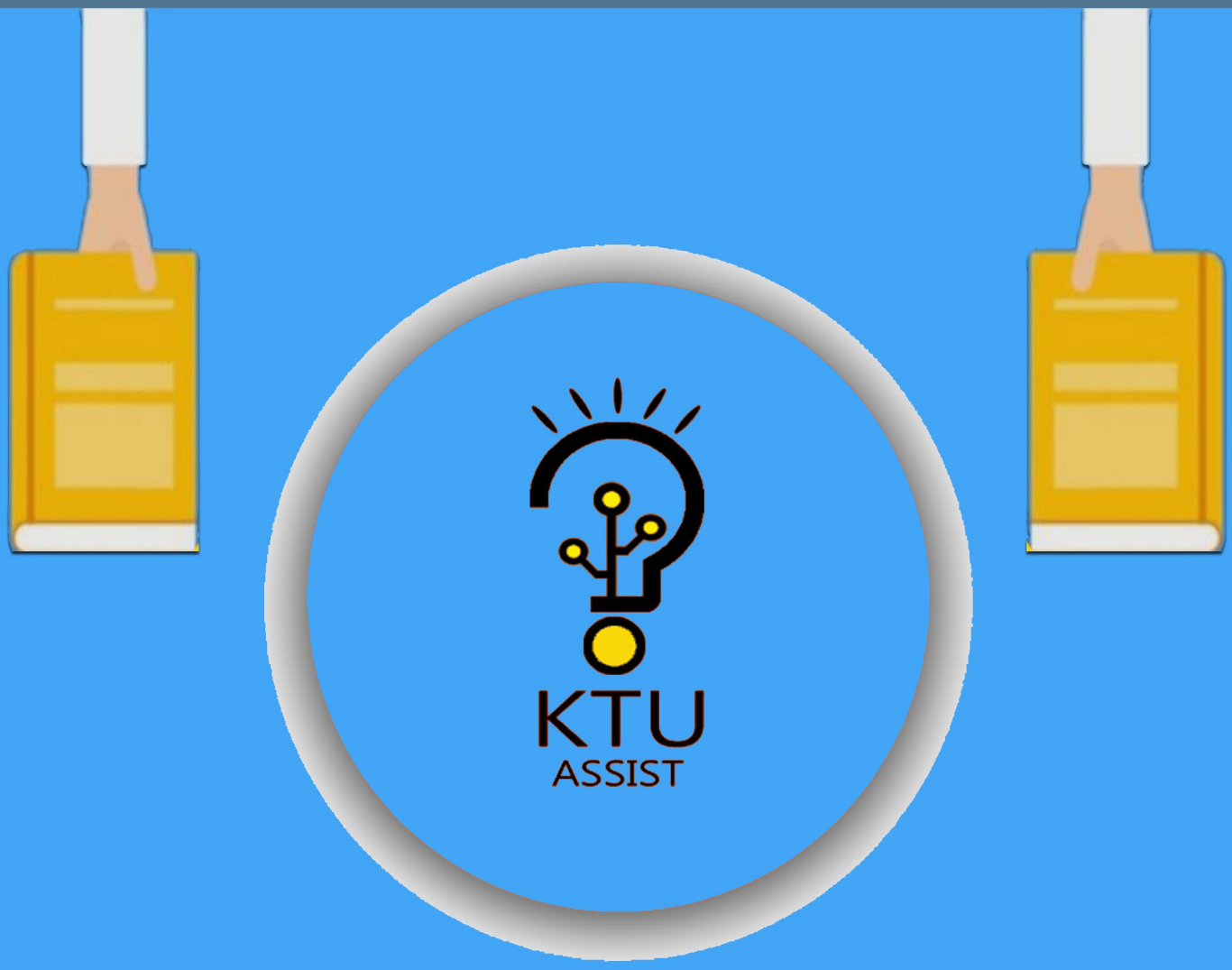
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

