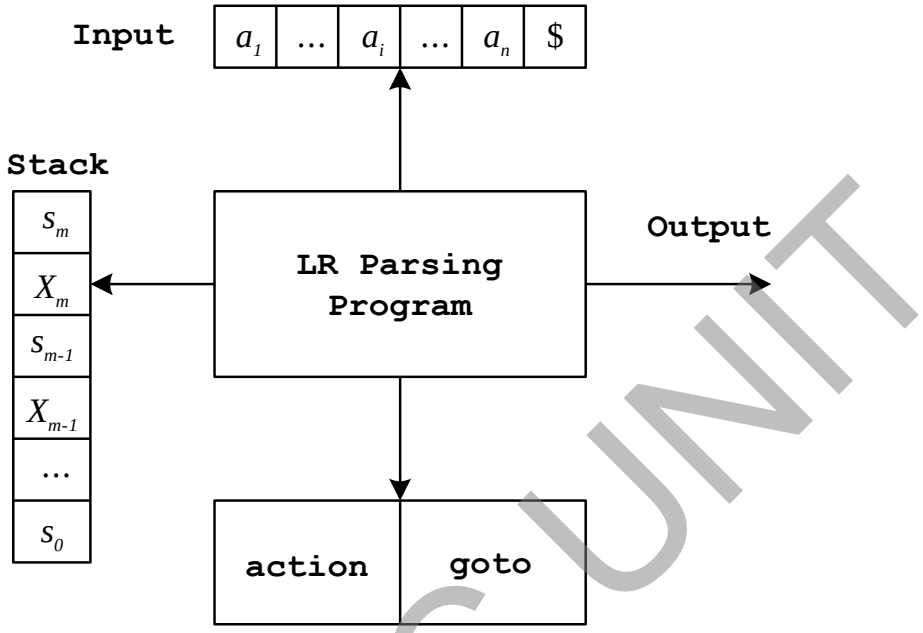


b)



8) An *LR Parser* consists of an input, output, a stack, a driver program and a parsing table that contains actions and gotos. While the driver program is the same for all LR parsers, each parser uses a different parsing table derived from the particular grammar. The *parsing program* reads characters from

the input buffer and, and uses a *stack* to keep strings of *symbols* and *system states*. Each state summarizes the information contained in the stack below it, so that the combination of the state on the stack top and the current input symbol and used to index the parsing table to determine the shift-reduce parsing ecision.



A *configuration* of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpected input:

(s_0 X_1 s_1 X_2 s_2 ... X_m s_m , a_i a_{i+1} ... a_n $)

The configuration represents the right-sentential form:

$$X_1 \; X_2 \; ... \; X_m \; a_i \; a_{i+1} \; ... \; a_n$$

The possible configurations resulting after each type of move are:

1) If *action*[s_m, a_i] = **shift** *s*, the parser executes a shift move, entering the configuration:

(s_0 X_1 s_1 X_2 s_2 ... X_m s_m a_i s , a_{i+1} ... a_n $)

2) If *action*[s_m, a_i] = **reduce** $A \rightarrow \beta$, then the prser executes a reduce move, entering the configuration:

$$s_0 \; X_1 \; s_1 \; X_2 \; s_2 \; ... \; X_{m-r} \; s_{m-r} \; A \; s \; , \; a_i \; a_{i+1} \; ... \; a_n \; \$)$$

where $s = goto[\, s_{m-r}, A \,]$ and r is the length of β.

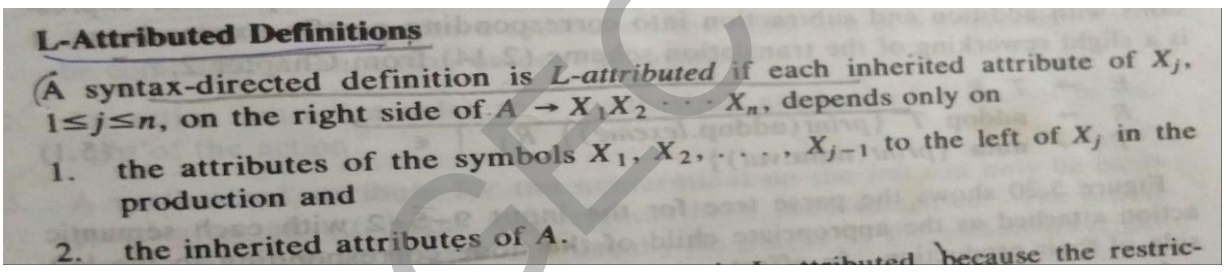
3) If *action*[s_m, a_i] = **accept** , parsing is completed

4) If *action*[s_m, a_i] = **error** , the parser has discovered an error and calls an error recovery routine.

9) When a grammar is not carefully thought out, the parser generated from the grammar may face two kinds of dilemmas:

1. Shift/Reduce Conflict:
   Enough terms have been read and a grammar rule can be recognized according to the accumulated terms. In this situation, the parser can make a reduction. If, however, there is also another grammar rule which calls for more terms to be accumulated and the look ahead token is just what the second grammar rule expected. In this situation, the parser may also make a shift operation. This dilemma faced by the parser is called the Shift/Reduce Conflict.
2. Reduce/Reduce Conflict:
   Enough terms have been read and two grammar rules are recognized based on the accumulated terms. If the parser then decides to make a reduction, should it reduces the accumulated terms according to the first or second grammar rules? This type of difficulty faced by the parser is called the Reduce/Reduce Conflict.

10) L-attributed grammars are a special type of attribute grammars. They allow the attributes to be evaluated in one left-to-right traversal of the abstract syntax tree. As a result, attribute evaluation in L-attributed grammars can be incorporated conveniently in top-down parsing.



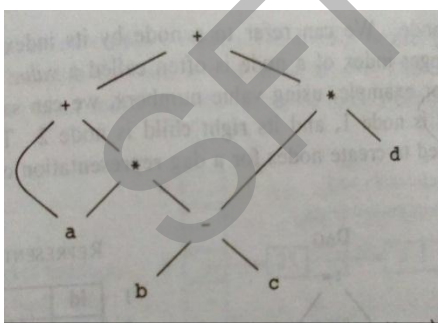
L-Attributed Definitions

A syntax-directed definition is *L-attributed* if each inherited attribute of X_j, $1 \le j \le n$, on the right side of $A \rightarrow X_1 X_2 \cdots X_n$, depends only on

1. the attributes of the symbols $X_1, X_2, \ldots, X_{j-1}$ to the left of X_j in the production and
2. the inherited attributes of A.

...because the restric-

11)



12) Done on small class of grammars

- Operator grammar
- No production on the right side have €
- no two adjacent non terminals

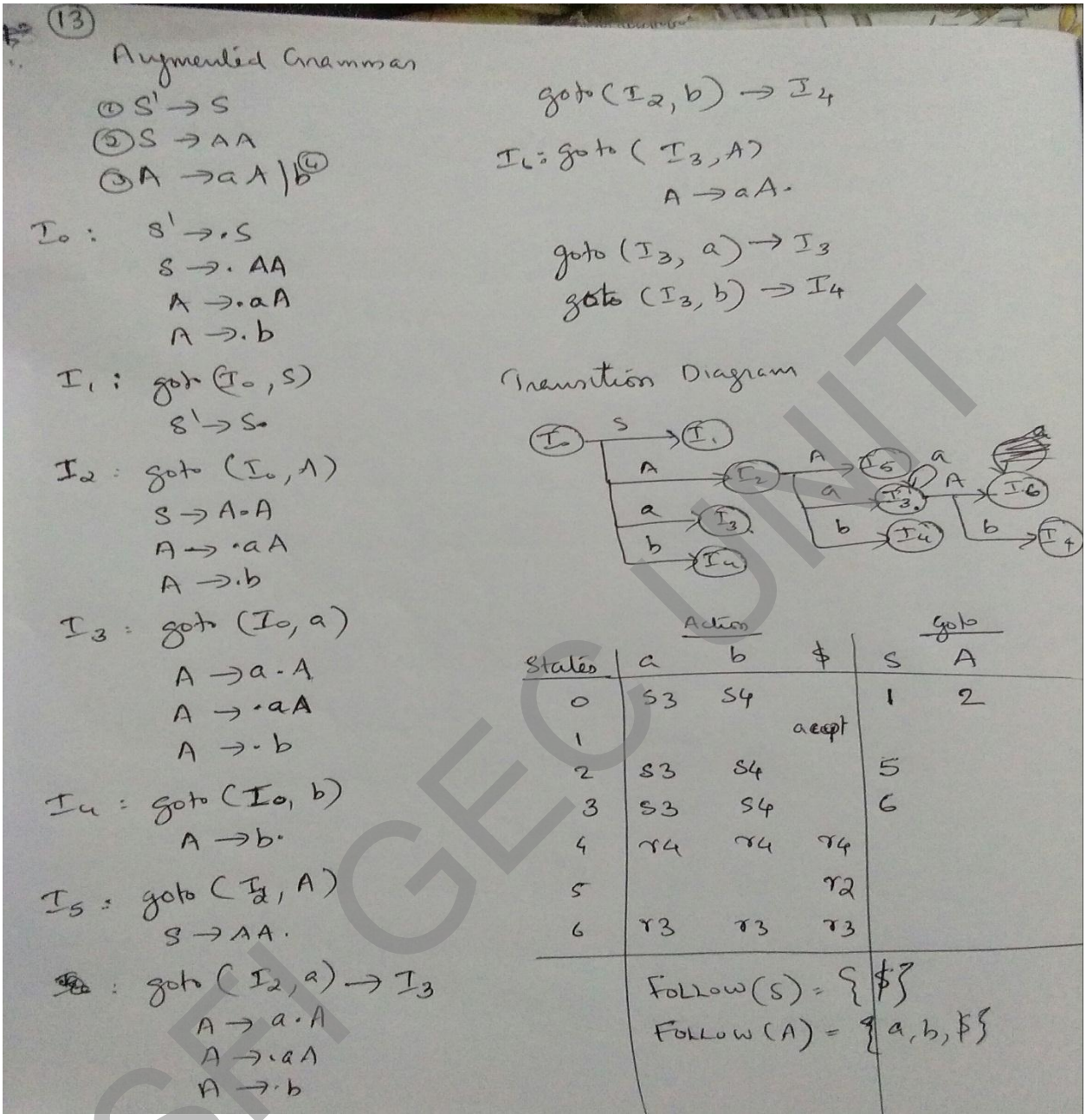
$$E \rightarrow EAE \mid (E) \mid -E \mid \mathbf{id}$$
$$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$$

-
- Not an operator grammar
- By substituting A we get
- $E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid \mathbf{id}$
-
- Three disjoint precedence relations are defined between the pair of terminals

$$<\cdot,\ \doteq,\ \text{and}\ \cdot>$$

RELATION	MEANING
$a <\cdot b$	a "yields precedence to" b
$a \doteq b$	a "has the same precedence as" b
$a \cdot> b$	a "takes precedence over" b

(definition 5 marks+ example 4 marks)



14)a) Synthesized attributes

- Synthesized attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:
- S → ABC
- If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.
- Synthesized attributes never take values from their parent nodes or any sibling nodes.

