COMPILER DESIGN

NOTE:

WBUT course structure and syllabus of 7th Semester has been changed from 2013. **LANGUAGE PROCESSOR [CS 701]** has been redesigned as **COMPILER DESIGN [CS 702]** in present curriculum. Taking special care of this matter we are providing the relevant WBUT questions and solutions of **LANGUAGE PROCESSOR** papers from 2006 to 2012, so that students can get an idea about university questions patterns.

INTRODUCTION TO COMPILER

Multiple Choice Type Questions

1. Cross-compiler is a compiler [WBUT 2006]
   a) which is written in a language that is different from the source language
   b) that generates object code for its host machine
   c) which is written in a language that is same as the source language
   d) that runs on one machine but produces object code for another machine

Answer: (d)

2. Consider the statement "if(x>=10)", where 'if' has been misspelled. The error is detected by the compiler in the phase [WBUT 2006]
   a) lexical analysis b) syntax analysis
   c) semantic analysis d) syntactic analysis

Answer: (c)

3. A language L allows declaration of arrays whose sizes are not known during compilation. It is required to make efficient use of memory. Which one of the following is true? [WBUT 2007]
   a) A compiler using static memory allocation technique can be written for L
   b) A compiler cannot be written for L, an interpreter must be used
   c) A compiler using dynamic memory allocation technique can be written for L
   d) None of these

Answer: (c)

4. The role of the preprocessor is [WBUT 2008]
   a) Produce output data b) Produce output to compilers
   c) Produce input to compilers d) None of these

Answer: (c)

5. Which one of the following error will not be detected by the compiler?
   a) Lexical error b) Syntactic error [WBUT 2010, 2012]
   c) Semantic error d) Logical error

Answer: (d)

6. Which is used to keep track of currently active activations? [WBUT 2012]
   a) control stack b) activation c) executio ı d) symbol

Answer: (a)

Short Answer Type Questions

1. With the help of a block diagram, show each phase including symbol table and error handler of a compiler. [WBUT 2007]

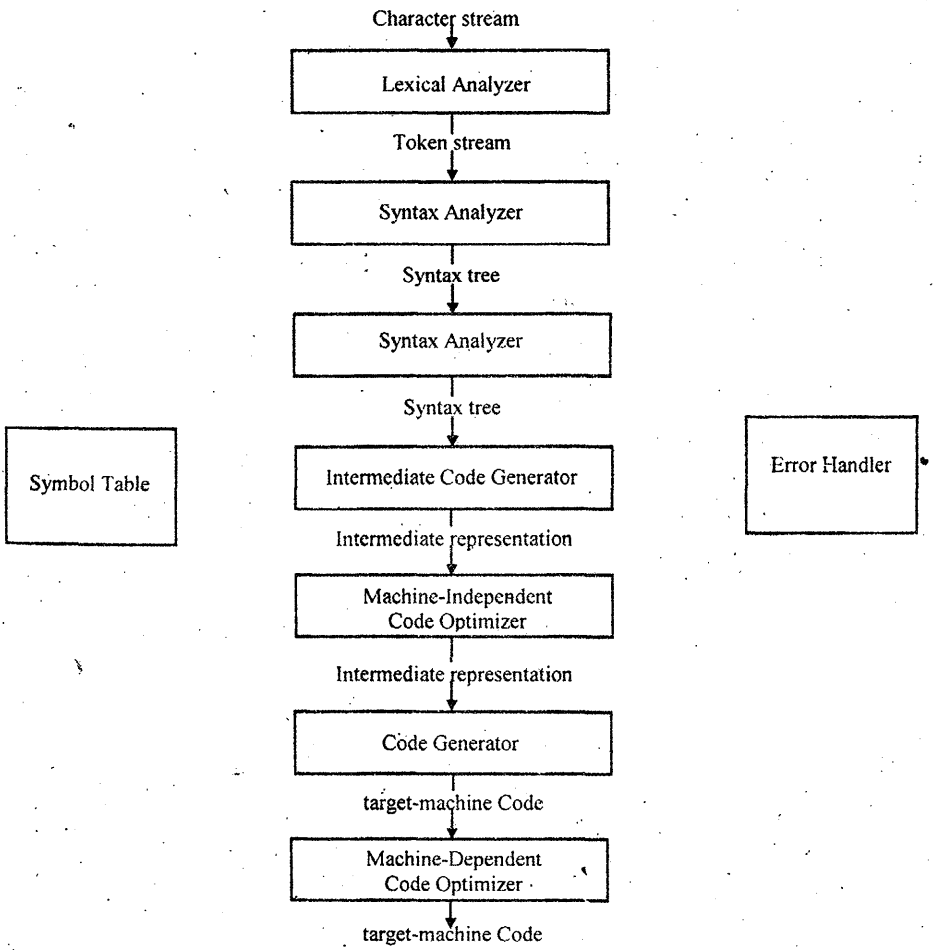
Answer:

Various Stages of Compiler:

Generally we have treated a compiler as a single box that maps a source program into a semantically equivalent target program. If we open up this box a little, we see that there are two parts to this mapping: analysis and synthesis.

The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

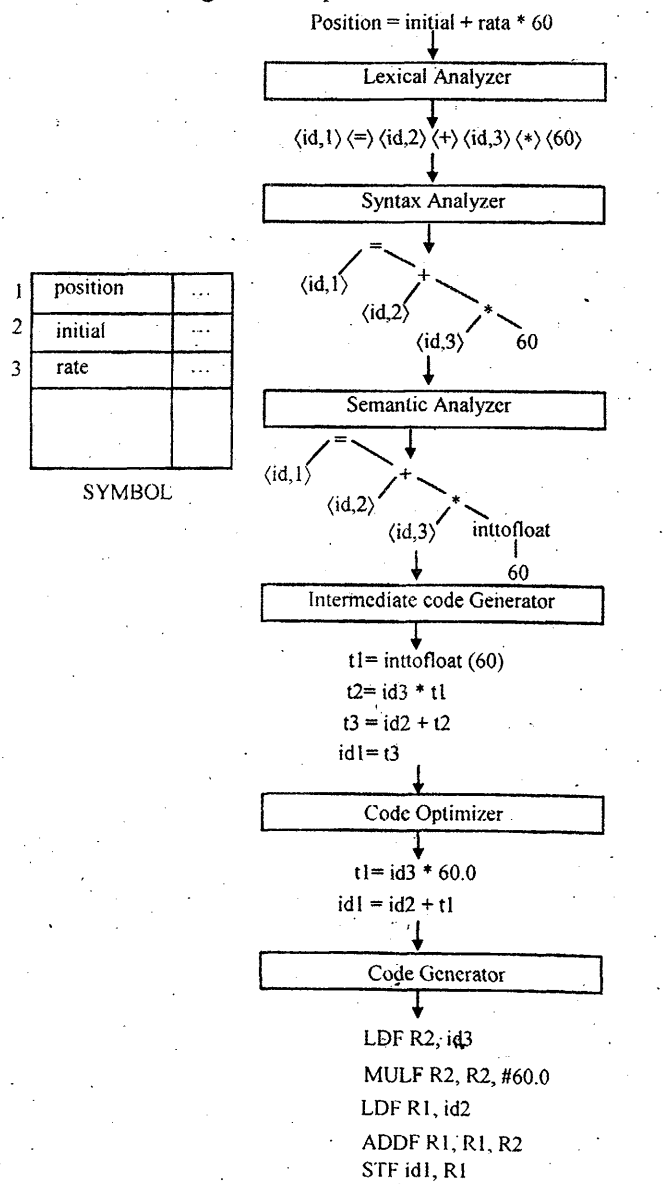
If we examine the compilation process in more details, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases in show in figure below. In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly. The symbol table, which stores information about the entire source program, is used by all phases of the compiler.



Phases of a compiler

Some compilers have a machine-independents optimization phase between the front end and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target program than it would have otherwise produced from an optimization intermediate representation. Since optimization is optional, one or the two optimization phases shown in above figure may be missing.

All phases are described using an example:



Translation of an assignment statement

2. What are the analysis phase and synthesis phase of an assemble? [WBUT 2010]

Answer:

There are two main stages in the compiling process --- analysis and synthesis.

The analysis stage breaks up the source program into pieces, and creates a generic (language-independent) intermediate representation of the program. Then, the synthesis stage constructs the desired target program from the intermediate representation. Usually, the analysis stage of a compiler is called its front-end and the synthesis stage is its back-end.

There are four phases in the analysis stage of compiling:

Here, the stream of characters making up a source program is read from left to right and grouped into *tokens* --- sequences of characters having a collective meaning.

Here, the tokens found are grouped together using a context-free grammar. The output of this parsing phase is called a *parse tree*.

Here, the parse tree is checked for semantic errors i.e., detection of such things like the use of undeclared variables, function calls with improper arguments, access violations, incompatible operands, type mismatches, etc.

Here, the intermediate representation of the source program is created, usually in a form like *three-address code*.

The synthesis stage can have up to three phases:

Here, code in the intermediate representation (e.g., Three Address Code) is converted into a streamlined version, still in the intermediate representation, but with attempts to produce the smallest, fastest and most efficient running result (collectively called *optimization*).

Here, the target program is generated, as machine code or assembly code. Memory locations are selected for each variable and instructions are chosen for each operation.

This is an optional optimization pass following code generation and transforms the object code into tighter, more efficient object code by considering features of the hardware to make efficient usage of processor(s), registers, specialized instructions, pipelining, branch prediction and other peephole optimizations.

3. Describe analysis phase of a compiler in respect of the following example.
position = initial + rate * 60 [WBUT 2011]

Answer:

The analysis phase of a compiler, also known as its "front-end", breaks up the source program into pieces, and creates a generic (language independent) intermediate representation of the program. There are four stages in the analysis phase of compiling:

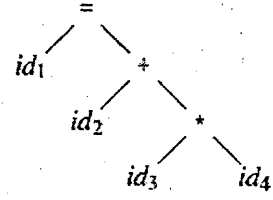
Lexical Analysis: Here, the stream of characters making up a source program is read from left to right and grouped into tokens—sequences of characters have a collective meaning.

For the given example, lexical analysis will break the sentence into a stream of seven tokens:

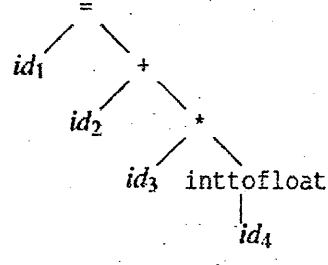
id1 = id2 + id3 _ id4

id1, id2 and id3 will be 'bound' to Symbol Table entries corresponding to variables position, initial, and rate, respectively, while id4 will be bound to the constant 60.

Syntax Analysis or Parsing: Here, the tokens found are grouped together using a context-free grammar. The output of this parsing phase is called a parse tree. The parse tree for the given example will be:



Semantic Analysis: Here, the parse tree is checked for semantic errors i.e., detection of such things like the use of undeclared variables, function calls with improper arguments, access violations, incompatible operands, type mismatches, etc. Assuming that in our code the variables are of float type, the semantic analysis will insert a int-to-float conversion of the constant 60 to give:



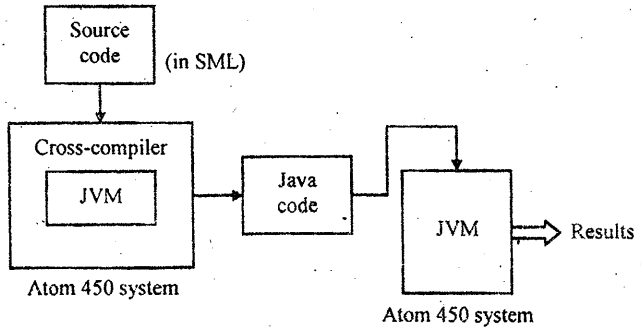
Intermediate Code Generation: Here, the intermediate representation of the source program is created, usually in a form like three-address code. The set of three-address codes for the example will be

t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

4. What is a cross-compiler? Create a cross-compiler for SML (Sensor Mark-up Language) using Java compiler, written in ATOM-450, producing code in ATOM-450 and a SML language producing code for XML written in Java. [WBUT 2012]

Answer:

A cross-compiler is a piece of software which runs on a particular environment (processor + OS) and generates code that can run on another environment. For example a 'C' cross-compiler running on X86-Windows generating code for ARM-Linux.



Long Answer Type Questions

1. a) Explain the different phases of a compiler, showing the output of each phase, using the example of the following statement: **[WBUT 2009]**

Position: = initial + rate * 60

OR,

How the following statement is translated via the different phases of compilation?

Position : = initial + rate *70. **[WBUT 2010]**

Answer:

Refer to Question No. 3 of Short Answer Type Questions.

b) Compare compiler and interpreter. **[WBUT 2009]**

OR,

Distinguish between interpreter and compiler. **[WBUT 2012]**

Answer:

Compiler scans whole code at once so it is fast on the other hand interpreter scans the byte code line by line and converts in machine code so it is slow.

Compiler creates executable file that runs directly on the CPU but interpreter does not create direct executable file.

Compiler uses more memory - all the execution code needs to be loaded into memory.

Interpreter uses less memory, source code only has to be present one line at a time in memory.

2. a) Apply all the phases of compiler and show the corresponding output in every phase for the following code of the source program:

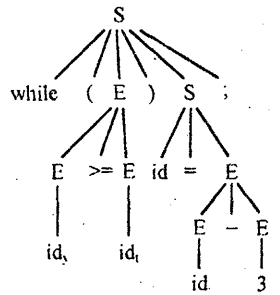
while (y ≥ t) y = y – 3; **[WBUT 2012]**

Answer:

Lexical analysis generates the tokens

'while', 'C', 'id_y', '>=', 'id_t', '=', 'id_y', '–', '3' and ';'

The syntax analysis stage generates the parse tree



The intermediate code generates 3-address code as:

100 : if (y>=t) goto 102

101 : goto 105

102 : T=y–3

103 : Y=T

104 : goto 100

The code optimizer optimized this code to

100 : if (y<t) go to 103

101 : y=y–3

102 : goto 100

The code generator can translate this into machine language statements of the target machine.

b) What do you mean by passes of compiler? Explain advantages and disadvantages of one-pass and two-pass over each other. [WBUT 2012]

Answer:

One-pass compiler is a compiler that passes through the parts of each compilation unit only once, immediately translating each part into its final machine code. This is in contrast to a **multi-pass compiler** which converts the program into one or more intermediate representations steps in between source code and machine code, and which reprocesses the entire compilation unit in each sequential pass.