STUDY MATERIALS



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)**

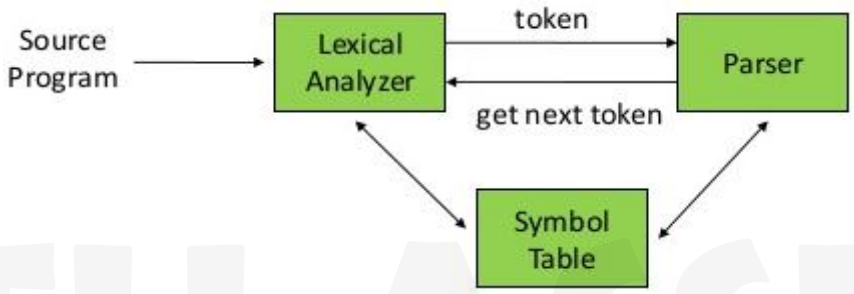
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

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)

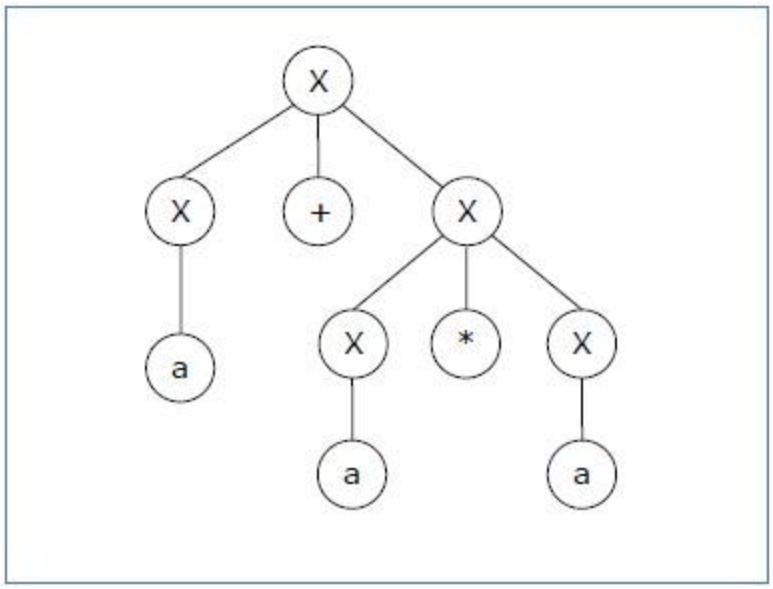
4) An **ambiguous grammar** is a context-free grammar for which there exists a string that can have more than one leftmost derivation or parse tree, while an **unambiguous grammar** is a context-free grammar for which every valid string has a unique leftmost derivation or parse tree.

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.

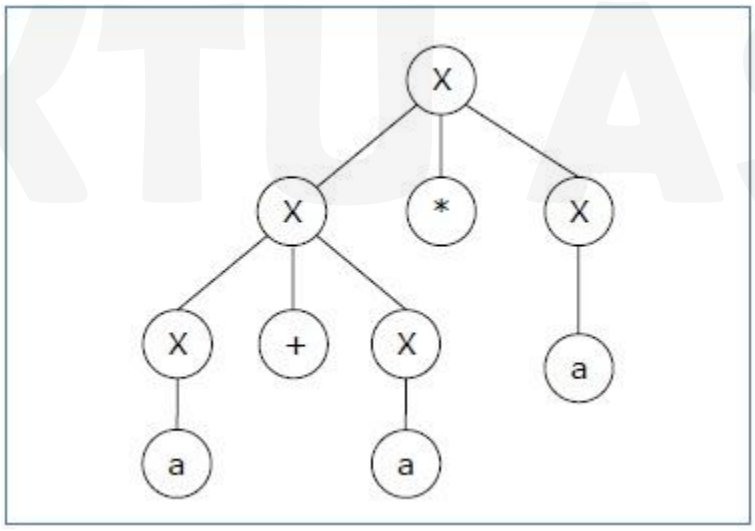
Derivation 1 − X → X+X → a +X → a+ X*X → a+a*X → a+a*a