Inherited attributes

- In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,
- S → ABC
- A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

b)

- Synthesized Attributes can be evaluated by a bottom-up parser as the input is being analyzed avoiding the construction of a dependency graph.
- The parser keeps the values of the synthesized attributes in its stack.
- Whenever a reduction $A \rightarrow \alpha$ is made, the attribute for A is computed from the attributes of α which appear on the stack.
- Thus, a translator for an S-Attributed Definition can be simply implemented by extending the stack of an LR-Parser.
- Extra fields are added to the stack to hold the values of synthesized attributes.
- In the simple case of just one attribute per grammar symbol the stack has two fields: *state* and *val*.

state	*val*
Z	$Z.x$
Y	$Y.x$
X	$X.x$
...	...

- The current top of the stack is indicated by the pointer variable *top*.
- Synthesized attributes are computed just before each reduction:
- – Before the reduction $A \rightarrow XY\ Z$ is made, the attribute for A is computed
- $A.a := f(val[top]; val[top - 1]; val[top - 2])$.

- 15) After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program.
- This IR should have two important properties:
  - It should be easy to produce
  - it should be easy to translate into target program.
- IR is an intermediate stage of the mapping from source level abstractions to target machine abstractions.
- Kinds of Intermediate representations
  - Syntax trees
  - Postfix notation
  - Three address code
- Three-address code is a sequence of statements of the general form x := y *op* z
- Where x, y and z are names, constants, or compiler generated temporaries.

- op stands for any operator, such as fixed- or floating-point arithmetic operator, or a logical operator on Boolean-valued data.
- Thus a source language expression like x + y * z might be translated into a sequence
- t1 := y * z
- t2 := x + t1
- where t1 and t2 are compiler-generated temporary names.
- Here is a list of the common three-address instruction forms:
  - Assignment instructions of the form x = y op z, where op is a binary arithmetic or logical operation, and x, y, and z are addresses.
  - Assignments of the form x = op y, where op is a unary operation. Essential unary operations include unary minus, logical negation , shift operators etc.
  - Copy instructions of the form x = y, where x is assigned the value of y.
  - An unconditional jump goto L. The three-address instruction with label L is the next to be executed.

1. Conditional jumps such as if x relop y goto L, which apply a relational operator «, ==, >=, etc. to x and y, and execute the instruction with label L next if x stands in relation relop to y. If not , the three-address instruction following if x relop y goto L is executed next , in sequence.
2. Procedure calls and returns are implemented using the following instructions: param x for parameters; call p, n and y = call p, n for procedure and function calls , respectively; and return y, where y, representing a returned value , is optional. Their typical use is as the sequence of three

- address instructions
- param Xl
- param X2
- param xn
- call p , n

generated as part of a call of the procedure p(Xl , X2 , . . . ,xn) .

Indexed copy instructions of the form x = y [i] and x [i] = y. The instruction x = y [i] sets x to the value in the location i memory units beyond location y. The instruction x [i] = y sets the contents of the location I units beyond x to the value of y.

Address and pointer assignments of the form x = &y , x = *y , and *x = y

16) The different storage allocation strategies are :

1. Static allocation - lays out storage for all data objects at compile time

2. Stack allocation - manages the run-time storage as a stack.

3. Heap allocation - allocates and deallocates storage as needed at run time from a data area known as heap. (explain each briefly)

17)

Methods of Translating Boolean Expressions:

There are two principal methods of representing the value of a boolean expression. They are :

* To encode true and false numerically and to evaluate a boolean expression analogously to an arithmetic expression. Often, 1 is used to denote true and 0 to denote false.
* To implement boolean expressions by flow of control, that is, representing the value of a boolean expression by a position reached in a program. This method is particularly convenient in implementing the boolean expressions in flow-of-control statements, such as the if-then and while-do statements.

Numerical Representation

Here, 1 denotes true and 0 denotes false. Expressions will be evaluated completely from left to right, in a manner similar to arithmetic expressions.

For example :

* The translation for a or b and not c is the three-address sequence
* t1 : = not c
  t2 : = b and t1
  t3 : = a or t2
* A relational expression such as a < b is equivalent to the conditional statement
* if a < b then 1 else 0

which can be translated into the three-address code sequence (aga statement numbers at 100) :

100 : if a < b goto 103 101 : t : = 0

102 : goto 104

103 : t : = 1

104 :

Short-Circuit Code:

We can also translate a boolean expression into three-address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called "short-circuit" or "jumping" code. It is possible to evaluate boolean expressions without generating code for the boolean operators and, or, and not if we represent the value of an expression by a position in the code sequence.

Translation of a < b or c < d and e < f

100 : if a < b goto

103 101 : t1 : = 0

102 : goto 104 103 : t1 : = 1

104 : if c < d goto

107 105 : t2 : = 0

106 : goto 108

107 : t2 : = 1

108 : if e < f goto 111 109 : t3 : = 0
110 : goto 112
111 : t3 : = 1
112 : t4 : = t2 and t3
113 : t5 : = t1 or t4

18)

- Consider an instruction of the form "x := y **op** z"
- Invoke **getreg** to determine the location L where the result of "y op z" will be placed
- Determine a current location y' of y from the address descriptor (register location preferred). If y' is not L, generate "MOV y', L"
- Generate "**op** z', L", where z' is a current location of z from the address descriptor.
- Update the address and register descriptors for x, y, z, and L "

- Consider an instruction of the form "x := y"
- If y is in a register, change the register and address descriptors
- If y is in memory,
  - if x has next use in the block, invoke **getreg** to find a register r, generate "MOV y, r", and make r the location of x
  - otherwise, generate "MOV y, x"
- Once all statements in the basic block are processed, we store those names that are *live on exit* and *not in their memory locations*

19) A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global. Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the

function it computes.

Function preserving transformations examples:

Common sub expression elimination

Copy propagation, (explain briefly)

Dead-code elimination

Constant folding

The other transformations come up primarily when global optimizations are performed.

Frequently, a program will include several calculations of the offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

20) 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

1. 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.
e. • 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.

f. **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:gotoigenerates 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:

a:=b+c

d:=a+e (a)

MOV b,R0

ADD c,R0

MOV R0,a (b)

MOV a,R0

ADD e,R0

MOV R0,d

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 :

1. Register allocation - the set of variables that will reside in registers at a point in the program is selected.
2. Register assignment - the specific register that a value picked•
3. 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.

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

SIXTH SEMESTER B.TECH DEGREE MODEL EXAMINATION, MARCH 2018

Department: **Computer Science and Engineering**

Subject: - CS304: COMPILER DESIGN

Time: 3 hours Max. Marks: 100

PART A
Answer *all* questions

1. **What are the tools used for compiler construction?** **(3)**
2. **What is a symbol table?** **(3)**
3. **Differentiate regular expression and regular definition with an example.** **(3)**
4. **Differentiate tokens, patterns, lexeme** **(3)**

Total: (12)

PART B
Answer any *two* full questions

5. **With a neat diagram, explain the different phases of a compiler. Mention the input and output of each phase.** **(9)**
6. **Explain Input buffering with example.** **(9)**
7. **Explain the construction of Predictive parsing table for the grammar.**

E->TE'

E'->+TE'|e

T->FT'

T'->*FT' | e

F->(E) | id

(9)

Total: (18)

PART C
Answer *all* questions

8. **What is an operator precedence parser?** **(3)**
9. **Define viable prefix, kernel & non-kernel items.** **(3)**
10. **Give the syntax-directed definition for if-else statement** **(3)**
11. **Explain synthesized and inherited attributes.** (3)

Total: (12)

PART D

Answer any *two* full questions

12. Construct LR(0) items of following grammar

S-> L=R
S -> R
L-> *R
L-> ID
R-> L

(9)

13. Construct CLR parsing table for the following grammar **(9)**

S -> CC

C -> cC | d

14. Explain bottom up evaluation of S attributed definitions. (9)

Total: (18)

PART E

Answer any four full questions

15. Explain Storage- allocation strategies. **(10)**

16. a. Explain implementations of three address statements. **(5)**

b. Draw syntax tree and DAG for statement a:=b*-c+b*-c **(5)**

17. Explain Principal sources of code optimization **(10)**

18. Explain the issues in the design of code generator. (10)

Total: (40)

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

SIXTH SEMESTER B.TECH DEGREE MODEL EXAMINATION, MARCH 2018

Department: **Computer Science and Engineering**

Subject: - CS304: COMPILER DESIGN

ANSWER KEY

Time: 3 hours Max. Marks: 100

PART A

Answer *all* questions

1. a)Parser Generators

Input: Grammatical description of a programming language
Output: Syntax analyzers.

Parser generator takes the grammatical description of a programming language and produces a syntax analyzer.

b)Scanner Generators

Input: Regular expression description of the tokens of a language
Output: Lexical analyzers.
Scanner generator generates lexical analyzers from a regular expression description of the tokens of a language.

c)Syntax-directed Translation Engines

Input: Parse tree.
Output: Intermediate code.
Syntax-directed translation engines produce collections of routines that walk a parse tree and generates intermediate code.

d)Automatic Code Generators

Input: Intermediate language.
Output: Machine language.
Code-generator takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for a target machine.

e)Data-flow Analysis Engines

Data-flow analysis engine gathers the information, that is, the values transmitted from one part of a program to each of the other parts. Data-flow analysis is a key part of code optimization. **(3)**

2. A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly. Whenever an identifier is detected by a lexical analyzer, it is entered into the symbol table. The attributes of an identifier cannot be determined by the lexical analyzer. **(3)**

3. **Regular expression: It is *representator* of regular language. Regular expression is mathematically represent by some expression called regular expression. Regular expression is character sequence that define a search pattern.**
   Eg: Regular expression for identifier is letter(letter/digit)*
   Regular definition: is the name given to regular expression
   eg:Regular definition is id---->letter(letter/digit)*

(3)