Advantages

One-pass compilers are smaller than Two-pass compiler and faster than multi-pass compilers.

Disadvantages

One-pass compilers are unable to generate as efficient programs, due to the limited scope of available information. Many effective compiler optimizations require multiple passes

over a basic block, loop, subroutine, or entire module. Some require passes over an entire program. Some programming languages simply cannot be compiled in a single pass, as a result of their design. For example PL/I allows data declarations to be placed anywhere within a program, so no code can be generated until the entire program has been scanned. In contrast, many programming languages have been designed specifically to be compiled with one-pass compilers, and include special constructs to allow one-pass compilation.

3. Write short note on the following:

a) Symbol table organization **[WBUT 2006, 2007]**

OR,

Symbol Table **[WBUT 2009, 2010, 2011, 2012]**

b) Cross Compiler **[WBUT 2008, 2010]**

c) Chomsky classification of grammar **[WBUT 2012]**

Answer:

a) Symbol table organization:

A symbol table is a data structure, where information about program objects is gathered. It is used in both the lexical analysis and code generation phases. The symbol table is built up during the lexical (and partially during) syntactic analysis. It provides help for other phases during compilation like during Semantic analysis in resolving type conflict, during Code generation in evaluating how much and what type of run-time space is to be allocated, during Error handling to produce the most appropriate error message, etc.

Symbol table management refers to the symbol table's storage structure, its construction in the analysis phase and its use during the whole compilation.

Requirements for symbol table management (or symbol table organization) include:

- quick insertion of an identifier
- quick search for an identifier
- efficient insertion of information (attributes) about an identifier
- quick access to information about a certain identifier
- Space-efficiency and time-efficiency

There may be several possible implementations of a Symbol Table:

Unordered list: Appropriate for a very small set of variables, that too for a simple symbol table.

Ordered linear list: Entries are sorted/indexed by symbol names. Insertion is expensive time-wise, but implementation is relatively easy.

Binary search tree: A variation of a sorted list with O(log n) time per operation for n variables.

Hash table: The most commonly used implementation with very efficient insertion/lookup, provided the memory space is adequately larger than the number of variables.

b) Cross Compiler:

A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is run. Cross compiler tools are used to generate executables for embedded system or multiple platforms. It is used to compile for a platform upon which it is not feasible to do the compiling, like microcontrollers that don't support an OS.

The fundamental use of a cross compiler is to separate the build environment from the target environment. This is useful in a number of situations:

- Embedded computers where a device has extremely limited resources.
- Compiling for multiple machines—a company supporting several different versions of OS, a single build environment can be set up to compile for each of these targets.
- Compiling on a server farm.
- Compiling native code for emulators.

c) ***Type 0***, Grammar is any phrase structure grammar without any restrictions.

To define the other types we need a definition.

In a production of the form $\phi A\psi \rightarrow \phi\alpha\ \psi$, where A is a variable, ϕ is called the left context ψ, the right context and $\phi\alpha\ \psi$ the replacement string.

Example: (a) i) In abAbcd->abABbcd, ab is the left context, bcd is the right context, α=ab

ii) In AC->A, A is the left context, ^ is the right context. α=^ The production simply erases C when the left context is A and the right context is ^.

iii) a) In C-> ^, the left and right context are ^.α=^ The production simply erases C when in any context.

Type 1, Grammars is a production of the form $\phi A\psi \rightarrow \phi\alpha\ \psi$ is called type 1 production if $\alpha \neq$ ^. In type 1 production erasing of A is not permitted.

A grammar is called ***Type 1*** or ***context sensitive*** or ***context dependent*** if all the productions are Type 1 production. The production S->^ is also allowed in a Type 1 grammar, but in this case S does not appear on the R.H.S of any production.

Example: (b) i) In aAbcD->abcDbcD is the type 1 production. a, bcD are the left context, and right context.

ii) AB -> AbBc is a type 1 production. The left context is A and right context is ^.

iii) A-> abA is a type 1 production. The both left and right context is ^.

Type 2, Grammars is a production of the form A->α ,where $A,B \in V_N$ and $\alpha \in (V_N \cup \Sigma)^*$ In other words, the L.H.S has no left context or right context.

A grammar is called ***Type 2*** or ***context free grammar*** if all the productions are Type 2 productions.

Example: (c) S ->Aa, A ->a, B ->abc, A ->^, are type 2 productions.

Type 3, Grammars is a production S -> ^ is allowed in type 3 grammar, but in this case S does not appear on the right-hand side of any production.

A grammar is called ***Type 3*** or ***regular grammar*** if all the productions are Type 3 productions.

Example: (d) S ->aS, A ->a, A ->aB, are type 3 productions.

Language: A language L over an alphabet A is a collection of words on A, A^* denotes the set of all words on A. Thus a language L is simply a subset of A^*.

e.g., $\Sigma = \{a, b\}$ then

$\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, aab, bbb \ldots\}$

The set

$\{a, aa, aab\}$

is a language on Σ. Because it has a finite number of sentences, we call it a finite language. The set

$L = \{a^n b^n : n \geq 0\}$

is also a language on Σ. The string aabb and aaabbb are in the language L, but the string abb is not in L. This language is infinite.

LEXICAL ANALYSIS

Multiple Choice Type Questions

1. The regular expression 0*(10*)* denotes the same set as: [WBUT 2008]
   a) (1 * 0) * 1* b) 0+(0+10)*
   c) (0+1) * 10(0+1)* d) None of these
Answer: (b)

2. If x is a terminal then FIRST (x) is [WBUT 2008, 2010]
   a) ε b) { x } c) x* d) none of these
Answer: (b)

3. The regular expression (*a* | *b*)* *abb* denotes [WBUT 2009]
   a) all possible combinations of *a*'s and *b*'s
   b) set of all strings endings with *abb*
   c) set of all strings starting with *a* and ending with *abb*
   d) none of these
Answer: (d)

4. In a programming language, an identifier is permitted to be a letter followed by any number of letters or digits. If L and D denote the set of letters and digits respectively, which of the following expressions defines an identifier? [WBUT 2009]
   a) $(L \cup D)^+$ b) L.(L ∪ D)* c) (L.D)* d) L.(L.D)*
Answer: (b)

5. The following productions of a regular grammar generates a language L.
S→aS | bS | a | b
The regular expression for L is [WBUT 2009]
   a) $a+b$ b) $(a+b)^*$ c) $(a+b)(a+b)^*$ d) $(aa+bb)a^*$
Answer: (c)

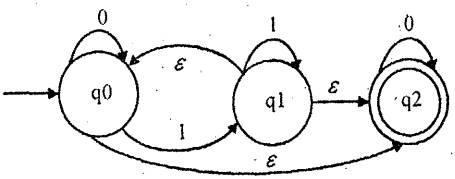
6. White spaces and tabs are removed in [WBUT 2011]
   a) Lexical analysis b) Syntax analysis
   c) Semantic analysis d) all of these
Answer: (a)

Short Answer Type Questions

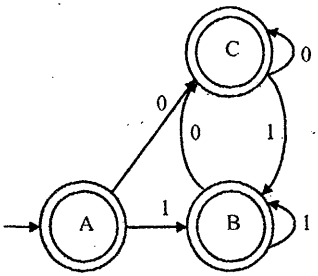
1. Give the NFA for the Regular Expression $(0*1^{+})*0*$. Then find a DFA for the same language. [WBUT 2006]

Answer:

The NFA is easy to find and is



From this, the required DFA can be easily constructed as:



2. When does a lexical analyzer report an error? Write some error recovery actions taken by a lexical analyzer when it finds an error. [WBUT 2006]

Answer:

A lexical analyzer reports error when it is unable to proceed because none of the provided patterns for token matches a prefix of the remaining input.

The simplest error recovery strategy, called the "panic mode" recovery, is where successive characters are deleted from the remaining input until a well formed token is found. Other error-recovery actions could be:

- deleting an extraneous character
- inserting a missing character
- replacing an incorrect character by a correct character
- transposing two adjacent characters

3. What do you understand by terminal table and literal table? [WBUT 2006, 2008, 2010, 2011]

Answer:

During lexical analysis, after a token is identified, first the terminal table is examined. If the token is not found in the table, a new entry is made. Only the 'name' attribute of the token goes in, the remaining information is inserted in later phases. On the other hand,

numbers, quoted character strings and other self-defining data are classified as 'literals' and go to the literal table. If the literal is not found in the table, a new entry is made. The lexical analyzer can determine all the attributes of a literal as well as its internal representation, by looking at it.

4. What is a lookahead operator? Give an example. With the help of the look ahead concept show how identifiers can be distinguished from keywords.

[WBUT 2007, 2009]

Answer:

For certain programming languages, the lexical analyzers may have to look ahead a few symbols beyond the end of a lexeme before they can determine a token with certainty. In LEX, the lookahead operator is '/'.

For example, in Fortran-77, the token IF can be the keyword "IF" or it can be the name of a function/array as in IF(10, 5). It is the keyword only if a pair of parentheses has been seen, followed by a letter. Hence, in a Lexical

Analyzer for Fortran-77, we would specify the keyword IF as:

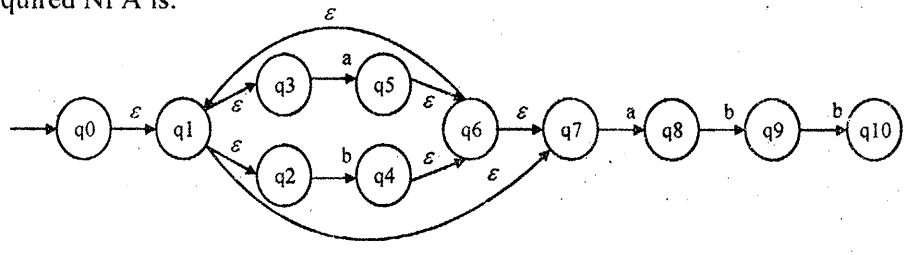
IF / : {letter} where we assume that the definition {letter} means any letter.

5. Give the NFA for the following Regular Expression. Then find a DFA for the same language.

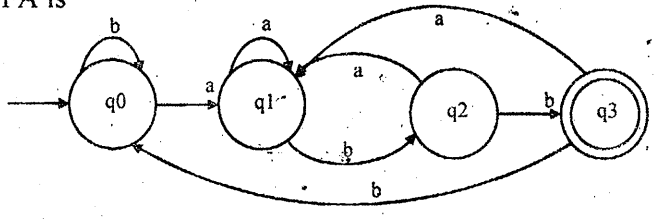
(a|b) * abb **[WBUT 2008]**

Answer:

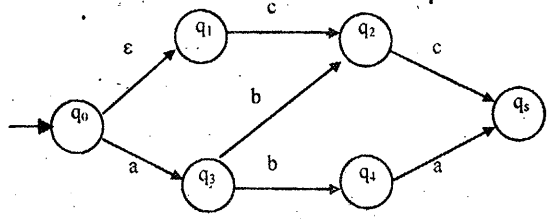
The required NFA is:



The equivalent DFA is



6. Convert the none-deterministic FA below to its equivalent DFA. **[WBUT 2009**



Answer:

We start with the set of states $\{q_0, q_1\}$, which is the ε-closure of the start state of the NF and hence the start state of the required DFA.

Now $\delta(\{q_0, q_1\}, a) = \{q_3\}$ *and* $(\{q_0, q_1\}, c) = \{q_2\}$.

Thus we get two new states in the DFA - $\{q_3\}$ and $\{q_2\}$.

Now $\delta(\{q_3\}, b) = \{q_2, q_4\}$ *and* $\delta(\{q_2\}, c) = \{q_5\}$.

Also, $\delta(\{q_2, q_4\}, a) = \delta(\{q_2, q_4\}, c) = \{q_5\}$ *but* $\{q_5\}$ has no transition.

Hence, remaining states as:

$\{q_0, q_1\} \rightarrow A$

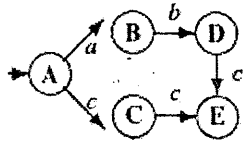
$\{q_3\} \rightarrow B$

$\{q_2\} \rightarrow C$

$\{q_2, q_4\} \rightarrow D$

$\{q_5\} \rightarrow E$

The required DFA is:



7. Consider the following lexically nested C code: **[WBUT 200**

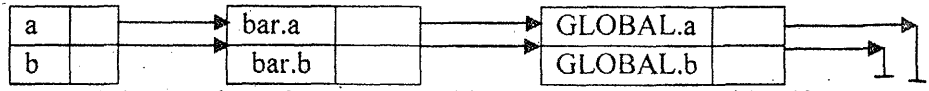
```
int a, b;
int foo( ) {int a, c;}
int bar( ) { int a, b; / * HERE*/}
```

a) How can symbol tables represent the state of each scope at the point mark HERE? Draw a diagram.

b) What symbols are visible / not visible at point HERE?

Answer:

a) The symbol table can maintain a "stack" of entries corresponding to each identifi The top of the refers to the most recent entry for the identifier and hence the one in sco In the given problem, there are three variables — a, b and c. At the point HERE howev c, is neither in scope nor does the extent of c include the point. Hence, the appearance the symbol table at point HERE is:



Here, we have used the notation <Scope>.<variable> to indicate which identifier we are referring to where <Scope> can be either GLOBAL or the name of a function.

b) Symbols visible at HERE are bar.a and bar.b. The symbols GLOBAL.a and GLOBAL.b are not visible and of course foo.a and foo.c are non-existent.

8. a) How does Lexical Analyzer help in the process of compilation? Explain it with an example. **[WBUT 2009]**

Answer:

The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form.

⟨token-name, attribute-value⟩

That it passes on the subsequent phases, syntax analysis. In the token, The first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute value points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis and code generation.