Parse tree 1 −



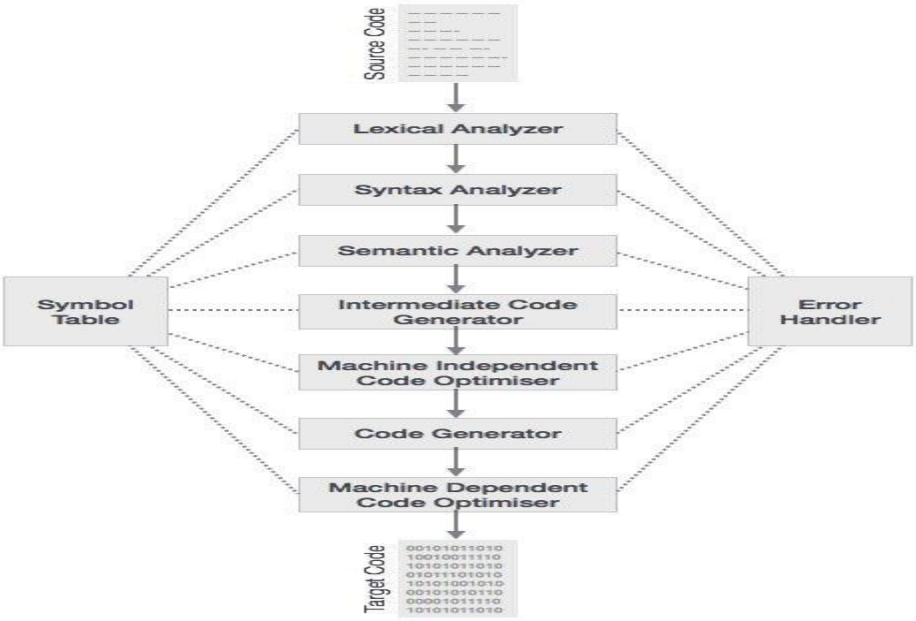
Derivation 2 − X → X*X → X+X*X → a+ X*X → a+a*X → a+a*a

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)

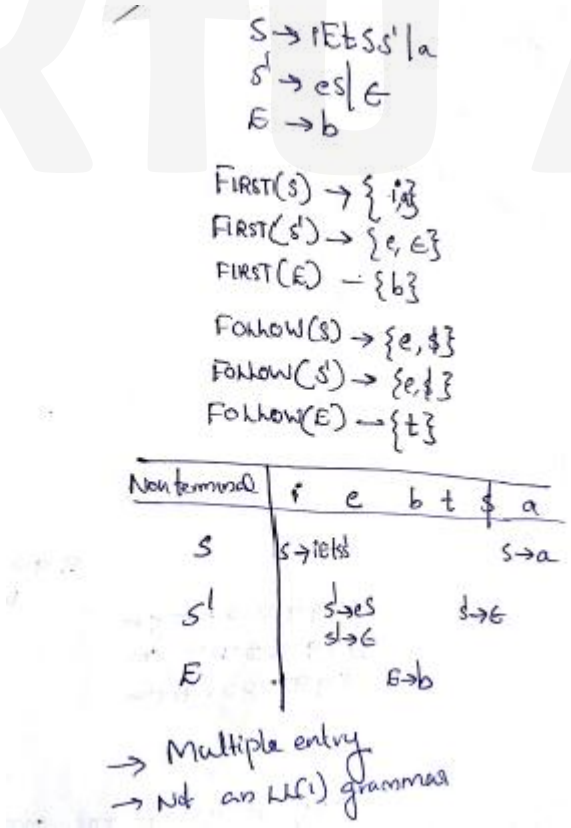
a) To specify a lexical analyzer we need a state machine, sometimes called Transition Diagram(TD), which is similar to a FSA.Transition Diagrams depict the actions that take place when the lexer is called by the parser to get the next token . FSA accepts or rejects a string. TD reads characters until 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).