4. **Tokens- Sequence of characters that have a collective meaning.**

Patterns- There is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token

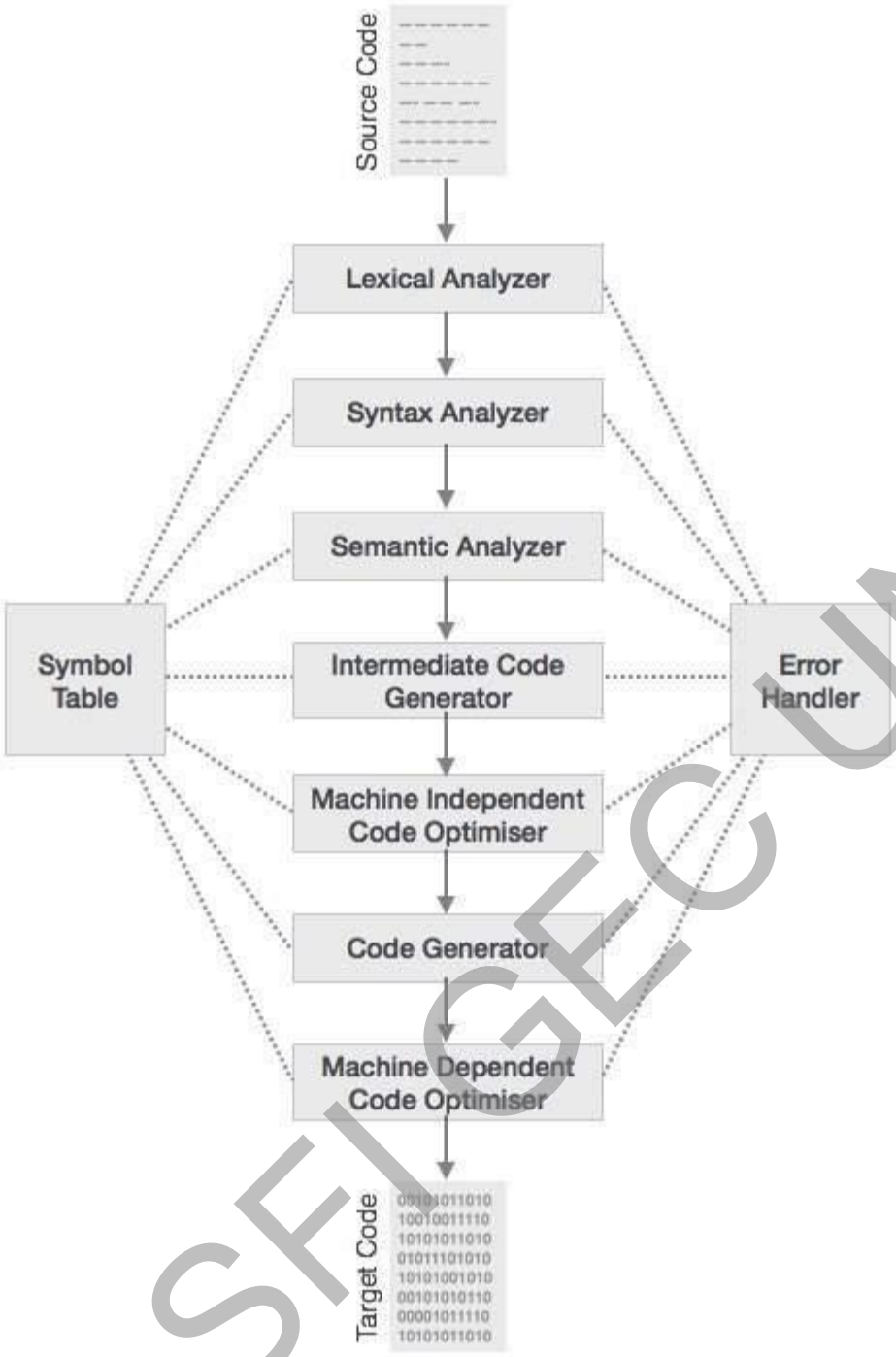
Lexeme- A sequence of characters in the source program that is matched by the pattern for a token.

(3)

Total: (12)

PART B
Answer any *two* full questions

5. **The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.**



Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

```
<token-name, attribute-value>
```

Syntax Analysis

The next phase is called the syntax analysis or parsing. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

(9)

6. **Explain Input buffering with example.**

To ensure that a right lexeme is found, one or more characters have to be looked up beyond the next lexeme.

• Hence a two-buffer scheme is introduced to handle large lookaheads safely.

• Techniques for speeding up the process of lexical analyzer such as the use of sentinels to mark the buffer end have been adopted.

There are three general approaches for the implementation of a lexical analyzer:

(i) By using a lexical-analyzer generator, such as lex compiler to produce the lexical analyzer from a regular expression based specification. In this, the generator provides routines for reading and buffering the input.

(ii) By writing the lexical analyzer in a conventional systems-programming language, using I/O facilities of that language to read the input.

(iii) By writing the lexical analyzer in assembly language and explicitly managing the reading of input.

Buffer Pairs

Because of large amount of time consumption in moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

Fig shows the buffer pairs which are used to hold the input data.

Scheme

• Consists of two buffers, each consists of N-character size which are reloaded alternatively.

• N-Number of characters on one disk block, e.g., 4096.

• N characters are read from the input file to the buffer using one system read command.

• *eof* is inserted at the end if the number of characters is less than N.

Pointers

Two pointers *lexemeBegin* and *forward* are maintained.

***lexeme Begin* points to the beginning of the current lexeme which is yet to be found.**

***forward* scans ahead until a match for a pattern is found.**

• Once a lexeme is found, *lexemebegin* is set to the character immediately after the lexeme which is just found and *forward* is set to the character at its right end.

• Current lexeme is the set of characters between two pointers.

Disadvantages of this scheme

• This scheme works well most of the time, but the amount of lookahead is limited.

• This limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer.

(eg.) DECLARE (ARGl, ARG2, . . . , ARGn) in PL/1 program;

• It cannot determine whether the DECLARE is a keyword or an array name until the character that follows the right parenthesis.

Sentinels

• In the previous scheme, each time when the forward pointer is moved, a check is done to ensure that one half of the buffer has not moved off. If it is done, then the other half must be reloaded.

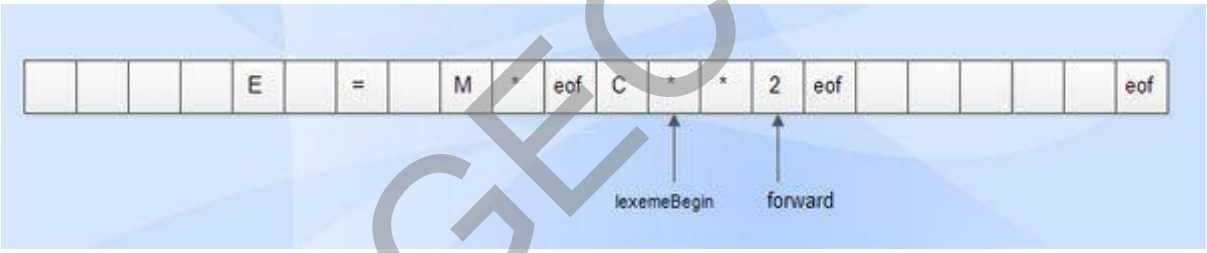
• Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.

Test 1: **For end of buffer.**

Test 2: **To determine what character is read.**

• The usage of sentinel reduces the two tests to one by extending each buffer half to hold a sentinel character at the end.

• The sentinel is a special character that cannot be part of the source program. *(eof* **character is used as sentinel).**



Advantages

• Most of the time, It performs only one test to see whether forward pointer points to an *eof*.

• Only when it reaches the end of the buffer half or *eof,* it performs more tests.

• Since N input characters are encountered between *eofs,* the average number of tests per input character is very close to 1.

(9)

7.

Predictive parser can be implemented by recursive-descent parsing (may need to manipulate the grammar, e.g eliminating left recursion and left factoring).

Requirement: by looking at the first terminal symbol that a nonterminal symbol can derive, we should be able to choose the right production to expand the nonterminal symbol.

If the requirement is met, the parser easily be implemented using a non-recursive scheme by building a parsing table.

(9)

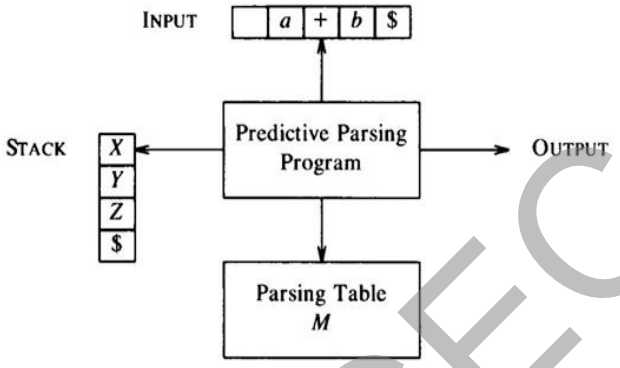


Fig. 4.13. Model of a nonrecursive predictive parser.

- **First(a) - the set of tokens that can appear as the first symbols of one or more strings generated from a. If a is empty string or can generate empty string, then empty string is also in First(a).**
- **Predictive parsing won't work on some type of grammars:**
  - **Left recursion: A->Aw**
  - **Have common left factor: A->aB | aC**

Predictive parsers can be constructed for a class of grammars called LL(1).

L->Left
L->Leftmost derivation
1->One input symbol at each step

No left recursive or ambiguous grammar can be LL(1)

<u>First(a):</u> Here, a is a string of symbols. The set of terminals that begin strings derived from a. If a is empty string or generates empty string, then empty string is in First(a).

<u>Follow(A):</u> Here, A is a nonterminal symbol. Follow(A) is the set of terminals that can immediately follow A in a sentential form

E->TE' **First(E)={(, id}, Follow(E)={), $}**

- **E'->+TE'|e First(E')={+, e}, Follow(E') = {), $}**
- **T->FT' First(T) = {(, id}, Follow(T) = {+,), $}**
- **T'->*FT' | e First(T') = {*, e}, Follow(T') = {+,), $}**
- **F->(E) | id First(F) = {(, id}, Follow(F) = {*, +,), $}**

Algorithm for construction of parsing table

INPUT :- Grammar G
OUTPUT:- Parsing table M
For each production A -> α , do the following :
For each terminal 'a' in FIRST(A), add A-> α to M[A,α].
If ε is in FIRST(α) then for each terminal b in FOLLOW(A) add A -> α to M[A,b]. If b is $ then also add A -> α to M[A,$].
If there is no production in M[A,a] , then set M[A,a] to error.

Non Terminal			INPUT SYMBOLS			
	id	+	*	(	)	$
E	E -> TE'			E -> TE'		
E'		E -> +TE'			E' -> ε	E' ->
T	T -> FT'			T -> FT'		
T'		T' -> ε	T' -> *FT'		T' -> ε	T' ->
F	F -> id			F -> (E)		

Step	Stack	Input	Next Action
1	$E	id*id$	E→TE'
2	$E'T	id*id$	T→FT'
3	$E'T'F	id*id$	F→id
4	$E'T'id	id*id$	match id

5	$E'T'	*id$	T'→*FT'
6	$T'F*	*id$	match *
7	$T'F	id$	F→id
8	$T'id	id$	match id
9	$T'	$	T'→ε
10	$	$	accept

(9)

8. A grammar is said to be operator precedence if it possess the following properties:

1. No production on the right side is ε.

2. There should not be any production rule possessing two adjacent non terminals at the right hand side.

(3)

9. The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes.

Kernel items, whish include the initial item, S'→ .S, and all items whose dots are not at the left end.

Non-kernel items, which have their dots at the left end. (3)

10.