For example, suppose a source program contains the assignment statement

Position = initial + rate * 60

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. Position is a lexeme that would be mapped into a token ⟨**id**,1⟩, where **id** is an abstract symbol standing for identifier and 1 points to the symbol-table entry for position. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.
2. The assignment symbol = is a lexeme that is mapped into the token ⟨=⟩. Since this token needs no attribute-value, we have omitted the seconed component. We could have used any abstract symbol such as **assign** for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.
3. Initial is a lexeme that is mapped into the token ⟨**id**,2⟩, where 2 points to the symbol-table entry for initial.
4. + is a lexeme that is mapped into the token ⟨+⟩

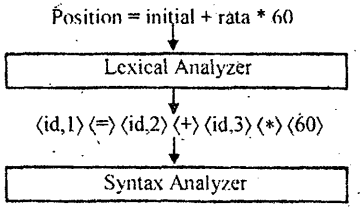
5. Rate is a lexeme that is mapped into the token ⟨**id**,3⟩, where 3 points to the symbol-table entry for rate.
6. * is a lexeme that is mapped into the token⟨*⟩
7. 60 is a lexeme that is mapped into the token⟨60⟩.

Blanks separating the lexemes would be discarded by the lexical analyzer.

Figure below shows the representation of the assignment statement (1.1) after lexical analysis as the sequence of tokens

⟨**id**,1⟩ ⟨=⟩ ⟨**id**,2⟩ ⟨+⟩ ⟨**id**,3⟩ ⟨*⟩ ⟨60⟩

In this representation, the token names = , +, and * are abstract symbols for the assignment, addition, and multiplication operators, respectively.



Working Principle of Lexical Analyzer

b) Consider the following conditional statement: **[WBUT 2009]**

if (x > 3) then y = 5 else y = 10

From the above statement how many tokens are possible and what are that?

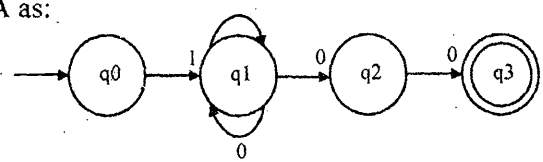
Answer:

There are 14 tokens.

They are,

1. keyword(if)
2. left-parenthesis
3. identifier(variable x)
4. relational-operator(greater-than)
5. identifier(constant 3)
6. right-parenthesis
7. keyword(then)
8. identifier(variable y)
9. operator(assignment)
10. identifier(constant 5)
11. keyword(then)
12. identifier(variable y)
13. operator(assignment)
14. identifier(constant 10)

9. Convert the following NFA into its equivalent DFA:
The set of all strings with 0 and 1, beginning with 1 & ending with 00. [WBUT 2010]
Answer:
We can draw the NFA as:



The states of the required DFA are from the power set of $\{q_0, q_1, q_2, q_3\}$. We will not consider the ``dead state" or transitions in an out of such a state. The start state of the required DFS is $\{q_0\}$.

Now,

$\delta(\{q_0\},1) = \{q_1\}$

$\delta(\{q_1\},1) = \{q_1\}$

$\delta(\{q_1\},0) = \{q_1, q_2\}$

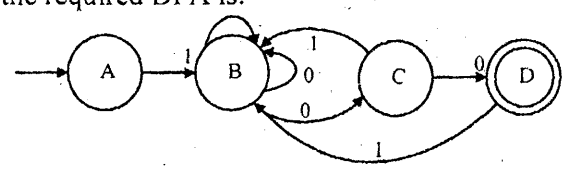
$\delta(\{q_1, q_2\},1) = \{q_1, q_2\}$

$\delta(\{q_1, q_2\},0) = \{q_1, q_{2,3}\}$

$\delta(\{q_1, q_2, q_3\},1) = \{q_1\}$

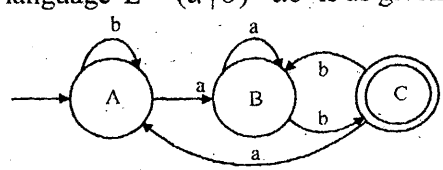
$\delta(\{q_1, q_2, q_3\},0) = \{q_1, q_2, q_3\}$

Hence, the states of the required DFA are $\{q_0\}$ (let's call it A), $\{q_1\}$ (let's call it B), $\{q_1, q_2\}$ (let's call it C) and $\{q_1, q_2, q_3\}$ (let's call it D which is also the final state of the DFA). Hence, the required DFA is:



10. Construct DFA directly from [not by generating NFA] the regular expression $L = (a \mid b)^* ab$ [WBUT 2010, 2011]

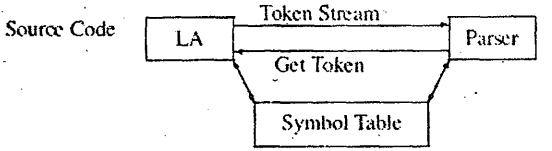
Answer:
The required DFA for the language $L = (a \mid b)^* ab$ is as given below:



11. Describe with diagram the working process of lexical Analyzer. [WBUT 2011]

Answer:

The operation of the Lexical Analyzer is explained in the diagram below.



The Lexical Analyzer breaks down the source program into tokens like keywords, constants, identifiers, operators and other simple tokens. A token is the smallest piece of text that the language defines.

Keywords are words the language defines, and which always have specific meaning in the language like **if, else, int, char, do, while, for, struct, return**, etc in C/C++.

Constants are the literal valued items that the language can recognize, like numbers, strings, and characters.

Identifiers are names the programmer has given to something. These include variables, functions, classes (in C++ and other object-oriented languages), enumerations, etc. Each language has rules for specifying how these names can be written.

Operators are the mathematical, logical, and other operators that the language can recognize.

For identifiers, the Lexical Analyze also uses a "Symbol Table" — a data structure it shares with the Parser and subsequent phases of compilation.

12. What is error handling? Describe the Panic Mode and Phrase level error recovery technique with example. [WBUT 2011]

Answer:

Programs submitted to a compiler often have errors of various kinds. When a compiler detects an error, i.e., when the symbols in a sentence do not match the compiler's current position in the syntax diagram, the error handler is invoked. The error handler warns the programmer by issuing an appropriate message and then attempts to recover from it so that it can detect more errors.

The simplest form of error recovery technique is "panic mode". A small set of 'safe symbols' are used to delimit 'clean points' in the input. When an error occurs, a panic mode recovery algorithm deletes input tokens until it finds a safe symbol, then backs the parser out to a context in which that symbol might appear. In the following fragment of Pascal code: if a b then x else y;

Compiler discovers the error at b and a panic-mode recovery algorithm very likely skips forward to the semicolon, thereby missing the then. When the parser later finds the else, it again produces a spurious error message.

In phrase-level error recovery, the quality of recovery is improved by employing different sets of safe symbols in different contexts. When compiler discovers an error in an

expression, it deletes input tokens until it reaches something that is likely to follow an expression. This more local recovery is better than always backing out to the end of the current statement because it gives the compiler the opportunity to examine the parts of the statement that follow the erroneous expression. In our above example, a phrase level recovery would use the then and else tokens as the next safe symbols and give a more realistic error message.

13. Define regular expression. Write the regular expression over alphabet $\{a,b,c\}$ containing at least one 'a' and at least one 'b'. What is dead state? Explain with suitable example. **[WBUT 2012]**

Answer:

A regular expression (RE) is defined recursively as follows:

ε is an RE

a is an RE, where 'a' $\in \varepsilon$

If R_1 and R_2 are REs, then so are

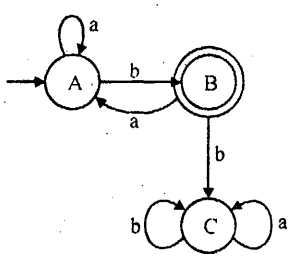
$R_1 + R_2$, $R_1 R_2$ and R^*.

Required RE is:

$$\left(\left((a+b+c)^* a(a+b+c)^* b(a|b|c)^*\right)\left((a+b+c)^* b(a+b+c)^* a(a+b+c)^*\right)\right)$$

In a finite state automata; a dead state is one such that once that state is reached, any input keeps the automata in that state is reached, any input keeps the automata in that state.

Example:



Here, C is a dead-state.

14. How does lexical analyzer help in the process of compilation? Consider the following statement and find the number of tokens with type and value as applicable:

```
void main ( )
    {
        int x;
        x = 3;
    }
```

[WBUT 2012]

Answer:

The lexical analyzer breaks down the source code into a stream of 'tokens'. The grammar of a language is expressed in terms of tokens where similar types of entities are treated similarly. For examples, variables and constants are considered as "identifiers". This helps in the compiler being based on a language of token types, which is not dependent on the particular source code being compiled.

There are total 13 tokens:

Void : Keyword
main : identifier
(
)
{
int : keyword
x : identifier
;
x
=
3 : identifier
;
}

Long Answer Type Questions

1. Consider the (very artificial) language, over the alphabet of letters and digits and the dollar ($), having the following three kinds of tokens: numbers, consisting of one or more consecutive digits; short identifiers, consisting of a single letter and long identifiers; consisting of one or more letters followed by a single $.

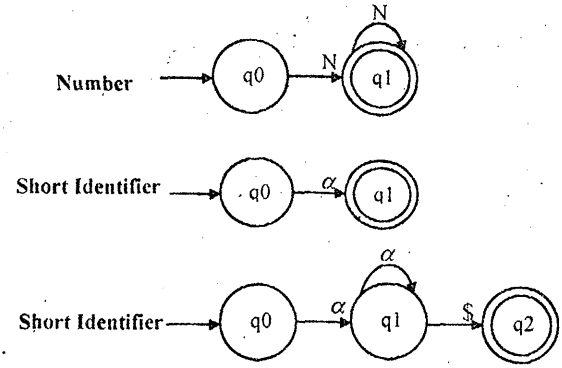
i. **Write down a regular expression for reach of the three token patterns.**
ii. **Draw DFA for each expression.**
iii. **Combine the DFA's into a common NFA with a common start state and distinct final states for each token.**
iv. **Convert your combined NFA into a DFA using whatever algorithm you like.**
v. **Suppose you use this DFA to perform lexical analysis, backtracking to the most recently encountered final state when necessary.**

Give an example of an input that will require backtracking and rereading exactly 5 characters. [WBUT 2007]

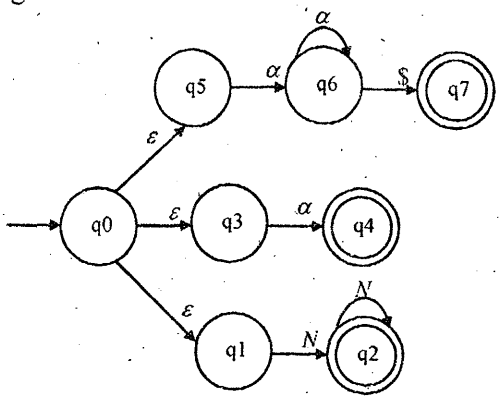
Answer:

The required regular expressions (in lex syntax are [0-9]+ for numbers, [A-Za-z] for short identifiers and [A-Za-z]+$ for long identifiers.

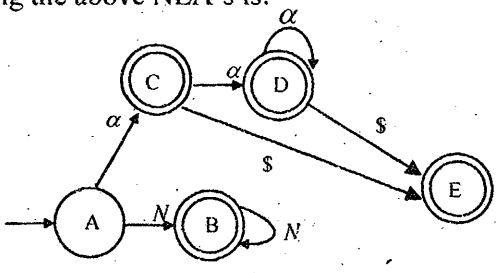
The required DFA-s are (where N is a shortcut for any numerical digit, a is shortcut for any letter):



The NFA after combining the above DFA-s is:



The DFA after converting the above NEA-s is:



Since a whitespace is not a part of the alphabet, if the lexical analyzer encounters the input string abcde9 (say), it will read upto the last 9 before realizing that it is not a long identifier and hence backtrack back 5 characters and return a as a short identifier.

2. Write short notes on the following:

a) LEX **[WBUT 2007, 2009, 2012]**

b) YAAC **[WBUT 2006, 2007, 2009, 2012]**

c) Thompson's Construction Rule **[WBUT 2007]**

d) LEX and YAAC **[WBUT 2010, 2011]**

Answer:

a) LEX:

Lex is a program that generates lexical analyzers. Lex is commonly used with the yacc parser generator. Lex is the standard lexical analyzer generator on many Unix systems. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexical analyzer in the C programming language.

The structure of a lex file is intentionally similar to that of a yacc file; files are divided up into three sections, separated by lines that contain only two percent signs, as follows:

Definition section

%%

Rules section

%%

C code section

The definition section is the place to define macros and to import header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file. The rules section is the most important section; it associates patterns with C statements. Patterns are simply regular expressions. When the lexer sees some text in the input matching a given pattern, it executes the associated C code. This is the basis of how lex operates. The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file and link it in at compile time.

b) YAAC:

yacc assumes that the user has supplied a lexical analyzer which is an integer-valued function called yylex. The function returns an integer, the token number, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable yylval. The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place.

In normal practice, one uses LEX to generate the lexical analyzer which is then piggybacked into the parser code generated by yacc.

c) Thompson's Construction Rule:

Given any RE, it is possible to algorithmically construct an NFA such that the language accepted by the NFA is exactly the language expressed by the RE.

This is done by systematically using the constructs for the basic primitives and operators of an RE and building ups using Thompson's Construction process. The steps are:

Step-1, Parse the given RE into its constituent sub-expressions.

Step-2, Construct NFA-s for each of the basic symbols in the given RE.

For ε, construct the NFA as in Fig-1. Here i is a new start state and f is a new accepting state. It is clear that this NFA recognizes the RE {ε}.

For every a in the alphabet S, construct the NFA as shown in Fig-2. Here again i is a new start state and f is a new accepting state. It is clear that this NFA recognizes the RE {a}. If a occurs several times in the given RE, a separate NFA is constructed for each occurrence of a.

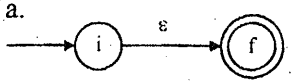


Fig 1. NFA for e

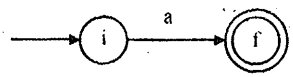


Fig 2. NFA for a character of the alphabet

Step-3, Combine the intermediate NFA-s inductively until the NFA for the entire expression is obtained. Each intermediate NFA produced during the course of construction corresponds to a sub-expression of the given RE and has several important properties – it has exactly one final state, no edge enters the start state and no edge leaves the final state.