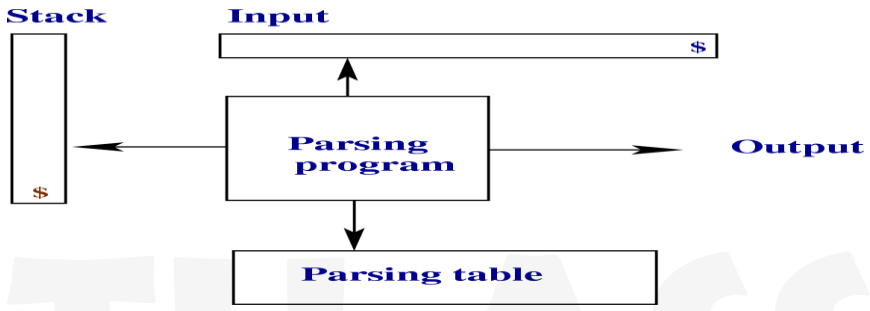
b)



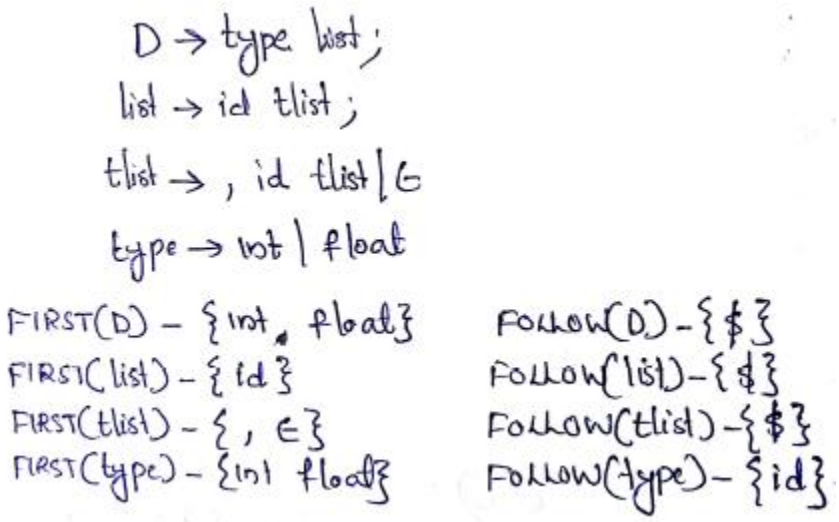
7**)**

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 $.

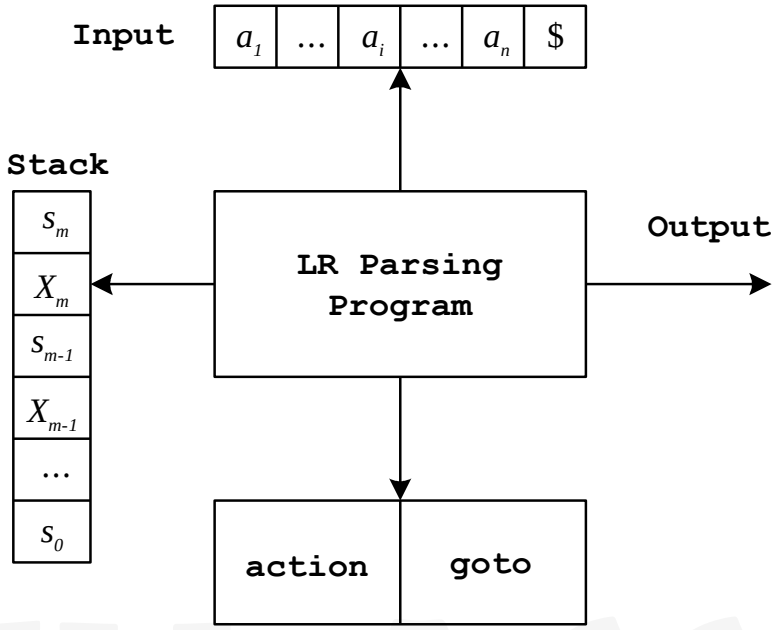


b)



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.