1. S → if E then S1

E.true := new_label()

E.false :=S.next

S1.next :=S.next

S.code :=E.code | | gen_code(E.true ': ') | | S1.code

2. S → if E then S1 else S2

E.true := new_label()

E.false := new_label()

S1.next :=S.next

S2.next :=S.next

S.code :=E.code | | gen_code(E.true ‘: ‘) | | S1.code| | gen_code(‘go to’,S.next) | |gen_code(E.false ‘:’) | |

S2.code **(3)**

11.

Synthesized attributes

These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

S → ABC

If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

As in our previous example (E → E + T), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

Inherited attributes

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

S → ABC

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

(3)

PART D
Answer any *two* full questions

12. LR Parsers

+ Can be constructed to recognize virtually all programming languages constructed from context-free grammars
+ Most general non-backtracking shiftreduce parsing method
+ Very fast at detecting syntactic errors - Too much work to construct LR parsing by hand

- Item: production rule with information about current parsing state

$A \to XYZ$ yields the four items

$A \to \cdot XYZ$
$A \to X \cdot YZ$
$A \to XY \cdot Z$
$A \to XYZ \cdot$

I_0: $S' \to \cdot S$
$S \to \cdot L = R$
$S \to \cdot R$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$
$R \to \cdot L$

I_1: $S' \to S \cdot$

I_2: $S \to L \cdot = R$
$R \to L \cdot$

I_3: $S \to R \cdot$

I_4: $L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_5: $L \to \mathbf{id} \cdot$

I_6: $S \to L = \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

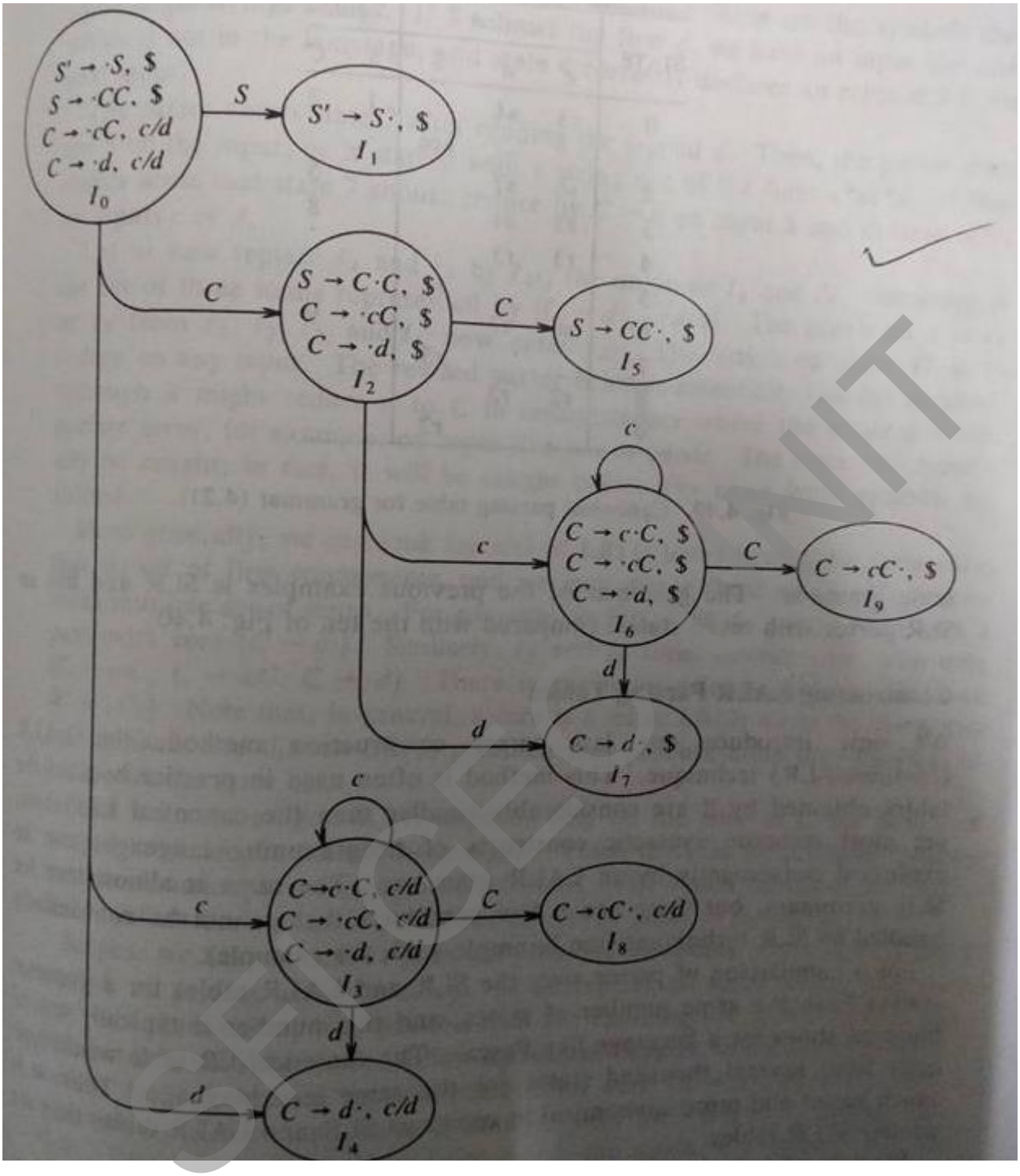
I_7: $L \to * R \cdot$

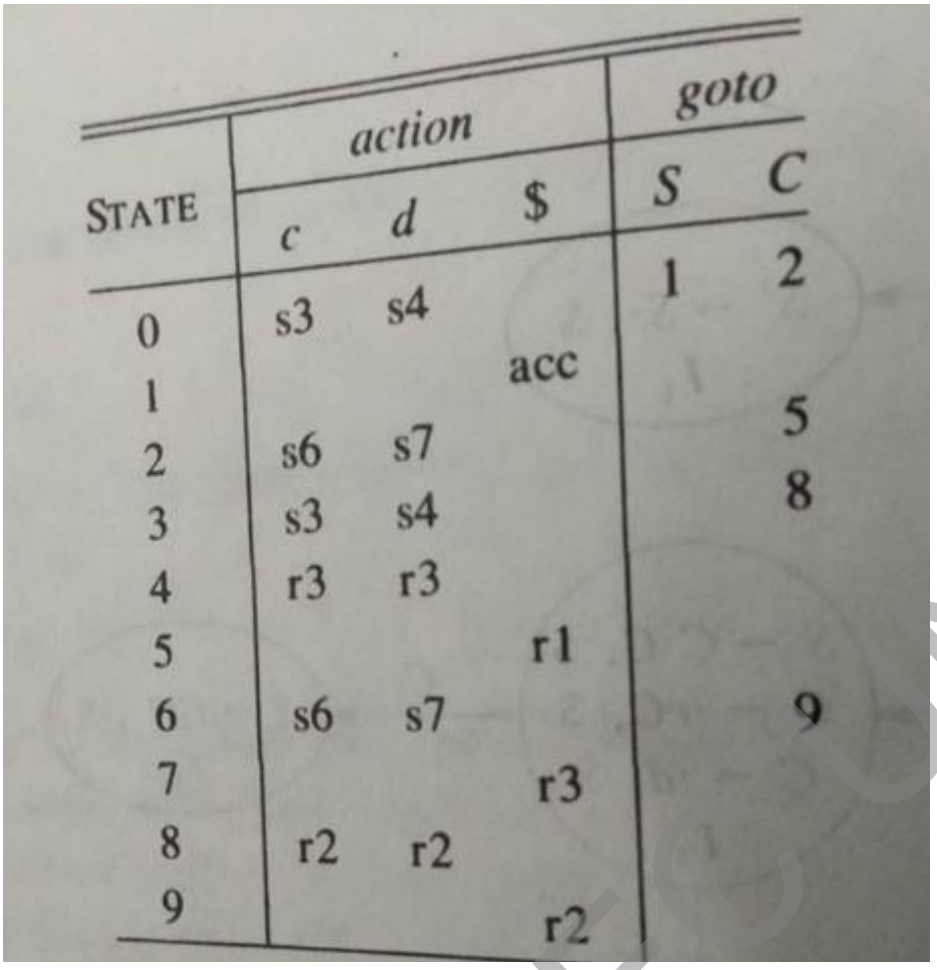
I_8: $R \to L \cdot$

I_9: $S \to L = R \cdot$

(9)

13.



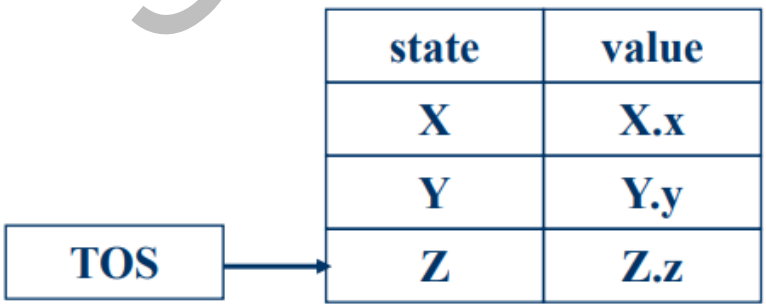


STATE	action			goto	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

(9)

14.

- Syntax-directed definitions with only synthesised attributes
- can be evaluated by a bottom up parser (BUP) as input is parsed
- values associated with the attributes can be kept on the stack as extra fields
- implementation using an LR parser (e.g. YACC)
- e.g. A => XYZ and A.a := f (X.x, Y.y, Z.z)



S-attributed Definitions: Parser Example

production	semantic rules
L => E \n	print(val[TOS])
E => E_1 + T	val[NTOS] := val[TOS-2] + val[TOS]
E => T	
T => T_1 * F	val[NTOS] := val[TOS-2] * val[TOS]
T => F	
F => (E)	val[NTOS] := val[TOS-1]
F => id	

- NTOS = TOS - r + 1 (r is # symbols reduced on RHS)

example of 3 * 5 + 4

S-attributed Definitions: Parser Example

input	state	value	production used
3 * 5 + 4 \n	-	-	
* 5 + 4 \n	3	3	
* 5 + 4 \n	F	3	F => digit
* 5 + 4 \n	T	3	T => F
5 + 4 \n	T *	3 :	
+ 4 \n	T * 5	3 : 5	
+ 4 \n	T * F	3 : 5	F => digit
+ 4 \n	T	15	T => T * F
+ 4 \n	E	15	E => T
4 \n	E +	15 :	
\n	E + 4	15 : 4	
\n	E + F	15 : 4	F => digit
\n	E + T	15 : 4	T => F
\n	E	19	E => E + T
	E \n	19 :	
	L	19	L => E \n

(9)

PART E
Answer any four full questions

15. **The different storage allocation strategies are;**

1. Static allocation lays out storage for all data objects at compile time.

2. Stack allocation manages the run-time storage as a stack.

3. Heap allocation: allocates and de-allocates storage as needed at runtime from a data known as the heap.

Static allocation

- **In static allocation, names bound to storage as the program is compiled, so there is no need for a run-time support package.time**
- **Since the bindings do not change at runtime, every time a procedure activated, its run-time, names bounded to the same storage location.**
- **Therefore values of local names retained across activations of a procedure. That is when control returns to a procedure the values of the locals are the same as they were when control left the last time.**

Stack allocation : Storage allocation strategies

- **All compilers for languages that use procedures, functions or methods as units of user functions define actions manage at least part of their runtime memory as a stack run-time stack.**
- **Each time a procedure called, space for its local variables is pushed onto a stack, and when the procedure terminates, space popped off t stack.the**

Calling Sequences: Storage allocation strategies

- **Procedures called implemented in what is called as calling sequence, which consists of code that allocates an activation record on the stack and enters information into its fields.**
- **A return sequence is similar to code to restore the state of a machine so the calling procedure can continue its execution after the call.**
- **The code is calling sequence of often divided between the calling procedure (caller) and a procedure is calls (callee)(callee).**
- **When designing calling sequences and the layout of activation record, the following principles are helpful:**
  1. Value communicated between caller and callee generally placed at the caller beginning of the callee's activation record, so they as close as possible to the
  caller's activation record.
  2. Fixed length items generally placed in the middle. Such items typically include the control link, the access link, and the machine status field.,
  3. Items whose size may not be known early enough placed at the end of the activation record.
  4. We must locate the top of the stack pointer judiciously. A common approach is to have it point to the end of fixed length fields in the activation is to have it point to fix the end of fixed length fields in the activation record. Fixed length data can then be accessed by fixed offsets, known to the intermediate code generator, relative to the top of the stack pointer.