Suppose N(s) and N(t) are the NFA-s for regular expressions s and t, respectively. The NFA-s for regular expression s|t, st and s*, the composite NFA-s N(s|t), N(st) and N(s*) are constructed as shown in

Fig- 3, Fig-4 and Fig-5, respectively:

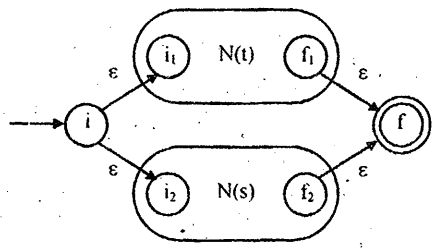


Fig: 3

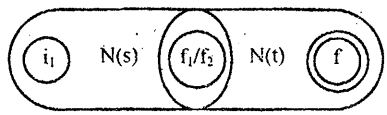


Fig: 4

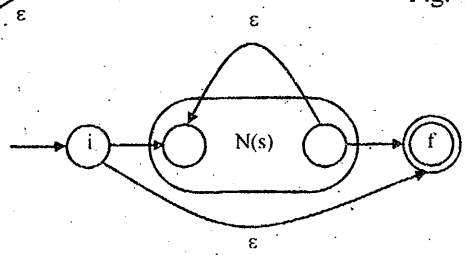


Fig.:5

d) LEX: ***Refer to Question no. 4(a) of Long Answer Type Questions.***

YAAC: ***Refer to Question no. 4(b) of Long Answer Type Questions.***

PARSING AND CONTEXT FREE GRAMMAR

Multiple Choice Type Questions

1. Which of the following expressions has no l-value? [WBUT 2006]
   a) a[i+1] b) a c) 3 d) *a

Answer: (c)

2. A grammar in which every production rule of the form X → a is known as:
   a) LL(0) b) LL(1) c) Context-free d) Regular [WBUT 2007]

Answer: (d)

3. Semantic analysis is applied to determine: [WBUT 2007]
   a) the argument types b) the type of intermediate results
   c) both (a) and (b) d) none of these

Answer: (c)

4. In a block structured language if procedure A invokes B with nested depths nA and nB, respectively, then: [WBUT 2007]
   a) nB - nA <= 1 b) nB - nA >= 1 c) nB - nA = 1 d) None of these

Answer: (a)

5. If all productions in a grammar G = (V,T,S,P) are of the form A→xB or A→x, A,B ∈ V and x ∈ T*, then it is called: [WBUT 2008]
   a) Context-sensitive grammar b) Non-linear grammar
   c) Right-linear grammar d) Left-linear grammar

Answer: (c)

6. An inherited attributes is the one whose initial value at a parse tree node is defined in terms of [WBUT 2009]
   a) attributes at the parent and / or siblings of that node
   b) attributes at children nodes only
   c) attributes at both children nodes and parent and / or siblings of that node
   d) none of these

Answer: (c)

7. The intersection of a regular language and a context free language is [WBUT 2009]
   a) always a regular language b) always a context free language
   c) always a context sensitive language d) none of these

Answer: (b)

8. The grammar E →E+E | E*E | α is **[WBUT 2010]**

a) ambiguous **b) unambiguous**

c) not given sufficient information **d) none of these**

Answer: (a)

9. Parse tree is generated in the phase of **[WBUT 2011]**

a) Syntax Analysis **b) Semantic Analysis**

c) Code Optimization **d) Intermediate Code Generation**

Answer: (a)

10. If the attributes of the parent node depends on its children, then its attributes are called **[WBUT 2012]**

a) TAC **b) synthesized** **c) inherited** **d) directed**

Answer: (c)

Short Answer Type Questions

1. What is 'handle'? Consider the grammar E→E +n|E *n|n. For a sentence n+n*n, write the handles in the right-sentential forms of the reduction.
What is predictive parsing? **[WBUT 2006]**

Answer:

A handle of a right sentential form gamma is a production A!b and a position in gamma where the string beta may be found and replaced by A to get the previous right-sentential form.

The right-most productions are shown below. The handles are underlined.

E→<u>E*n</u>→<u>E+n</u>*n→n+n*n

The handles are underlined.

A predictive parser is a recursive descent parser that does not require backtracking. Predictive parsing is possible only for the class of LL(k) grammars, which are the context-free grammars for which there exists some positive integer k that allows a recursive descent parser to decide which production to use by examining only the next k tokens of input. (The LL(k) grammars therefore exclude all ambiguous grammars, as well as all grammars that contain left recursion. Any context-free grammar can be transformed into an equivalent grammar that has no left recursion, but removal of left recursiondoes not always yield an LL(k) grammar.) A predictive parser runs in linear time.

A table-driven non-recursive predictive parser (for an LL(1) grammar) uses an explicit parsing stack and a parsing table M[A;a] (where A is a nonterminal and a is a terminal) where an entry M[A:a] is either a production rule or an error. A predictive parsing algorithm uses this table to carry out top-down parsing.

2. Prove that the following grammar is ambiguous: **[WBUT 2007]**

S→AB
A→aa|a
B→ab|b

Answer:

Clearly, the string "aab" has two productions:

$$(i) S \Rightarrow AB \Rightarrow aaB \Rightarrow aab$$

$$(ii) S \Rightarrow AB \Rightarrow aB \Rightarrow aab$$

Hence, the grammar is ambiguous.

3. Consider the following context-free grammar: **[WBUT 2008, 2009]**

S → SS+ | SS* | a

a) Show how the string aa+a* can be generated by this grammar.

b) Construct a parse tree for the given string.

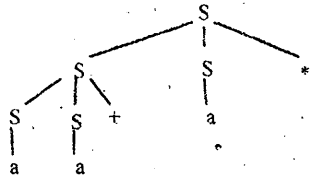
c) What language is generated by this grammar?

Answer:

a) The string aa+a* can be generated by the rightmost production:

$$S \rightarrow SS^* \rightarrow Sa^* \rightarrow SS+a^* \rightarrow Sa+a^* \rightarrow aa+a^*$$

b) The required parse tree is:



c) This grammar generates Binary Postfix Expressions involving a with + and * as the only operators.

4. Consider the following left-linear grammar: **[WBUT 2008]**

$$S \rightarrow Sab \mid Aa$$

$$A \rightarrow Abb \mid bb$$

Find out an equivalent right-linear grammar.

Answer:

This grammar generates the regular language (bb)+a(ab)+ where + means "one or more occurrences". An equivalent right-linear grammar is:

$$S \rightarrow bbS \mid bbaA$$

$$A \rightarrow abA \mid ab$$

5. What is a handle? **[WBUT 2008]**

Consider the grammar $E \rightarrow E+E \mid E*E \mid id$

Find the handles of the right sentential forms of reduction of the string id + id * id

Answer:

A handle of a right sentential form gamma is a production A-->b and a position in gamma where the string beta may be found and replaced by A to get the previous right-sentential form.

The given grammar is ambiguous. There are two right-most derivations for the string id + id * id. We give both the rightmost derivations, underlining the handles in each case.

$E \to E + E \to E + E * E \to E + E * id \to E + id * id \to id + id * id$

$E \to E * E \to E * id \to E + E * id \to E + id * id \to id + id * id$

6. Consider the following grammar G. Alternate the production so that it may free from backtracking.

Statement → if Expression then Statement else Statement

Statement → if Expression then Statement **[WBUT 2012]**

Answer:

Statement → if Expression then Statement Trailer

Trailer → else Statement

Trailer → ε

Long Answer Type Questions

1. Construct a predictive parsing table for the grammar: **[WBUT 2010]**

S → iEtSS' | a

S' → eS | ε

E → b

Here S is star symbol and S' are non-terminals and i, t, a, e, b are terminals. Explain the steps in brief.

Answer:

From rules $S \to iEtSS' \mid a$, we get $FIRST(S) = \{i, a\}$.

From rules $S' \to eS; \mid \varepsilon$, we get $FIRST(S') = \{e, \varepsilon\}$.

From rule $E \to b$, we get $FIRST(E) = \{b\}$.

Using the FIRST sets and the rules, we get:

$FOLLOW(S) = \{\$\} \cup FIRST(S') = \{e, \$\}$.

$FOLLOW(S') = \{e, \$\}$.

$FOLLOW(E) = \{t\}$.

Suppose the predictive parsing table is the 2-dimensional array $P[A, a]$, where A is a non-terminal and a is a terminal.

Since $i \in FIRST(S)$, $P[S, i] = S \to iEtSS'$.

Since $a \in FIRST(S)$, $P[S, a] = S \to a$.

Since $e \in FIRST(S')$, $P[S',e] = S' \rightarrow eS$. Also, since $\varepsilon \in FIRST(S')$, $P[S',e] = S' \rightarrow \varepsilon$ because $e \in FOLLOW(S')$. Hence, we have multiple entries for $M[S',e]$. Since $\$ \in FOLLOW(S')$ and obviously $\varepsilon \in FIRST(\varepsilon)$, we have $P[S',\$] = S' \rightarrow \varepsilon$.

Since $b \in FIRST(E)$, $P[E,b] = E \rightarrow b$.

No other rule is left to be scanned. So the predictive parsing table is:

Non-terminal	Input Symbol					
	i	t	e	a	b	$
S	$S \rightarrow iEtSS'$			$S \rightarrow a$		
S'			$S' \rightarrow eS$ $S' \rightarrow \varepsilon$			$S \rightarrow \varepsilon$
E					E → b	

2. a) Define LL(1) grammar. Consider the following grammar:

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

Test whether the grammar is LL(1) or not and construct a predictive parsing table for it. **[WBUT 2012]**

Answer:

FIRST (A) = $\{\varepsilon\}$, FIRST (B) = $\{\varepsilon\}$

FIRST (S) = $\{a, b\}$

FOLLOW (A) = $\{a, b\}$, FOLLOW (B) = $\{a, b\}$, FOLLOW (S) = $\{\$\}$

The predictive parsing table is:

	a	b	$
S	S → AaAb	S → BbBa	
A	A → ε	A → ε	
B	B → ε	B → ε	

Since there are no conflicts, the grammar is LL(1)

b) Consider the following Context Free Grammar (CFG) G and reduce the grammar by removing all unit productions. Show each step of removal **[WBUT 2012]**

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow C|b$

$C \rightarrow D$

$D \rightarrow E$

$E \rightarrow a$

Answer:

State : S → AB
A → a
B → C | b
C → D
D → E
E → a

Step-1: S → AB
A → a
B → C | b
C → D
D → a
E → a

Step-2: S → AB
A → a
B → C | b
C → a
D → a
E → a

Step-3: S → AB
A → a
B → a | b
C → a
D → a
E → a

c) Consider the following grammar G. Show that the grammar is ambiguous by constructing two different leftmost derivations for the sentence '*abab*'.

$S \rightarrow aSbS|bSaS|\varepsilon$ **[WBUT 2012]**

Answer:

The two possible left-most derivations for 'abab' are:

(i) $S \xrightarrow{S \rightarrow aSbS} a\underline{S}bS \xrightarrow{S \rightarrow \varepsilon} ab\underline{S} \xrightarrow{S \rightarrow aSbS} aba\underline{S}bS \xrightarrow{S \rightarrow \varepsilon} ababS \xrightarrow{S \rightarrow \varepsilon} abab$

(ii) $S \xrightarrow{S \to aSbS} aSbS \xrightarrow{S \to bSaS} abSaSbS \xrightarrow{S \to \varepsilon} abaSbS \xrightarrow{S \to \varepsilon} ababS \xrightarrow{S \to \varepsilon} abab$

3. Write short note on the following: **[WBUT 2008]**

a) left factoring

b) context-free grammar

c) Inherited attributes

Answer:

a) left factoring:

Left factoring is an important step required to transform a given grammar to one that is suitable for building an LL (i.e., top-down) parser. This step is carried out after removing all left recursion. Even if a context-free grammar is unambiguous and non-left-recursion, it still may not be LL(1).

The problem is that there is only look-ahead buffer. The parser generated from such grammar is not efficient as it requires backtracking. To avoid this problem we left factor the grammar.

To left factor a grammar, we collect all productions that have the same left hand side and begin with the same symbols on the right hand side. We combine the common strings into a single production and then append a new nonterminal symbol to the end of this new production. Finally, we create a new set of productions using this new nonterminal for each of the suffixes to the common production.

Suppose we have production rules:

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots\ldots \mid \alpha\beta_n$$

After left factoring the above grammar is transformed into

$$A \to \alpha A_1 A_1 \to \beta_1 \mid \beta_2 \mid \ldots\ldots \mid \beta_n$$

The above grammar is correct and is free from conflicts.

b) context-free grammar:

The formalism of Context-free grammars was developed in the mid-1950s by Noam Chomsky who used in the study of human languages (i.e., Natural Languages). Later, Context-free Grammars found an extremely important application in the specification and compilation of programming languages.

A grammar for a programming language is the starting point in the design of compilers and interpreters for programming languages. Most compilers and interpreters contain a component called a parser that extracts the meaning of a program prior to generating the compiled code or performing the interpreted execution. A number of methodologies facilitate the construction of a parser once a Context-free grammar is available. Some tools even automatically generate the parser from the grammar.

In terms of generative power, Context-free Grammars, or CFG-s, are more expressive than Regular Grammars (or equivalent formalisms like Regular Expression and Finite Automata).