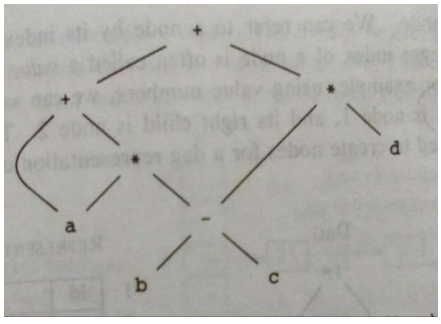
L-Attributed Definitions

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

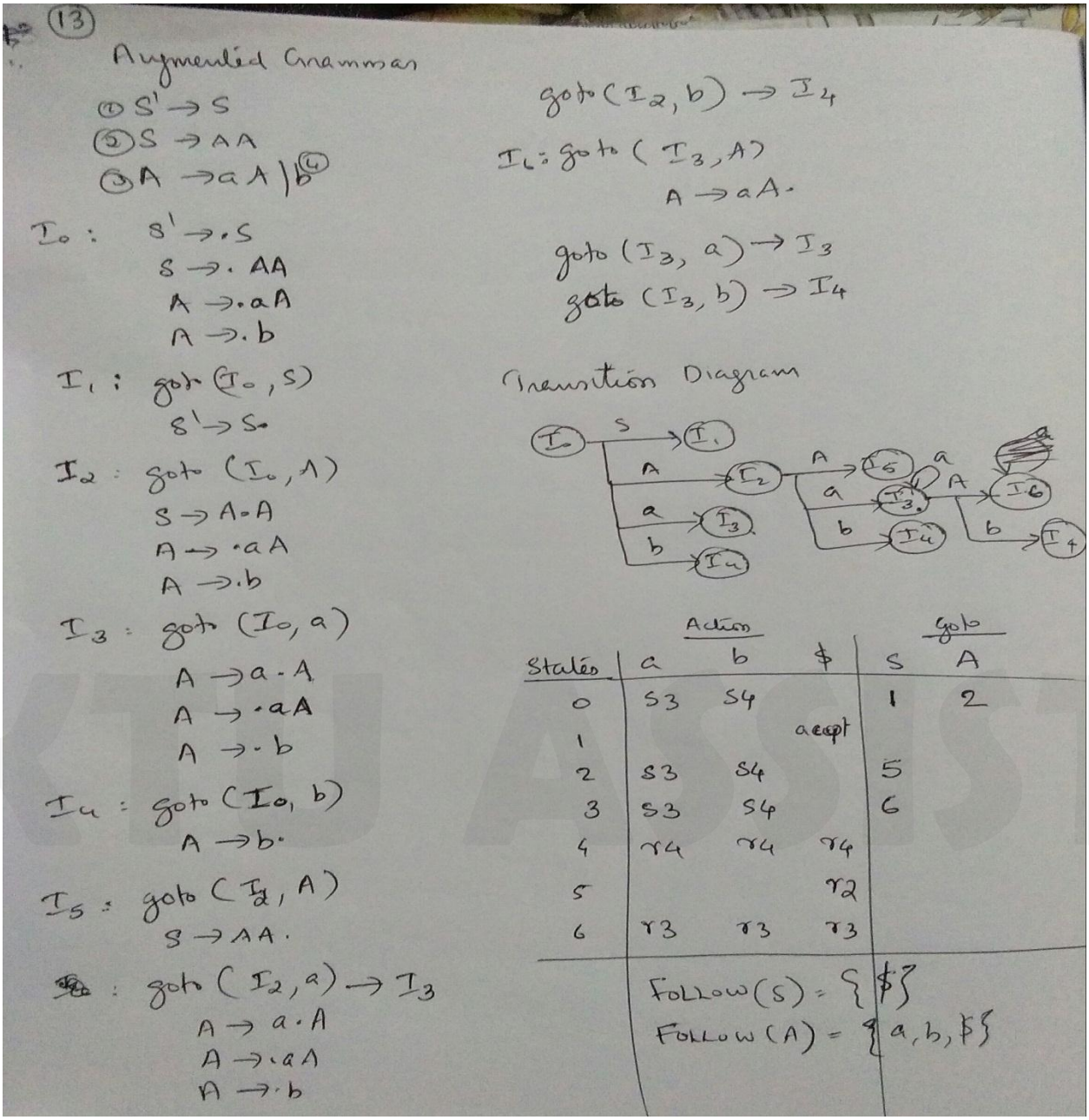
$$E \rightarrow EAE \mid (E) \mid -E \mid \mathbf{id}$$
$$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$$