Storage allocation strategies

- **The calling sequence and its division between caller and callee are as follows:**
  1. The caller evaluates the actual parameters.
  2. The caller stores a return address and the old value of top_sp into the callee's activation record. The caller then increments the top_sp to the respective positions.
  3. The callee-saves the register values and other status information.
  4. The callee initializes its local data and begins execution.

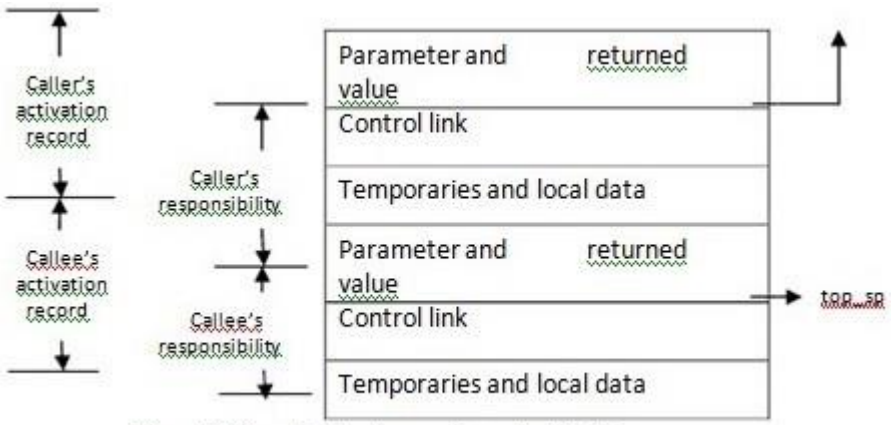


Fig. Division of task between caller and callee6.4

- **A suitable, corresponding return sequence is:**
  1. The callee places the return value next to the parameters.
  2. Using the information in the machine status field, the callee restores top_sp and other registers, and then branches to the return address that the caller placed in the status field.
  3. Although top_sp has been decremented, the caller knows where the return value is, relative to the current value of top_sp; the caller, therefore, may use that value.

Variable length data on stack: Storage allocation strategies

- **The runtime memory management system must deal frequently with the allocation of management objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack.**
- **The same scheme works for objects of any type if they are local to the procedure called local have a size that depends on the parameter of the call.**

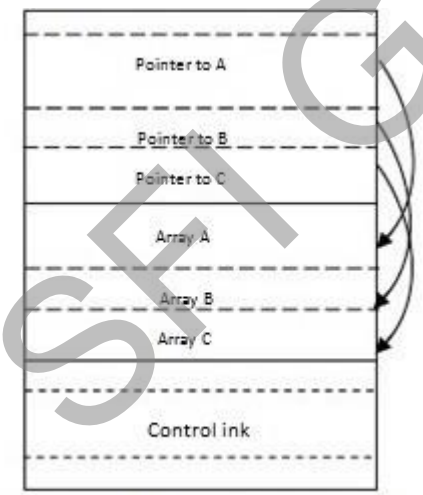


Fig Access to dynamically allocated arrays

Dangling Reference (Storage allocation strategies)

- **Whenever storage allocated, the problem of dangling reference arises. The dangling reference occurs when there is a reference to storage that has been allocated.**
- **It is a logical error to u dangling reference, since, the value of de use de-allocated storage is undefined according to the semantics of most languages.**
  - **Whenever storage allocated, the problem of dangling reference arises. The dangling reference occurs when there is a reference to storage that has been allocated.** (10)

16. a.

Three-Address Code

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code.

For example:

```
a = b + c * d;
```

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

```
r1 = c * d;
r2 = b + r1;
a = r2
```

r being used as registers in the target program.

A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms : quadruples, triples, Indirect Triples

Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format:

Op	arg_1	arg_2	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3

=	r3		a

Triples

Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

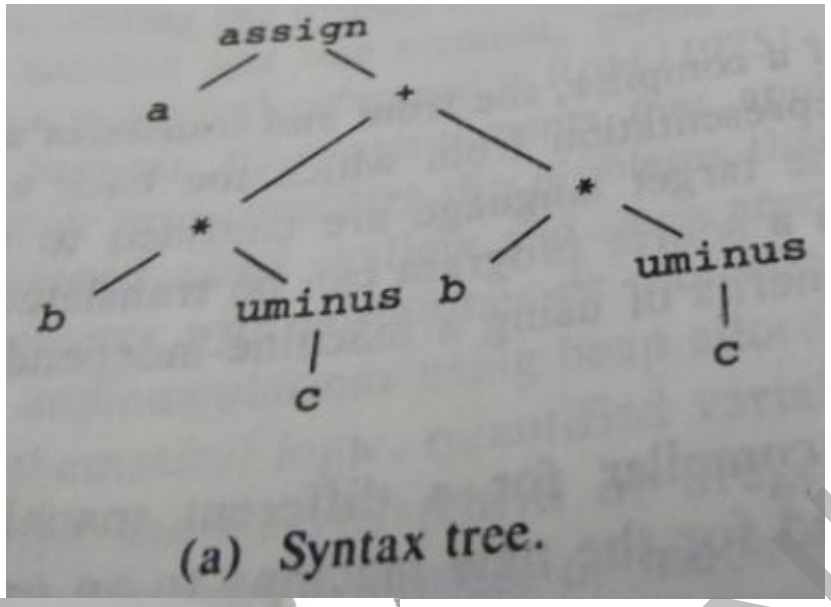
Op	arg_1	arg_2
*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

Triples face the problem of code immovability while optimization, as the results are positional and changing the order or position of an expression may cause problems.

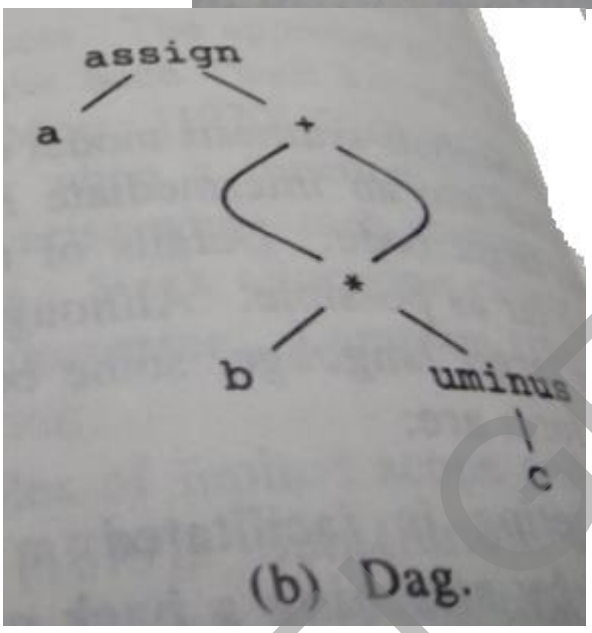
Indirect Triples

This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

b. (5)



(a) Syntax tree.



(b) Dag.

(5)

17. **A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global. Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.**

Function-Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes.

Function preserving transformations examples:

- **Common sub expression elimination**
- **Copy propagation,**
- **Dead-code elimination**
- **Constant folding**

The other transformations come up primarily when global optimizations are performed.

Frequently, a program will include several calculations of the offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

***

Common Sub expressions elimination:

- **An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.**

- **For example**

```
t1: = 4*i
t2: = a [t1]
t3: = 4*j
t4: = 4*i
t5: = n
t6: = b [t4] +t5
```

The above code can be optimized using the common sub-expression elimination as

```
t1: = 4*i
 t2: = a [t1]
t3: = 4*j
t5: = n
t6: = b [t1] +t5
```

The common sub expression t4: =4*i is eliminated as its computation is already in t1 and the value of i is not been changed from definition to use.

Copy Propagation:

Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

• For example:
x=Pi;

A=x*r*r;

The optimization using copy propagation can be done as follows: A=Pi*r*r;

Here the variable x is eliminated

Dead-Code Eliminations:

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.

Example:

```
i=0;
if(i=1)
{
a=b+5;
}
```

Here, 'if' statement is dead code because this condition will never get satisfied.

Constant folding:

Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding. One advantage of copy propagation is that it often turns the copy statement into dead code.

For example,

a=3.14157/2 can be replaced by
a=1.570 there by eliminating a division operation.