Formally, a Context-free Grammar is defined as a 4-tuple

$G = < V_n , V_t , P , S >$, where:

- V_t is a finite set of terminals. The set of terminals constitute the alphabet S for the language as well as e.
- V_n is a finite set of non-terminals. The non-terminals constitute a special alphabet that is disjoint from the terminals.
- P is a finite set of production rules of the type $V_n \rightarrow (V_n \cup V_t)^*$.
- S is an element of V_n, the distinguished starting non-terminal, often call the Start Symbol.

c) Inherited attributes:

Inherited attributes are attributes that are passed to a rule, as opposed to synthesized attributes, which are returned from a rule. These attributes are computed from the values of the attributes of both the siblings and the children nodes of Annotated Parse Trees. Inherited attributes are convenient for expressing the dependence of a programming language construct on the context in which it appears. For example, we can use an inherited attribute to keep track of whether an identifier appears on the left or the right side of an assignment in order to decide whether the address or the value of the identifier is needed. Although it is always possible to rewrite a syntax directed definition to use only synthesized attributes, it is often more natural to use definitions with inherited attributes. For example, if in a grammar, the production rule D→TL is associated with the semantic rule L:in = T:type (this is a case of variable declaration in C/C++ where L lists a list of identifiers), the types of the identifiers are inherited from the type declaration encoded in the non-terminal T. The declared type is percolated down the syntax tree headed by L down to each identifier.

OPERATOR PRECEDENCE PARSING

Multiple Choice Type Questions

1. Which data structure is mainly used during shift-reduce parsing? **[WBUT 2010, 2012]**

a) Pointers b) Arrays c) Stacks d) Queues

Answer: (c)

Short Answer Type Questions

1. Define an operator grammar. **[WBUT 2006]**

OR,

What is an operator grammar? Give an example. **[WBUT 2009]**

Answer:

A grammar is called an Operator Grammar if there is no production and no right hand side of any production has two adjacent non-terminals.

For example, the grammar:

S→x|y|z|S + S|S - S|S * S|S / S|(S)

is an operator grammar while the grammar:

S→ASA

A→a|b

is not an operator grammar since the rule A → ASA has two non-terminals side by side.

2. Describe two situations where operator precedence parser can discover errors. **[WBUT 2007]**

Answer:

Two situations where operator precedence parser can discover errors are:

i. When no precedence relation holds between the terminal on the top of the stack and the current input. Hence shift will fail.

ii. When a handle has been found (i.e., tos > current input) but there is no production with this handle as the right hand side.

Long Answer Type Questions

1. a) What is left-recursion? Illustrate with suitable example. Consider the following grammar G. Find out the left recursion and remove it: **[WBUT 2012]**

$S \rightarrow Bb|a$

$B \rightarrow Bc|Sd|e$

Answer:

1st Part:

Left recursion: when one or more productions can be reached from themselves with no tokens consumed in-between. *Left recursion* is a particular form of recursion that cannot be directly handled by the simple LL (1) parsing algorithm. Left recursion just refers to any recursive non-terminal that, when it produces a sentential form containing itself, that new copy of itself appears on the left of the production rule.

2nd Part:

Step-1: Remove immediate left recursion in $B \rightarrow Bc$

$B \rightarrow SdB' \mid eB'$

$B' \rightarrow cB' \mid \varepsilon$

Step-2:

(i) $S \rightarrow SdB'b \mid cB'b \mid a$

$B' \rightarrow SdB' \mid eB'$

$B' \rightarrow cB' \mid \varepsilon$

(ii) After removing immediate left recursion is

$S \rightarrow SdB'b$

$S \rightarrow eB'bS' \mid aS'$

$S' \rightarrow dB'bS' \mid \varepsilon$

$B \rightarrow SdB' \mid eB'$

$B' \rightarrow cB' \mid \varepsilon$

The grammar is now free from left recursion.

b) What is Operator Precedence Parsing? Discuss about the advantage and disadvantage of Operator Precedence Parsing. Consider the following grammar:

$E \rightarrow TA$

$A \rightarrow +TA \mid \varepsilon$

$T \rightarrow FB$

$B \rightarrow *FB \mid \varepsilon$

$F \rightarrow id$

Test whether this grammar is Operator Precedence Grammar or not and show how the string $w = id + id * id + id$ will be processed by this grammar. [WBUT 2012]

Answer:

1st Part: An operator-precedence parser is a simple shift-reduce parser capable of parsing a subset of LR (1) grammars. More precisely, the operator precedence parser can parse all LR (1) grammars where two consecutive non-terminals never appear in the right-hand side of any rule.

The technique of parsing through this parser is Operator Precedence Parsing.

2nd part:

Advantage:

- Because of its simplicity numerous compiler using operator precedence parsing technique.
- Can have been built for the entire language.

Disadvantage:

- Hard to handle token like unary minus
- Worse, since relation between a grammar for the language being parsed and the operator precedence parser itself is week. That is cannot always be sure the parser accepts exactly the desired language.

Last Part: The given grammar is NOT an operator precedence grammar. It is not even an operator grammar because:

(i) There are several productions (e.g., T → FB) where two non-terminals occur side by side in the right hand side.

(ii) There are several ε productions.

The string $w = id + id * id + id$ CANNOT be parsed by the given grammar since the terminal '=' is not in the terminal-set of the grammar.

TOP-DOWN PARSING

Multiple Choice Type Questions

1. Given a grammar G = (fEg;fid;+g;P;E); where P is given by E ! E +E;E !id, then FOLLOW(E) will contain: [WBUT 2006]
   a) {$} b) f+g c) {$,+} d) {$, id,+}

Answer: (c)

2. A dangling reference is a [WBUT 2006, 2008]
   a) pointer pointing to storage which is freed
   b) pointer pointing to nothing
   c) pointer pointing to storage which is still in use
   d) pointer pointing to uninitialized storage

Answer: (a)

3. If x is a terminal, then FIRST(x) is: [WBUT 2007, 2012]
   a) ε b) {x} c) x* d) None of these

Answer: (b)

4. Which of the following is not a loop optimization? [WBUT 2008, 2009]
   a) Loop unrolling b) Loop jamming
   c) Loop heading d) Induction variable elimination

Answer: (c)

5. A top down parser generates [WBUT 2010, 2012]
   a) leftmost – derivation b) rightmost – derivation
   c) leftmost derivation in reverse d) rightmost derivation in reverse

Answer: (a)

6. $FIRST(\alpha\beta)$ is [WBUT 2011]
   a) $FIRST(\alpha)$
   b) $FIRST(\alpha) \cup FIRST(\beta)$
   c) $FIRST(\alpha) \cup FIRST(\beta)$ *if* $FIRST(\alpha)$ contains ε else $FIRST(\alpha)$
   d) none of these

Answer: (c)

7. A given grammar is not LL(1) if the parsing table of a grammar may contain
   a) any blank filed b) any e-entry [WBUT 2011]
   c) duplicate entry of same production d) more than one production rule

Answer: (d)

8. First pos of a (dot) node with leaves c1 and c2 is **[WBUT 2011]**

a) $firstpos(c1) \cup firstpos(c2)$

b) $firstpos(c1) \cap firstpos(c2)$

c) $if\,(nullable(c1))\,firstpos(c1) \cup firstpos(c2)\,else\,firstpos(c1)$

d) $if\,(nullable(c2))\,firstpos(c1) \cup firstpos(c2)\,else\,firstpos(c1)$

Answer: (c)

9. Left factoring guarantees **[WBUT 2011]**

a) not occurring of backtracking
b) cycle free parse tree
c) error free target code
d) correct LL(1) parsing table

Answer: (a)

Short Answer Type Questions

1. Eliminate left recursion from the following grammar: **[WBUT 2007]**

E →E +T|T
T →TF|F
F →F * |a|b

Answer:

After eliminating the left recursion, the resulting grammar (with three additional non-terminals E' and T') have the rules:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow FT' \mid \varepsilon$$
$$F \rightarrow aF' \mid bF'$$
$$F' \rightarrow *F' \mid \varepsilon$$

2. What is recursive descent parsing? Describe the drawbacks of recursive descent parsing for generating the string abc from the grammar: **[WBUT 2011]**

$S \rightarrow aBx$

$B \rightarrow bc \mid b$

Answer:

Recursive descent parsing is a simple way to construct a top-down parser by regarding each production rule as a function, where:

- The name of the function is the non-terminal on the left hand side of the rule,
- Each instance of a non-terminal on the right hand side is a call to the corresponding function.
- Each instance of a terminal on the right hand side tells us to match it with the input symbol and thereby 'consume' it.
- Parsing happens as the execution of the function corresponding to the start symbol.

In the given grammar, we have problem is constructing the function for the non-terminal B because both the B productions start with the terminal b.

Long Answer Type Questions

1. Which parser is used for the implementation of recursive descent parsing? Draw a model diagram for that parser. Construct the parsing table for the grammar: **[WBUT 2006]**

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid id$$

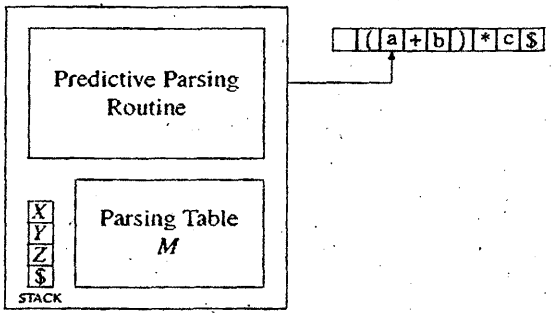
With the help of parsing table and other components how the string id + id * id is parsed?

Answer:

1st Part:

A commonly used non-recursive (in program code) implementation of a recursive descent parser is the Predictive Parser.

If we recall our lessons in Formal Language Theory, we may perhaps remember that the automata for a CFG is a Push-Down Automata (PDA). It may be worthwhile mentioning here that an LL parser is in effect an implementation of PDA. Now, a PDA was defined as working with a "push-down store" or stack. Where is this stack in the recursive descent parser? The stack in recursive descent parser is the same stack that programs use while using subroutine calls. As we have seen, a trivial implementation of a recursive parser for a given grammar (without left-recursion, say) may require too many backtracking. However, the backtracking may be eliminated by using the concepts of left-recursion removal, FIRST, FOLLOW, etc.



Predictive Parser in Action

2nd Part:

Clearly, FIRST(F) = {(, id}.

Also, FIRST(T′) = {*, ε} and FIRST(E′) = {+, ε}.

Now FIRST(E) contains everything in FIRST(T) and FIRST(T) contains everything in FIRST(F).

From this we get:

FIRST(E) = FIRST(T) = FIRST(F) = {(, id}.

Next, we can start with FOLLOW(E) = {), $}.

FOLLOW(T) = FIRST(E′) − {ε} = {+}.

FOLLOW(E′) = FOLLOW(E) = {), $}.

FOLLOW(T′) = FOLLOW(T) = {+} because FOLLOW(T) contains all non-ε symbols in FIRST(E′).

Also, since ε ∈ FIRST(E′), we get FOLLOW(T) = FOLLOW(T′) = {+,), $}.

FOLLOW(F) = FIRST(T′) − {ε} = {*} because FOLLOW(F) contains all non-ε symbols in FIRST(T′).

Also, since ε ∈ FIRST(T′), we get FOLLOW(F) = {+, *,), $}.

The Predictive Parsing Table is:

Non-Terminal	INPUT SYMBOL					
	+	*	(	)	id	$
E			E → TE′		E → TE′	
E′	E′ → + TE′			E′ → ε		E′ → ε
T			T → FT′		T → FT′	
T′	T′ → ε	T′ → *FT′		T′ → ε		T′ → ε
F			F → (E)		F → id	

The Predictive Parser's action for input id + id *id$ is as shown below:

Stack	Input	Output
$E	id + id * id$	
$E′T	id + id * id$	$E \rightarrow TE'$
$E′T′F	id + id * id$	$T \rightarrow FT'$
$E′Tid	id + id * id$	$F \rightarrow id$
$E′T′	+ id * id$	
$E′	+ id * id$	$T' \rightarrow \varepsilon$
$E′T+	+ id * id$	$E' \rightarrow +TE'$
$E′T	id * id$	
$E′T′F	id * id$	$T \rightarrow FT'$
$E′T′id	id * id$	$F \rightarrow id$
$E′T′	* id$	
$E′T′F*	* id$	$T' \rightarrow *FT'$
$E′T′F	id$	
$E′T′id	id$	$E \rightarrow id$
$E′T′	$	
$E′	$	$T' \rightarrow \varepsilon$
$	$	$E' \rightarrow \varepsilon$

2. Design an LL(1) parsing table for the following grammar: **[WBUT 2008]**

S→aAcd|BcAe

A→b|ε

B→C f |d

C→f e

With the help of the parsing table, show how the string fefcbe isparsed.

Answer:

Clearly, the given grammar is free from any left-recursion — there is not even any immediate left-recursion. So, we begin by computing the FIRST and FOLLOW sets for the non-terminals:

FIRST(S) = {a,d, f }, FIRST(A) = {b, ε}, FIRST(B) = {d, f } and FIRST(C) = {f }.

FOLLOW(S) = {$}, FOLLOW(A) = {c,e}, FOLLOW(B) = {}, and FOLLOW(C) = {f }.

The parsing table is therefore:

	a	b	c	d	e	f	$
S	$S \rightarrow aAcd$			$S \rightarrow BcAe$		$S \rightarrow BcAe$	
A		$A \rightarrow b$	$A \rightarrow \varepsilon$		$A \rightarrow \varepsilon$		
B				$B \rightarrow d$		$B \rightarrow Cf$	
C						$C \rightarrow fe$	

The parse for the string fefcbe proceeds as follows:

S→aAcd|BcAe

A→b|ε

B→C f |d

C→f e

Stack	Input	Output/ Action
$\$S$	fefcbe$	START
$\$eAcB$	fefcbe$	$S \rightarrow BcAe$
$\$eAcfC$	fefcbe$	$B \rightarrow Cf$
$\$eAcfef$	fefcbe$	$C \rightarrow fe$
$\$eAcfe$	efcbe$	POP
$\$eAcf$	fcbe$	POP
$\$eAc$	cbe$	POP
$\$eA$	be$	POP
$\$eb$	be$	$A \rightarrow b$
$\$e$	e$	POP
$	$	SUCCESS

3. a) Consider the following Grammar:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid id$$

i) Obtain the FIRST and FOLLOW sets for the above grammar.

ii) Construct the Predictive Parsing table for the above grammar.

b) Explain the Predictive Parser's action by describing the moves it would make on an input id + id *id$. **[WBUT 2008]**

Answer:

Refer to Question No. 1 of Long Answer Type Questions.