-
- 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

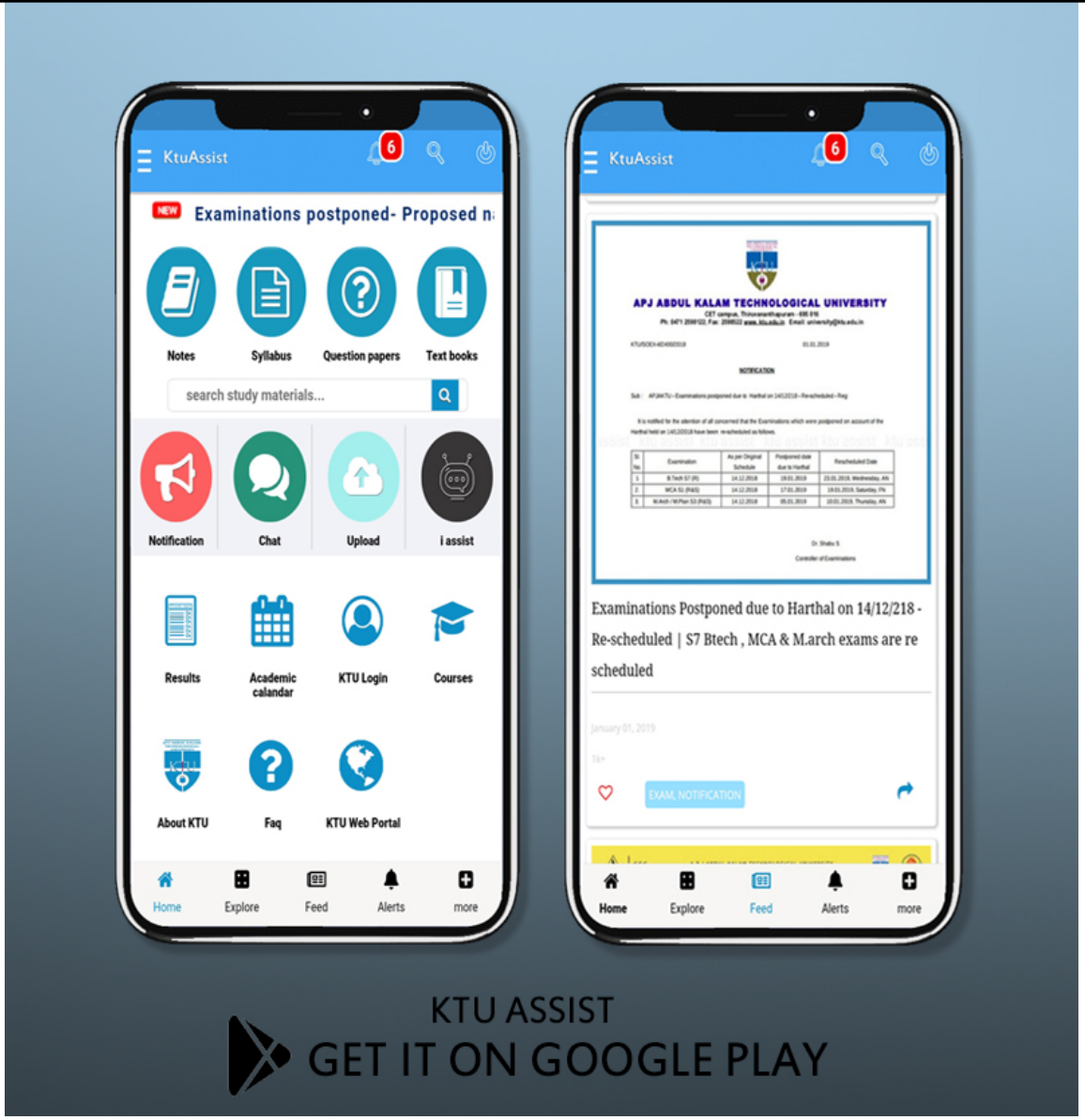
even register holds the remainder

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.



END

facebook.com/ktuassist

instagram.com/ktu_assist