Loop Optimizations:

In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization:

- **Code motion, which moves code outside a loop;**
- **Induction-variable elimination, which we apply to replace variables from inner loop.**

- **Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.**

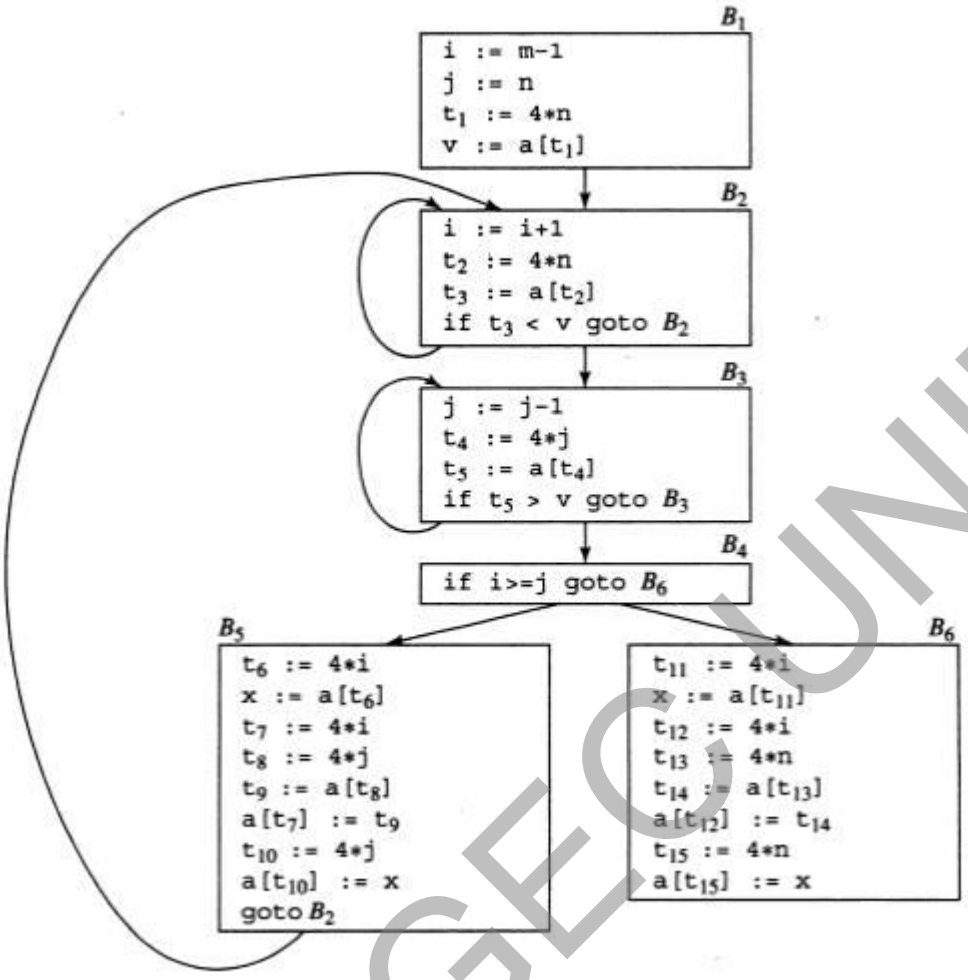


Fig. 5.2 Flow graph

Code Motion:

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion "before the loop" assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

```
while (i <= limit-2) /* statement does not change limit*/
```

Code motion will result in the equivalent of

```
t= limit-2;
while (i<=t) /* statement does not change limit or t */
```


Induction Variables :

Loops are usually processed inside out. For example consider the loop around B3. Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4. Such identifiers are called induction variables.

When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig.5.3 we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.

However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2- B5 is considered.

Example:

As the relationship t4:=4*j surely holds after such an assignment to t4 in Fig. and t4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t4:= 4*j-4 must hold. We may therefore replace the assignment t4:= 4*j by t4:= t4-4. The only problem is that t4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t4=4*j on entry to the block B3, we place an initializations of t4 at the end of the block where j itself is initialized, shown by the dashed addition to block B1 in Fig.5.3.

The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

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.

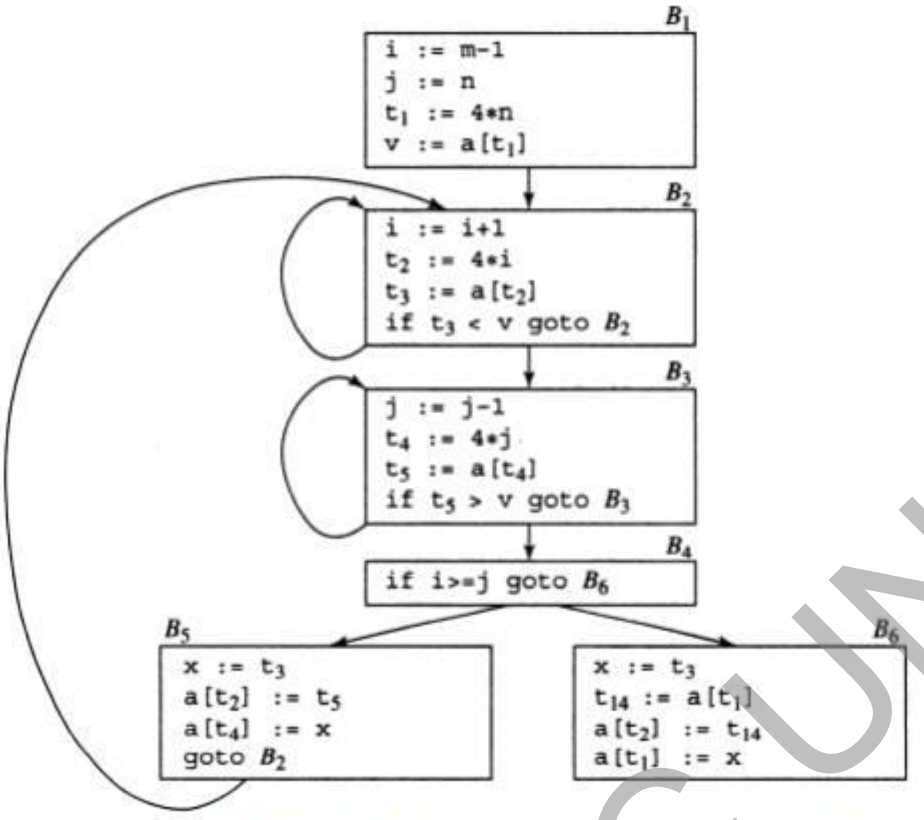


Fig. 5.3 B5 and B6 after common subexpression elimination

Fig. 5.3 B5 and B6 after common subexpression elimination

(10)

18. **Code generator phase generates the target code taking input as intermediate code.**

- **The output of intermediate code generator may be given directly to code generation or may pass through code optimization before generating code.**
- **Code generator main tasks:**
    1. **Instruction selection**
    2. **Register allocation and assignment**
    3. **Instruction ordering**

Issues in Design of Code generation:

- **Target code mainly depends on available instruction set and efficient usage of registers. The main issues in design of code generation are**

    1. Intermediate representation

- **Linear representation like postfix and three address code or quadruples and graphical representation like Syntax tree or DAG.**
- **Assume type checking is done and input in free of errors.**

2. Target Code

- **The code generator has to be aware of the nature of the target language for which the code is to be transformed.**
- **That language may facilitate some machine-specific instructions to help the compiler generate the code in a more convenient way.**
- **The target machine can have either CISC or RISC processor architecture.**
- **The target code may be absolute code, re-locatable machine code or assembly language code.**
- **Absolute code can be executed immediately as the addresses are fixed.**
- **But in case of re-locatable it requires linker and loader to place the code in appropriate location and map (link) the required library functions.**
- **If it generates assembly level code then assemblers are needed to convert it into machine level code before execution.**
- **Re-locatable code provides great deal of flexibilities as the functions can be compiled separately before generation of object code.**

1.1 Absolute machine language:

- **Producing an absolute machine language program as output has the advantage that it can be placed in a fixed location in memory and immediately executed.**

1.2 Relocatable machine language:

- **Producing a relocatable machine language program as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader.**
- **If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader, to link the separately compiled program segments.**

3. Assembly language:

- **Producing an assembly language program as output makes the process of code generation somewhat easier.**

Example

MOV R0, R1

ADD R1, R2

Target Machine supports for the following addressing modes

a. Absolute addressing mode

Example: MOV R0, M where M is the address of memory location of one of the operands. MOV R0, M moves the contents of register R0 to memory location M.

b. Register addressing mode where both the operands are in register.

Example: ADD R0, R1

c. Immediate addressing mode – The operand value appears in the instruction.

Example: ADD # 1, R0

d. Index addressing mode- this is of the form C(R) where the address of operand is at the location C+Contents(R)

Example: MOV 4(R0), M the operand is located at address = contents (4+contents (R0)) Cost of instruction is defined as cost of execution plus the number of memory access.

3. Address mapping

- **Address mapping defines the mapping between intermediate representations to address in the target code.**
- **These addresses are based on the runtime environment used like static, stack or heap.**
- **The identifiers are stored in symbol table during declaration of variables or functions, along with type.**
- **Each identifier can be accessed in symbol table based on width of each identifier and offset. The address of the specific instruction (in three address code) can be generated using back patching**

4. Instruction Set

- **The instruction set should be complete in the sense that all operations can be implemented.**
- **Sometimes a single operation may be implemented using many instruction (many set of instructions).**
- **The code generator should choose the most appropriate instruction.**
- **The instruction should be chosen in such a way that speed is of execution is minimum or other machine related resource utilization should be minimum.**
- **The code generator takes Intermediate Representation as input and converts (maps) it into target machine's instruction set.**
- **One representation can have many ways (instructions) to convert it, so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.**

Example

The factors to be considered during instruction selection are:

- **The uniformity and completeness of the instruction set.**
- **Instruction speed and machine idioms.**
- **Size of the instruction set.**

Eg., for the following address code is:

a := b + c

d := a + e

inefficient assembly code is:

```
MOV b, R0        R0 ← b
ADD  c, R0       R0 ← c + R0
MOV R0, a        a  ← R0
MOV a, R0        R0 ← a
ADD  e, R0       R0 ← e + R0
MOV R0 , d       d  ← R0
```

Here the fourth statement is redundant, and so is the third statement if

'a' is not subsequently used.

5. Register Allocation

- **A program has a number of values to be maintained during the execution.**
- **The target machine's architecture may not allow all of the values to be kept in the CPU memory or registers.**
- **Code generator decides what values to keep in the registers.**
- **Also, it decides the registers to be used to keep these values.**
- **Instructions involving register operands are usually shorter and faster than those involving operands in memory.**
- **Therefore efficient utilization of registers is particularly important in generating good code.**
- **During register allocation we select the set of variables that will reside in registers at each point in the program.**
- **During a subsequent register assignment phase, we pick the specific register that a variable will reside in.**

6. Memory Management

- **Mapping names in the source program to addresses of data objects in run-time memory is done cooperatively by the front end and the code generator.**
- **A name in a three- address statement refers to a symbol-table entry for the name.**
- **From the symbol-table information, a relative address can be determined for the name in a data area for the procedure.**

7. Evaluation of order

- **At last, the code generator decides the order in which the instruction will be executed.**
- **It creates schedules for instructions to execute them.**
- **The order in which computations are performed can affect the efficiency of the target code.**
- **Some computation orders require fewer registers to hold intermediate results than others.**

(10)

Total: (40)

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

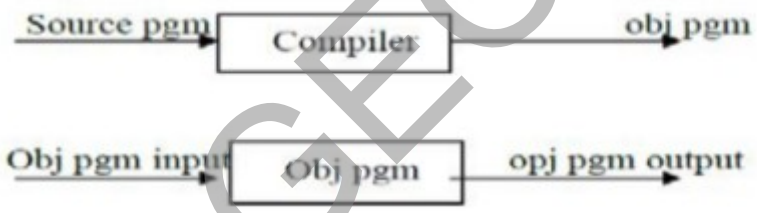
Max. Marks: 100 Duration: 3 hours

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.