4. a) Describe with a block diagram the parsing technique of LL(1) parser.

b) Parse the string abba using LL(1) parser where the parsing table is given below:

	A	b	$
S	S → aBa		
B	B → e	B → bB	

c) Check whether the following grammar is LL(1) or not.

$S \rightarrow iCt\,SE \mid a$

$E \rightarrow eS \mid \varepsilon$ **[WBUT 2011]**

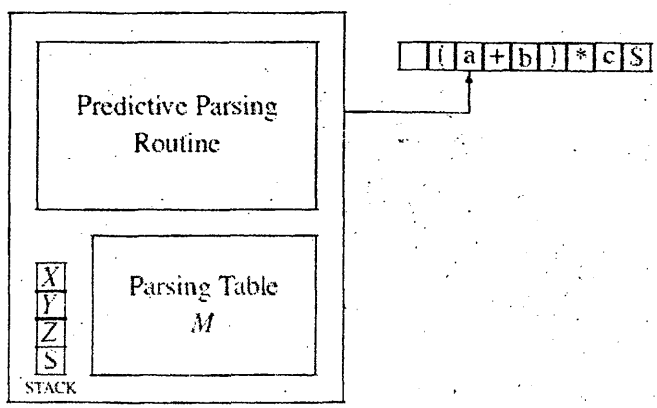
$C \rightarrow b$

Answer:

a) LL(1) parsers can be of various types. However from the question it appears that a description of a "Non-recursive Predictive Parser" is to be given.

A Non-Recursive Predictive Parser is an LL parser implementation based on a table driven algorithm that uses an explicit stack.

The block diagram of a typical predictive parser is shown below:



The parsing table is of type $M[A, a]$ which indicates which production to use if the top of the stack is a non-terminal A and the current token is equal to a. If there is a valid (i.e., nor "error") entry $M[A, a]$, we pop A from the stack and push all the right-hand side symbols of the production stored in the entry $M[A, a]$. If that happens to be $A \rightarrow XYZ$, we push symbols into the stack in the reverse order, i.e., we push Z first, followed by Y and X. We use

a special symbol $ to denote the end of file.

Assuming S to be the start symbol, the non-recursive predictive parsing algorithm

is as follows:

```
push(S);
a = CurrentInput();
do {
X = pop();
if (X is a terminal or '$') {
if (X == a) {
ConsumeInput();
a = CurrentInput();
}
else Error();
else if (M[X,a] == "X ? Y1 Y2 ... Yk") {
push(Yk);
...
push(Y1);
}
else Error();
} while X != '$';
```

b) The parsing action is summarized in the table below:

Stack	Input	Output
$S	abba$	
$aBa	abba$	s → aBa
$aB	bba$	
$aBb	bba$	B → bB
$aB	ba$	
$aBb	ba$	B → bB
$aB	a$	
$a	a$	B → e
$	$	

c) From grammar rules and properties of FIRST and FOLLOW we get:

FIRST(S) = {i,a}.

FIRST(E) = {e,ε}.FIRST(C)={b}.

FOLLOW(S) = {$} ∪ FIRST(E) = {e,$}.

FOLLOW(E) = {e,$}.

FOLLOW(C) = {t}.

The predictive parsing table then becomes:

Non-terminal	Input Symbol					
	I	t	e	a	b	$
S	S → iCtSE			S → a		
E			E → eS E → e			S → e
C					C → b	

Clearly, there is a duplicate entry in $P[E,e]$ contains multiple entries and hence the grammar is not LL(1).

BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY

Multiple Choice Type Questions

1. YACC builds up [WBUT 2006, 2007, 2010, 2011]
   a) SLR parsing table b) canonical LR parsing table
   c) LALR parsing table d) none of these

Answer: (c)

2. If a grammar is LALR (1) then it is necessarily: [WBUT 2006, 2007, 2008, 2010]
   a) SLR(1) b) LR(1) c) LL(1) d) none of these

Answer: (d)

3. Which data structure is mainly used during shift-reduce parsing? [WBUT 2007, 2008]
   a) Pointers b) Arrays c) Stacks d) Queues

Answer: (c)

4. If I is a set of valid items for a viable prefix γ, then GOTO (I , X) is a set of items that are valid for the viable prefix: [WBUT 2009]
   a) δ b) γ c) Prefix of γ d) None of these

Answer: (a)

5. Shift-reduce parsers are [WBUT 2009]
   a) Top-down parsers b) Bottom-up parsers
   c) May be top-down or bottom-up parsers d) None of these

Answer: (b)

Short Answer Type Questions

1. Given a grammar: [WBUT 2007]
E →E +T|T
T →T *F|F
F →id
Which is a set of valid items for a viable prefix E+.

Answer:

Suppose the viable prefix E+ leads to the state J from state I0. The last transition in the DFA of states is +. So, J is obtained from GoTo(I,+), where I contains the item E →E.+T.

So, the items in J, which are exactly the valid items for the viable prefix E+ (by Theorem on LR parsing), are:
{E →E +.T,T →.T *F,T →.F,F →.id}.

Long Answer Type Questions

1. Given a grammar G=({E,T,F},{id,+,*, (,)},P,E) , where, P is given by:
E →E +T|T,
T →T *F|F,
F→(E)|id
Construct the SLR(1) parsing table for G. **[WBUT 2006, 2009]**

Answer:

The augmented grammar is:

$\mathbf{E' \to E}$

$\mathbf{E \to E+T}$

$\mathbf{E \to T}$

$\mathbf{T \to T*F}$

$\mathbf{T \to F}$

$\mathbf{F \to (E)}$

$\mathbf{F \to id}$

Now, $s_0 = \text{closure}(\mathbf{E' \to .E.})$

We start with $s_0 = \{\mathbf{E' \to .E}\}$. Silence the 'dot' is in front of **E**, a non-terminal, items for all **E** productions with the dot at the beginning, will be appended with the s_0. So, s_0 becomes:

$s_0 = \{\mathbf{E' \to .E, E \to .E+T, E \to .T}\}$.

Again, the dot is front of a (new) non-terminal T and all T productions with the dot at the beginning, will be appended to s_0. So, s_0 becomes:

$s_0 = \{\mathbf{E' \to .E, E \to .E+T,}\}$.

$E \to .T, T \to .T * F, T \to .F.$

Again, the dot is front of a (new) non-terminal F and all F productions with the dot at the beginning, will be appended to s_0. So, finally, s_0 becomes:

$s_0 = \{\mathbf{E' \to .E, E \to .E+T,}\}$

$E \to .T, T \to .T * F, T \to .F, F \to .(E), F \to .id$

It is possible to have transitions from s_0 on **E, T, F, id** and (,because, these are the grammar symbols in the items in s_0, where the dot is just before it.

Consider:

$s_1 = \text{goto}(s_0, \mathbf{E})$.

This is simple. Simply let the dot cross **E** wherever the dot is in front of **E**.
This gives:

$s_1 = \{\mathbf{E}' \to \mathbf{E}., \mathbf{E} \to \mathbf{E}.+\mathbf{T}\}$.

Note that in both the items, the dot is in front of terminals. So, the "closure" operation will not add further items.
Similarly,

$s_2 = goto(s_0, \mathbf{T}) = \{\mathbf{E} \to \mathbf{T}., \mathbf{T} \to \mathbf{T}.*\mathbf{F}\}$

$s_3 = goto(s_0, \mathbf{F}) = \{\mathbf{T} \to \mathbf{F}.\}$

$s_5 = goto(s_0, \mathbf{id}) = \{\mathbf{E} \to \mathbf{id}.\}$

Much more interesting is $s_4 = goto(s_0, (\ )$

This start with:

$s_4 = \{\mathbf{F} \to (.\mathbf{E})\}$

and then with repeated application of closure, end with:

$s_4 = \{\mathbf{F} \to (.\mathbf{E}), \mathbf{E} \to .\mathbf{E}+\mathbf{T}, \mathbf{E} \to .\mathbf{T},\}$

$T \to .T*F, T \to .F, F \to .(E), F \to .id.$

We can proceed similarly to get the following:

$s_6 = goto(s_1, +) = \{\mathbf{E} \to \mathbf{E}+.\mathbf{T}, \mathbf{T} \to \mathbf{T}*\mathbf{F},\}$

$T \to .F, F \to .(E), F \to .id.$

$s_7 = goto(s_2, *) = \{\mathbf{T} \to \mathbf{T}*.\mathbf{F}, \mathbf{F} \to .(\mathbf{F}), \mathbf{F} \to .\mathbf{id}\}$.

$s_8 = goto(s_4, \mathbf{E}) = \{\mathbf{F} \to (\mathbf{E}.), \mathbf{E} \to \mathbf{E}.+\mathbf{T}\}$.

$s_9 = goto(s_6, \mathbf{T}) = \{\mathbf{E} \to \mathbf{E}+\mathbf{T}., \mathbf{T} \to \mathbf{T}.*\mathbf{F}\}$.

$s_{10} = goto(s_7, \mathbf{F}) = \{\mathbf{T} \to \mathbf{T}*\mathbf{F}.\}$.

$s_{11} = goto(s_8, \ )) = \{\mathbf{F} \to (\mathbf{E}).\}$.

Additionally:

$goto(s_4, \mathbf{T}) = s_2$

$goto(s_4, \mathbf{F}) = s_3$.

$goto(s_4, (\ ) = s_4$.

$goto(s_4, \mathbf{id}) = s_5$.

$goto(s_6, \mathbf{F}) = s_3$.

$goto(s_6,(\ ) = s_4.$

$goto(s_6,\mathbf{id}) = s_5.$

$goto(s_7,) = s_4.$

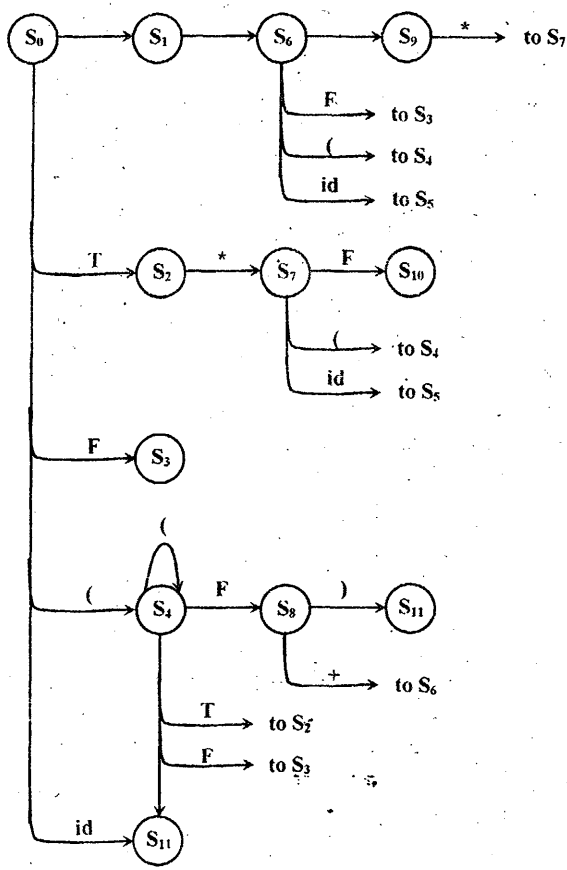
$goto(s_7,\mathbf{id}) = s_5.$

$goto(s_8,+) = s_6.$

$goto(s_9,*) = s_7.$

You can clearly visualize the DFA and are encouraged to draw a diagram for the same (see Fig).

We have completed **Step-1** of parser table construction process.



In **Step-2,** we can use the transitions (i.e. goto-s) numbered 4-7, 11, 14, 15 and 17-22 to create the **shift-j** entires of the **ACTION** part of the parsing table. For example:

- Using goto $(s_4(\ ) = s_4$, we get **ACTION** $[4',(\ '] = s^4.$

- Similarly, from goto $(s^4, \mathbf{id}) = s_5$ and goto $(s^6, \mathbf{id}) = s_5$, we get **ACTION** $[6', \mathbf{id}'] = s^5$. as also **ACTION** $[7', \mathbf{id}'] = s^5$.
- Again, since goto $(s_8\ )) = s_{11}$, we get **ACTION** $[8',\ )'] = s^{11}$.

Use the other transitions to complete the **shift** entires of the parsing table. For **Step-3,** we need to calculate the **FOLLOW** sets for non terminals **E, T** and **F**. We have already seen how to calculate the **FOLLOW** sets. So we know **FOLLOW (E)** = $\{\$, +,)\}$. Now consider the item $\mathbf{E} \rightarrow \mathbf{T}$. in s_2. The item comes from rule-2. Because of this, the entires **ACTION**$[2,\$]$, **ACTION** $[2,+]$and **ACTION** $[2,\ )]$are all **reduce by** $\mathbf{E} \rightarrow \mathbf{T}$ or r_2.

By similar logic, since **FOLLOW (F)** $=\{+,*,),\$\}$ and $\mathbf{F} \rightarrow \mathbf{id}$. (which comes from rule-6) is in s_5. we get **ACTION** $[2,\$]$, **ACTION** $[2,+]$, **ACTION** $[2,*)]$ and **ACTION** $[2,\ )]$ are all **reduce by** $\mathbf{F} \rightarrow \mathbf{id}$ or r_6.

We can use the other appropriate items to complete the **reduce** entires of the parsing table.

Using **Step-4,** we get **ACTION** $[1,\$]$**=acc.**

Using **Step-5** and the transitions numbered 1-3, 8-10, 12, 13 and 16, we get all the **GOTO** entires of the parsing table. For example, **GOTO[4,T]=2,** since $goto(s_4, \mathbf{T}) = s_2$.

By **Step-6,** All non-filled entires are error entires.

By **Step-7,** the initial state of the parser is s_0.

The Parsing Table is as below:

State	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

2. Construct SLR parsing table for the given grammar. [WBUT 2007]

$E \rightarrow E + T \mid T$

$T \rightarrow TF \mid F$

$F \rightarrow F^* \mid a \mid b$

Answer:

Using the standard method of obtaining the canonical collection of LR(0) items for the augmented, we get:

$$\{E' \rightarrow .E, E \rightarrow .E + T, E \rightarrow .T, T \rightarrow .TF, T \rightarrow .F, F \rightarrow .F*, F \rightarrow .a, F \rightarrow .b\} = I_0$$

$goto(I_0, E) = \{E' \rightarrow E., E \rightarrow E. + T\} = I_1$

$goto(I_0, T) = \{E \rightarrow T., T \rightarrow T.F, F \rightarrow .F*, F \rightarrow .a, F \rightarrow .b\} = I_2$

$goto(I_0, F) = \{T \rightarrow F., F \rightarrow F.*\} = I_3$

$goto(I_0, a) = \{F \rightarrow a.\} = I_4$

$goto(I_0, b) = \{F \rightarrow b.\} = I_5$

$goto(I_1, +) = \{E \rightarrow E + .T, T \rightarrow .TF, T \rightarrow .F, F \rightarrow .F*, F \rightarrow .a, F \rightarrow .b\} = I_6$

$goto(I_2, F) = \{T \rightarrow TF., F \rightarrow F.*\} = I_7$

$goto(I_2, a) = \{F \rightarrow a.\} = I_4$

$goto(I_2, b) = \{F \rightarrow b.\} = I_5$

$goto(I_3, *) = \{F \rightarrow F*.\} = I_8$

$goto(I_6, T) = \{E \rightarrow E + T., T \rightarrow T.F, F \rightarrow .F*, F \rightarrow .a, F \rightarrow .b\} = I_9$

$goto(I_6, F) = I_3$

$goto(I_6, a) = I_4$

$goto(I_6, b) = I_5$

$goto(I_7, *) = I_8$

$goto(I_9, F) = I_7$

$goto(I_9, a) = I_4$

$goto(I_9, b) = I_5$

Now, $FOLLOW(E) = \{+, \$\}$, $FOLLOW(T) = \{+, a, b, \$\}$ and $FOLLOW(F) = \{+, *, a, b, \$\}$.

State	ACTION					GOTO		
	+	*	a	b	$	E	T	F
0			s4	s5		1	2	3
1	s6				acc			
2	r2		s4	s5	r2			7
3	r4	s8	r4	r4	r4			
4	r6	r6	r6	r6	r6			
5	r7	r7	r7	r7	r7			
6			s4	s5			9	3
7	r3	s8	r3	r3	r3			
8	r5	r5	r5	r5	r5			
9	r1		s4	s5	r1			7

3. What is shift reduce parsing? What is handle? Give examples of handle. Show an illustration of shift reduce parsing for a suitable grammar and for Each reduction indicate the corresponding handle. [WBUT 2007]

Answer:

Shift-reduce parsing is a general style of bottom-up syntax analysis. This technique attempts to construct a parse tree for the input string beginning at the leaves and working up to the root. One can think of this as one of "reducing" a string w to the start symbol. At each reduction step, a particular substring the right side of a production is replaced by the symbol on the left of that production and if the substring chosen is correct at each step, a rightmost derivation is traced out in reverse.

A handle of a right sentential form gamma is a production A →b and a position in gamma where the string beta may be found and replaced by A to get the previous right-sentential form.

Consider the grammar:

$E \Rightarrow E+E|E*E|a|b|c$

One rightmost derivation for a+b*c is (handles underlined):

$E \Rightarrow E+E \Rightarrow E+E*E \Rightarrow E+E*c \Rightarrow E+b*c \Rightarrow a+b*c$

The reductions that happen in order are obtained from the last derivation by replacing the underlined handle by the non-terminal on the left (in this case, E).

4. Construct SLR parsing table for following grammar: [WBUT 2009]

S → AS | b

A → SA | a

Answer:

Part A

- ***First and Follow Set:***

  FIRST (S) → {a, b}

  FIRST (A) → {a, b}

  FOLLOW (S) → {a, b, $}

  FOLLOW (A) → {a, b}

- ***Reductions:***

  r1= S→AS

  r2= S→b

  r3= A→SA

  r4= A→a

- ***Augmented grammar:***

  S'→S.

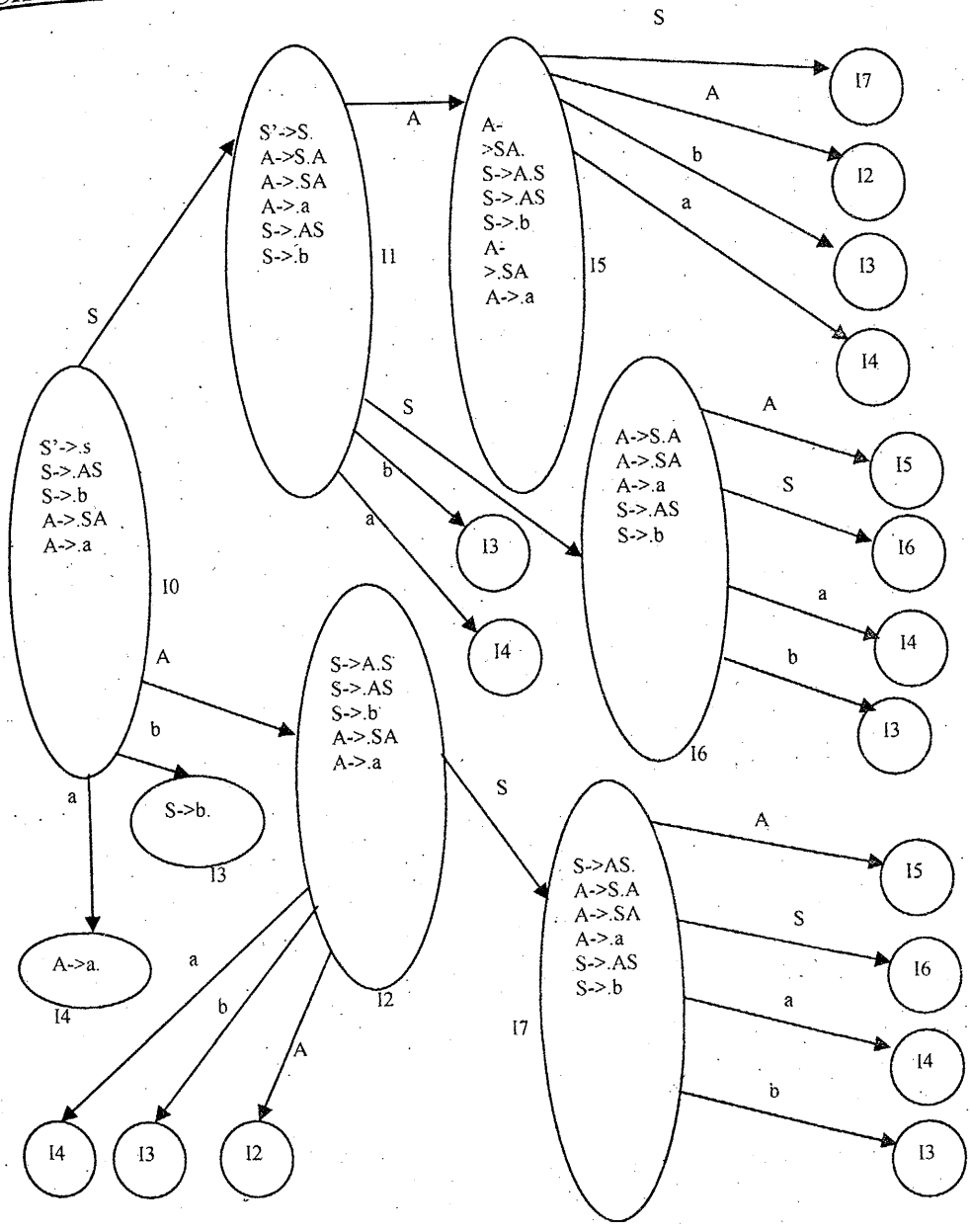
  S→AS

  S→b

  A→SA

  A→a

CHART:



SLR parsing table:

STATE	ACTION			GOTO	
	A	b	$	S	A
0	s4	s3	acc	1	2

1	s4	s3		6	5
2	s4	s3		7	2
3	r2	r2	r2		
4	r4	r4			
5	s4	s3		7	2
6	s4	s3		6	5
7	s4	s3		6	5

The grammar is in SLR as because there is no conflict.

5. a) Describe LR parsing with block diagram.
b) What are the main advantages of LR parsing?
c) Construct SLR parsing table for the grammar given below:

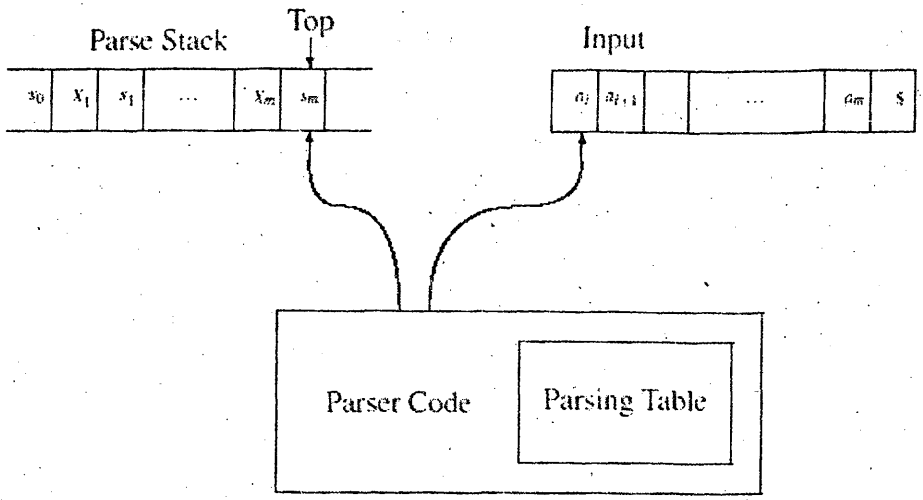
$S \rightarrow Cb$

$C \rightarrow bC \mid d$

[WBUT 2011]

Answer:

a) The block diagram of an LR parser is given below:



All LR-parsers have the same driving routine that uses a stack and through a state-based approach, tries to detect whether the stack contains the handle at its top. If so, the handle is replaced by the left hand side non-terminal for the handle.

The parsing algorithm uses two tables — ACTION and GOTO. The behavior can be summarized again as follows.

The parser sees s_m at the top of the stack and let a_i be the current input symbol.

Based on these, the parser can carry out one of the following operations:

- Shift-s. Here s is a state.

- Reduce by A → b
- Accept
- Error

Assuming that states are numbered $s_0, s_1, \ldots$ where s0 is a special case such that the parser begins with only s0 on the stack, the driving routine of an LR parser is:

```
push(0);
read_next_token();
for(;;)
{
s = top(); /* current state is taken from top of stack */
if (ACTION[s,current_token] == "s-i")
/* shift and go to state i */
{
push(current_token);
push(i);
read_next_token();
}
else if (ACTION[s,current_token] == "r-i")
/* reduce by rule i: X ? A1...An */
{
perform pop() 2 * n times;
s = top();
/* restore state before reduction from top of stack */
push(GOTO[s,X]); /* state after reduction */
}
else if (ACTION[s,current_token] == "succ")
success!!
else error();
}
```

b) The advantages of LR parsers are:

- Almost all programming languages have LR grammars.
- LR parsers take time and space linear in the size of the input (with a constant factor determined by the grammar).
- LR is strictly more powerful than LL (for example, every LL(1) grammar is also both LALR(1) and LR(1), but not vice versa).
- LR grammars are more "natural" than LL grammars (e.g., the grammars for expression languages get mangled when we remove the left recursion to make them LL(1), but that is not necessary for an LR(1) grammar).

c) We construct the augmented grammar:

$0.S' \rightarrow S$

$1.S \rightarrow Cb$

$2.C \rightarrow bC$

$3.C \rightarrow d$

Also, we note that

$FOLLOW(S)=\{\$\}$

$FOLLOW(C) = \{b\}$

We proceed to make the canonical collection of LR(0) items as follows:

$s_0 = CLOSURE(\{S' \rightarrow .S\}) = \{S' \rightarrow .S, S \rightarrow .Cb, C \rightarrow .bC, C \rightarrow .d\}$

$s_1 = CLOSURE(GOTO[s_0, S]) = \{S' \rightarrow S.\}$

$s_2 = CLOSURE(GOTO[s_0, C]) = \{S \rightarrow C.b\}$

$s_3 = CLOSURE(GOTO[s_0, b]) = \{C \rightarrow b.C, C \rightarrow .bC, C \rightarrow .d\}$

$s_4 = CLOSURE(GOTO[s_0, d]) = \{C \rightarrow d.\}$

$s_5 = CLOSURE(GOTO[s_2, b]) = \{S \rightarrow Cb.\}$

$s6 = CLOSURE(GOTO[s_3, C]) = \{S \rightarrow bC\}$

$GOTO(s_3, b) = s_3$

$GOTO(s_3, d) = s_4$

SLR parsing table:

STATE	ACTION			GOTO	
	Λ	b	$	S	A
0	s4	s3	acc	1	2
1	s4	s3		6	5
2	s4	s3		7	2
3	r2	r2	r2		
4	r4	r4			
5	s4	s3		7	2
6	s4	s3		6	5
7	s4	s3		6	5

The grammar is in SLR as because there is no conflict.

6. Write short notes on Handle pruning. **[WBUT 2012]**

Answer:

A handle of a string is a substring that matches the RHS of a production, and whose reduction to the lhs is one step along the reversal of a rightmost derivation. The process we went through can be viewed as "handle-pruning", where we're pruning the parse tree.

- A right-most derivation in reverse can be obtained by **handle-pruning**.

$$S=\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = \omega$$

input string

- Start from γ_n, find a handle $A_n \rightarrow \beta_n$ in γ_n, and replace β_n in by A_n to get γ_{n-1}.
- Then find a handle $A_{n-1} \rightarrow \beta_{n-1}$ in γ_{n-1}, and replace β_{n-1} in by A_{n-1} to get γ_{n-2}.
- Repeat this, until we reach S.