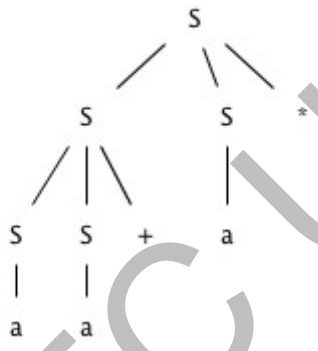
3.Consider the context-free grammar: S -> S S + | S S * | a and the string aa + a*. (3)

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.

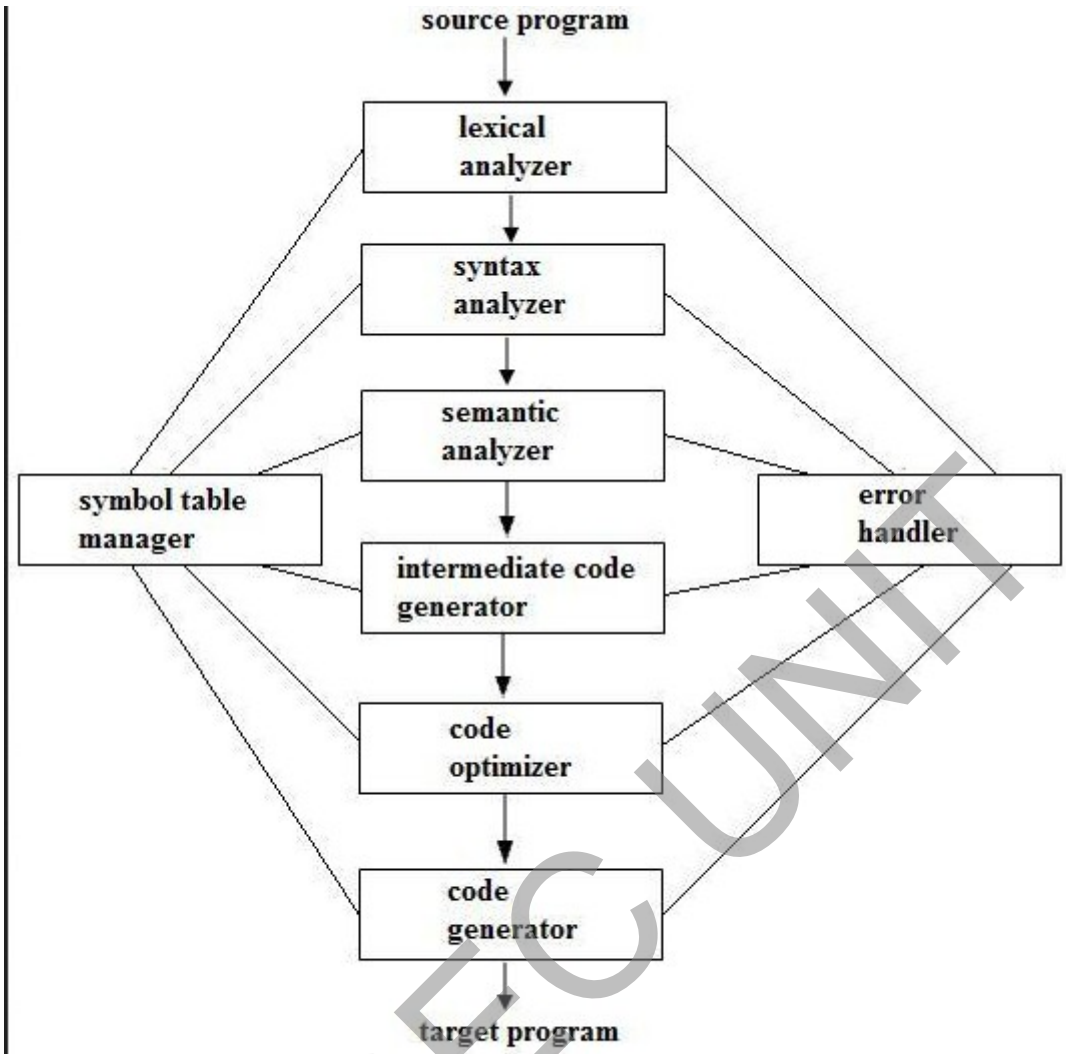
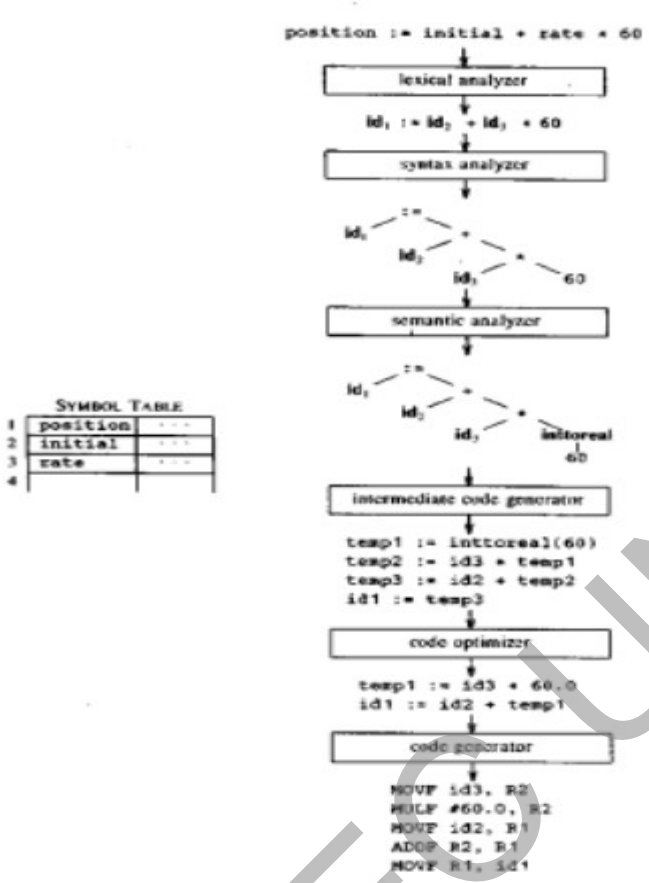


Fig 1.5 Phases of a compiler



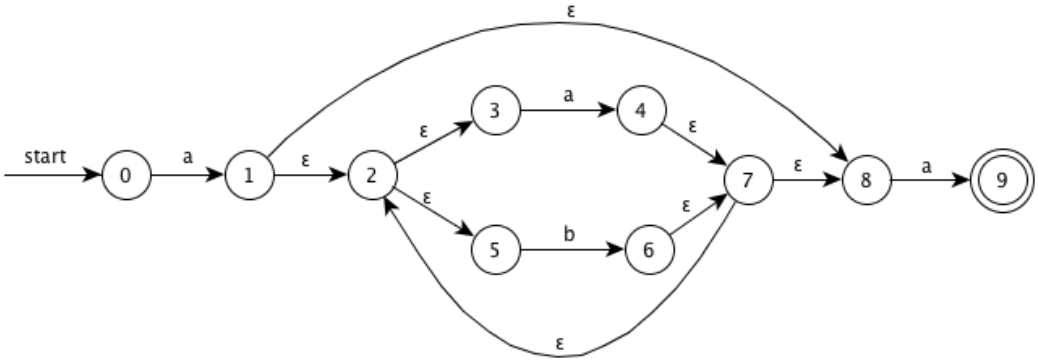
5 b)Calculate the number of tokens in the following C statement

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

No of Tokens-10

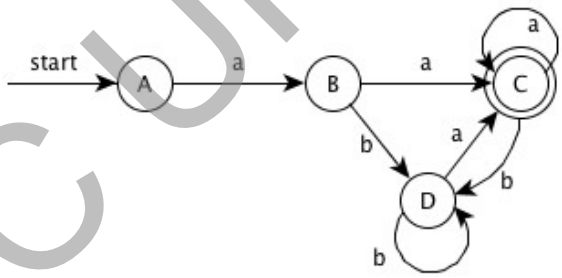
1.printf
2.(
3.“i=%d,&i=%x”
4.,
5.i
6.,
7.&
8.i
9.)
10.;

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.

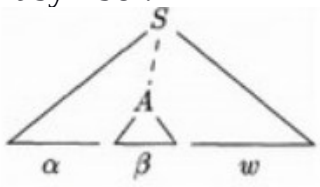


Figure 4.27: A handle $A \to \beta$ in the parse tree for $\alpha\beta w$

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

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

$D \rightarrow T\ L$
$\quad L.type := T.type$

$T \rightarrow int$
$\quad T.type := \text{INT}$

$T \rightarrow real$
$\quad T.type := \text{REAL}$

$L \rightarrow L_1\ ,\ id$
$\quad L_1.type := L.type;$
$\quad \text{addtype}(id.entry,\ L.type)$

$L \rightarrow id$
$\quad \text{addtype}(id.entry,\ L.type)$

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.

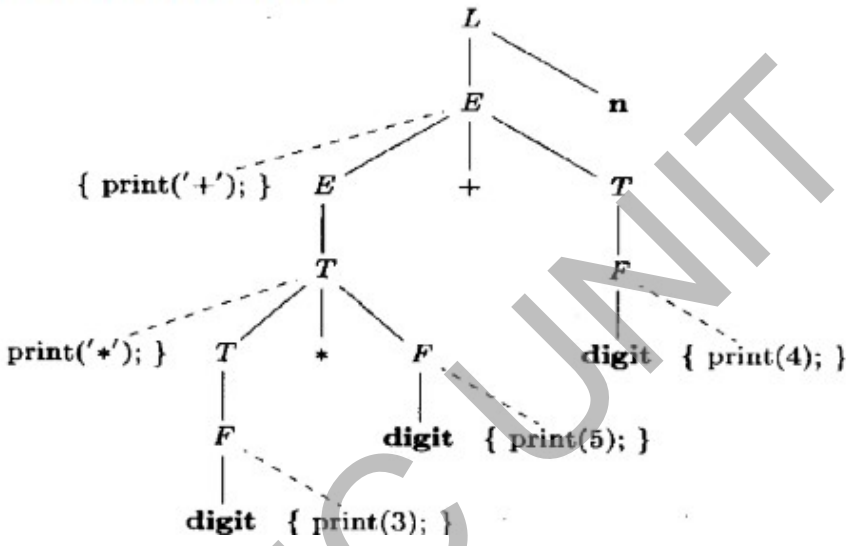
Stack	Input	Action
$	000111$	Shift
$0	00111$	Shift
$00	0111$	Shift
$000	111$	Shift
$0001	11$	Reduce S->01
$00S	11$	Shift
$00S1	1$	Reduce S->0S1
$0S	1$	Shift
$0S1	$	Reduce S->0S1
$S	$	Accept

Stack always contains viable prefixes.

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

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

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

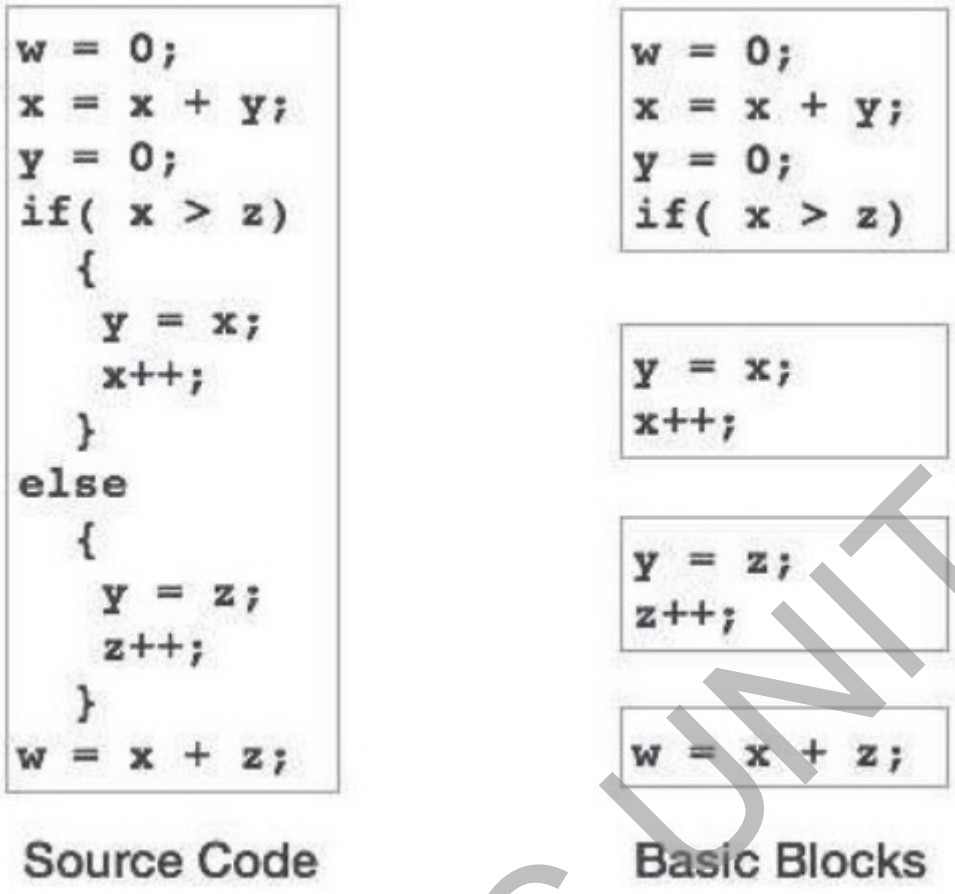
(Answer any four questions. Each carries 10 marks.)

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.

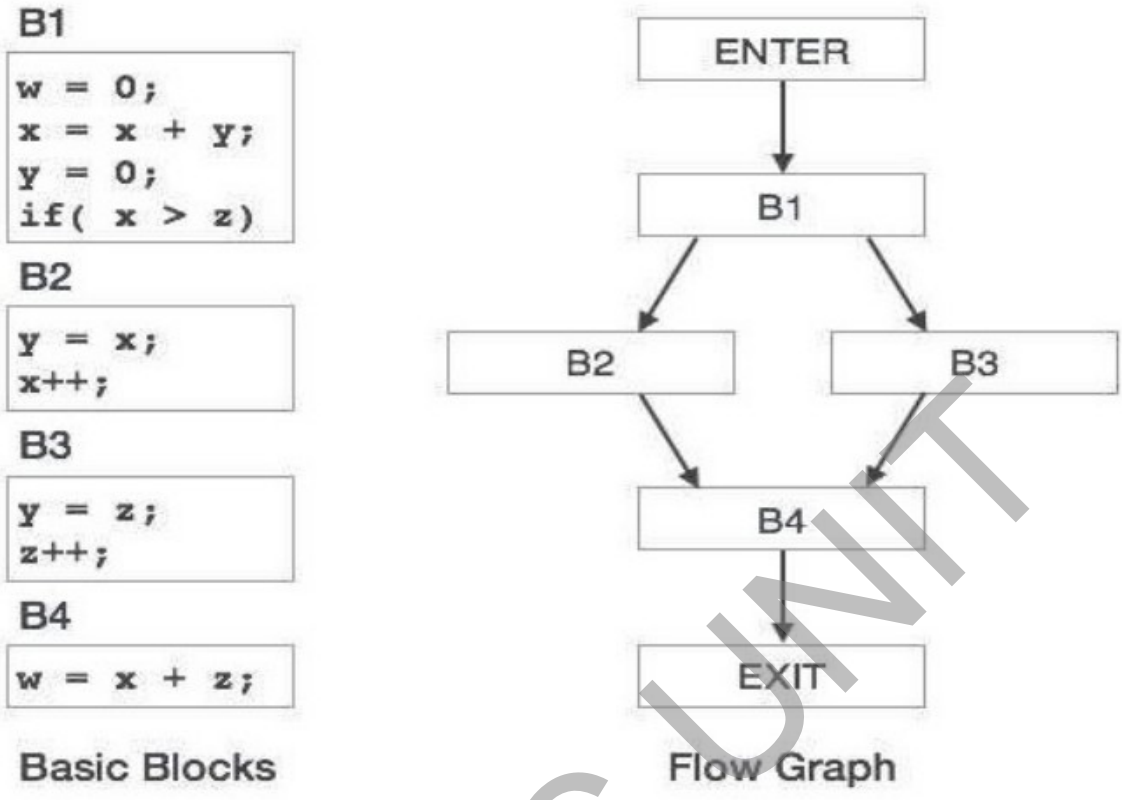
Basic blocks are important concepts from both code generation and optimization point of view.



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.

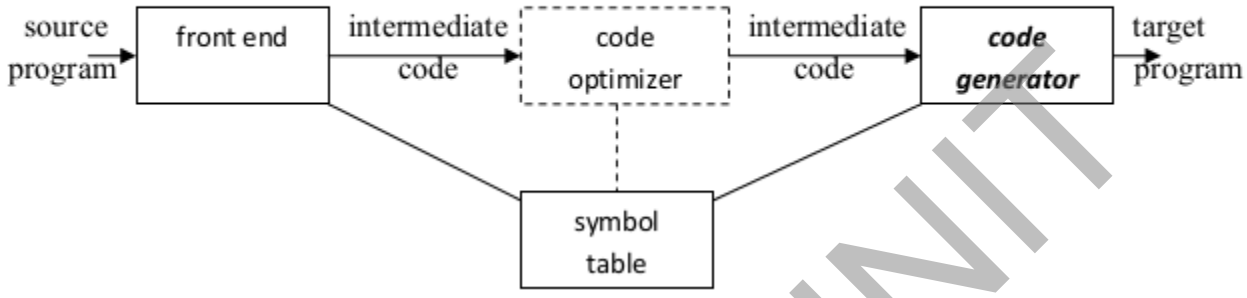
- A simple but effective technique for improving the target code is peephole optimization,a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, 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.

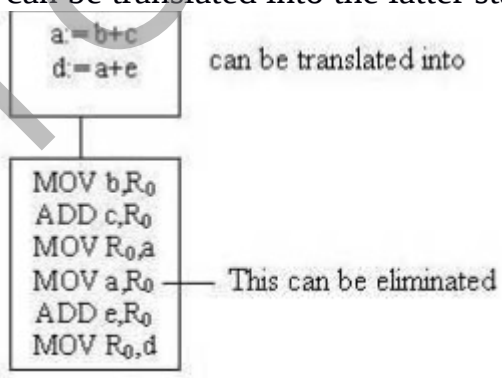
3. Memory management:

- 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:



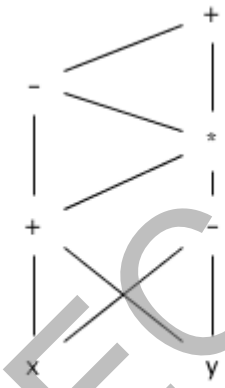
5. Register allocation

- 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.

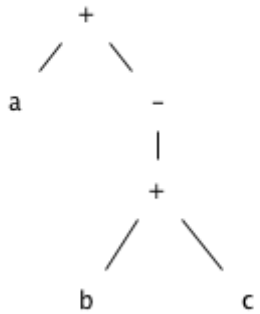
Register assignment – the specific register that a variable will reside in is picked. Certain machine requires even-odd register pairs for some operands and results. For example , consider the division instruction of the form :
D x, y where, x – dividend even register in even/odd register pair y – divisor 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

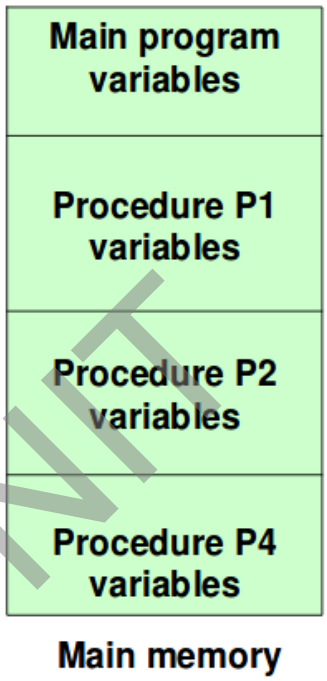
Names local to a procedure are allocated space on a stack

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

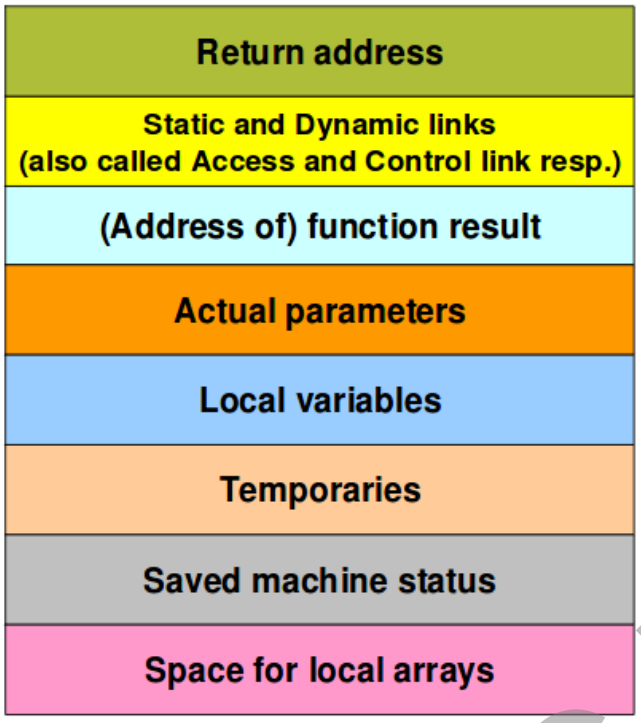
- Compiler allocates space for all variables (local and global) of all procedures at compile time
  - No stack/heap allocation; no overheads
  - Ex: Fortran IV and Fortran 77
  - Variable access is fast since addresses are known at compile time
  - No recursion



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.