SYNTAX DIRECTED TRANSLATION

Multiple Choice Type Questions

1. An annotated parse tree is [WBUT 2006, 2007, 2009]
   a) a parse tree with attribute values shown at the parse tree nodes
   b) a parse tree with values of only some attributes shown at parse tree nodes
   c) a parse tree without attribute values shown at parse tree nodes
   d) a parse tree with grammar symbols shown at parse tree nodes

Answer: (a)

2. The edges in a flow graph whose heads dominate their tails are called:
   a) Back edges b) Front edges [WBUT 2008]
   c) Flow edges d) None of these

Answer: (a)

3. A parse tree showing the values of attributes at each node is called in particular
   a) Syntax tree b) Annotated parse tree [WBUT 2011]
   c) Syntax Direct parse tree d) Direct Acyclic graph

Answer: (b)

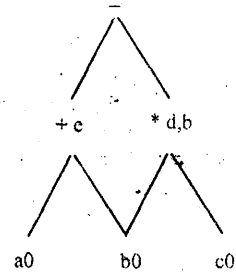
Short Answer Type Questions

1. Construct the DAG for the following basic block. [WBUT 2006, 2008, 2009]

   d := b * c
   e := a + b
   b := b * c
   a := e - d

Answer:

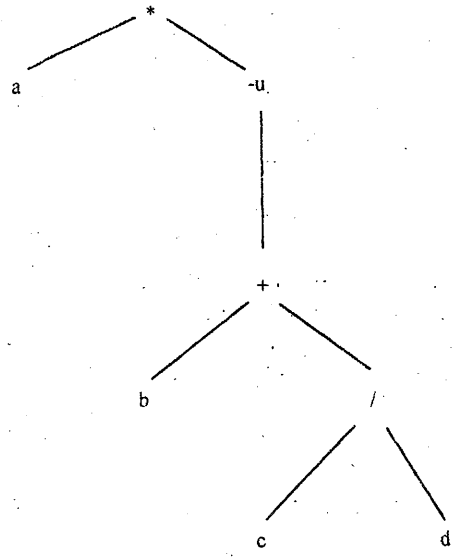
The required DAG is:



2. Translate the arithmetic expression a * - (b + c / d) into: [WBUT 2007, 2012]
Syntax tree

Answer:

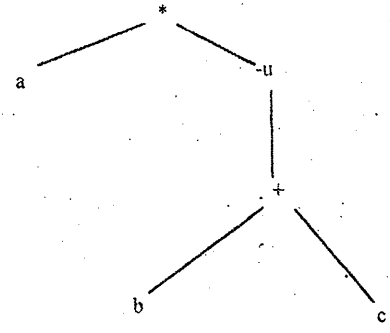
The Syntax Tree for the expression a * - (b + c / d) is as shown below:



3. Translate the arithmetic expression a * – (b + c) into: [WBUT 2008]
Syntax tree

Answer:

The syntax tree for the arithmetic expression a * -(b + c) is:



4. Explain inherited attribute and synthesized attribute for Syntax directed translation with suitable example. [WBUT 2010]

Answer:

Answer:

For Syntax Directed Translation, compilers extend the normal context-free grammar parse tree to an Annotated Parse Tree where *each* node of the tree is a record with a field for each attribute. The value of an attribute of a grammar symbol at a given parse tree node is defined by a *semantic rule* associated with the production used at that node.

There can be two kinds of attributes:

These attributes are computed from the values of the *attributes of the children nodes.*

These attributes are computed from the values of the *attributes of both the siblings and the children nodes.*

In a syntax directed definition, each production $\mathbf{A} \rightarrow \alpha$ is associated with a set of semantic rules of the form:

$$\mathbf{b = f(c_1, c_2, \ldots, c_n)}$$

where **f** is a function and **b** can be either a synthesized attribute of A and $c_1, c_2, \ldots, c_n$ are attributes of the grammar symbols in the right side of the production, OR an inherited attribute one of the grammar symbols in the right side of the production), and $c_1, c_2, \ldots, c_n$ are attributes of the grammar symbols in the production.

Example:

Syntax Rule	Semantic Rule	Comment
$D \rightarrow T\ L$	*L.in = T.type*	*L in* is inherited
$T \rightarrow$ **int**	*T.type = 'integer'*	*T type* is synthesized

5. What do you understand by L-attributed definitions? Illustrate with an example. **[WBUT 2011]**

Answer:

There are two classes of the syntax directed definitions — S-Attributed **Definitions**: where only synthesized attributes are used and L-Attributed Definitions where, in addition to synthesized attributes, inherited attributes may also be used in a restricted fashion.

Definition: Syntax directed definition is L-Attributed if each inherited attribute of X_j in a production $A \rightarrow X_1 X_2 \ldots . X_j \ldots X_n$, depends only on: i. The attributes of the symbols to the left of X_j, i.e., $X_1, X_2, \ldots, X_{j-1}$, and ii. The inherited attributes of A .

Every S-attributed definition is an L-attributed definition and an L-attributed definition is more conducive to LL (top-down) parsing due to a theorem which says that "Inherited attributes in L-Attributed Definitions can be computed by a Pre-order traversal of the parse-tree".

Consider the grammar for declaring variables in a restricted version (only 'char' and 'int' as allowed types and no pointers) of C:

$S \rightarrow T\ L$

$L \rightarrow L$ id | id

$T \rightarrow$ char | int

If each 'id' has to have the attribute type, then the second rule must be associated with the definition:

id.type = L.type

which is an example of an L-attributed definition.

6. Define grammar. What do you mean by ambiguity in grammar? Illustrate with suitable example. What is the necessity to generate parse tree? [WBUT 2012]

Answer:

1st Part: Grammar is a specification for the syntax of a programming language.

2nd Part: A grammar that produces more than one parse tree for some sentence is said to be ambiguous.

An ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for the same sentence.

3rd Part:

A parse tree is a data-structure generated by the parse which captures the 'meaning' of the input source code as per the language that the compiler caters to. Subsequent stage of the compiler like syntax-generated translation can convert the parse tree into the target code which can then be optimized further.

Long Answer Type Questions

1. Suppose a robot can be instructed to move one step east, north, west or south from its current position. A sequence of such instruction is generated by the following grammar:

Seq → Seq_1 instr | begin

Instr → east | north | west | south

i) Construct a syntax directed definition to translate an instruction sequence into a robot position.

ii) Draw a parse tree for: begin west south. [WBUT 2010]

Answer:

i) We shall use two attributes, $Seq.x$ and $Seq.y$, to keep track of the position resulting from an instruction sequence generated by the nonterminal Seq. initially, Seq generates **begin** and $Seq.x$ and $Seq.y$ are both initialized to 0.

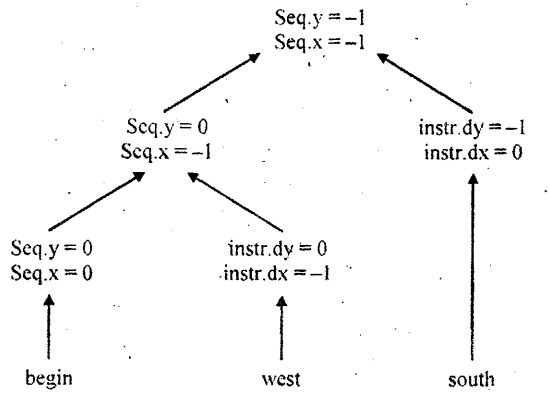
The change in position due to an individual instruction derived from non-terminal $instr$ is given by attributes $instr.dx$ and $inxtr.dy$.

A syntax-directed definition for translating an instruction sequence into a robot position is shown below:

PRODUCTION	SEMANTIC RULES
$Seq \rightarrow$ **begin**	$Seq.x = 0$; $Seq.y = 0$;

$Seq \rightarrow Seq_1\ instr$	$Seq.x = Seq_1.x + instr.dx;\ Seq.y = Seq_1.y + instr.dy;$
$instr \rightarrow$ **east**	$instr.dx = 1;\ instr.dy = 0;$
$instr \rightarrow$ **north**	$instr.dx = 0;\ instr.dy = 1;$
$instr \rightarrow$ **west**	$instr.dx = -1;\ instr.dy = 0;$
$instr \rightarrow$ **south**	$instr.dx = 0;\ instr.dy = -1;$

ii) The annotated parse tree for the expression is given below



2. What are the main contributions of Syntax Directed Translation in Compiler? Design a Dependency Graph and Direct acyclic graph for the string

a+ a * (b – c) + (b – c) * d **[WBUT 2010, 2011]**

Answer:

1st part:

For programming language grammars, the semantics of the input sentence (i.e. the source ode of the program being compiled) should get captured as a body of executable code (in compilers) or as direct execution, possibly in stages, as in an interpreter. Syntax-directed translation refers to a method of compiler implementation where the source language translation is completely driven by the parser. In other words, the parsing process and parse trees are used to direct semantic analysis and the translation of the source program.

The principle of Syntax Directed Translation states that *the meaning of an input sentence is related to its syntactic structure (i.e. to its Parse-Tree).*Syntax Directed Translation deals with formalisms for specifying translations for the source (programming) language constructs guided by the context-free grammar that describes the language. The complete sematic information (i.e., the ``object code'') for the given ``sentence'' (i.e., a program written in the language) *is incrementally computed from syntax analysis.*

For programming language compilers, the Syntax Directed Translation phase additionally checks and validates context-sensitive conditions like:

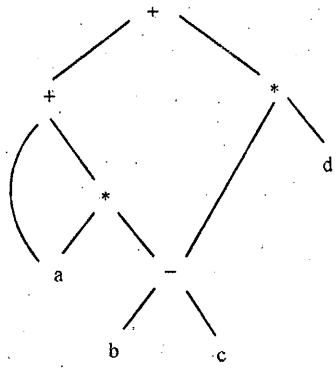
- Variables are declared before used.

- Type compatibility.
- Formal/actual parameter agreement in procedures.
- Resolve overloaded symbols.
- Others, depending upon that particular language

2nd part:

For the given expression, the Dependency Graph is simply the parse tree with every edge directed upwards in terms of dependency. This is because, because only one attribute, namely *E.val* is required in every (non-leaf) node.

The DAG for the expression is as given below:



3. Write short note on the following:

a) **Dependency graph** **[WBUT 2009]**

b) **Input buffering** **[WBUT 2009, 2011]**

c) **L-attributed definitions** **[WBUT 2009]**

Answer:

a) Dependency Graph:

The next logical stage in the translation process is to create a data structure called the Dependency Graph. The Dependency Graph is a superimposition on the annotated parse tree and sets up the dependencies between the attributes of the nodes.

A Dependency Graph is:

- Directed Graph.
- Shows interdependencies between attributes of the nodes of the annotated parse tree.

The following steps are carried out to construct a Dependency Graph:

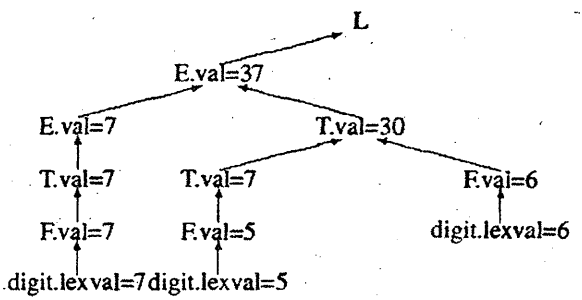
```
for each node n in the parse tree {
for each attribute a of the grammar symbol at n {
construct a node in the dependency graph for a
}
}
for each node n in the parse tree {
```

```
for each semantic rule b = f(c1, c2, ... ,cn)
for rule used at n {
for i= 1 to n {
construct an edge from the node for ci to the
node for b
}
}
}
```

The Dependency graph for a valid and translatable expression should be a Directed Acyclic Graph (or DAG). Otherwise, some attribute of a node would have circular dependence on itself which is ridiculous. To evaluate the attributes, first a topological sort is carried out on the DAG and that can now be used as a guide for the order of attribute evaluation. The examples given below should make the ideas clear.

Example:

Dependency Graph for 7 + 5 * 6 return



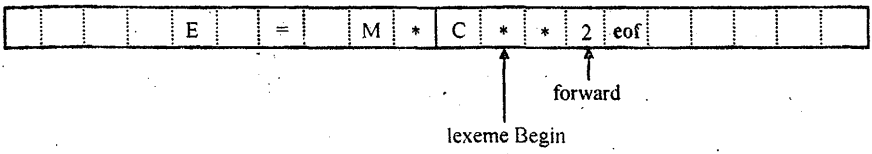
b) Input Buffering:

We should examine some ways that the simple but important task of reading the source program can be speeded. This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. There are many situations where we need to look at least one additional character ahead. For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for **id**. In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=. Thus, we shall introduce a two-buffer scheme that handles large lookaheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

Buffer Paris

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead

required to process a single input character. An important scheme involves two buffers that are alternately reloaded, as suggested in Figure below.



Using a pair of input buffers

Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read N characters into a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by **eof**, marks the end of the source file and is different from any possible character of the source program.

Two pointers to the input are maintained:

1. Pointer lexeme Begin marks the beginning of the current lexeme, whose extent we are attempting to determine.
2. Pointer forward scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter.

Once the next lexeme is determined, forward is set to the character at its right end. Them, after the lexeme is recorded as an attribute value of a token returned to the parser, lexeme Beg in is set to the character immediately after the lexeme just found. In the above figure, we see forward has passed the end of the next lexeme, ** (the Fortran exponentiation operator), and must be retracted one position to its left.

Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than N, we shall never overwrite the lexeme in its buffer before determining it.

c) L-attributed definitions:

L-Attributed Definitions contain both synthesized and inherited attributes but do not need to build a dependency graph to evaluate them.

Definition: A syntax directed definition is L-Attributed if each inherited attribute of Xj in a production $A \rightarrow X_1X_2 \ldots\ldots X_j \ldots\ldots X_n$, depends only on:

1. The attributes of the symbols to the left of X_j, i.e., $X_1;X_2; : : : ;X_j\text{-}1$, and
2. The inherited attributes of A.

Clearly, every S-attributed definition is an L-attributed definition. Also, an Lattribute definition is more conducive to LL (top-down) parsing.

Theorem: Inherited attributes in L-Attributed Definitions can be computed by a Pre-order traversal of the parse-tree.

Bottom-up Evaluation of L-attributed Grammars

For grammars with L-attributed definitions, special evaluation algorithms must be designed in case of bottom-up parsers. Such algorithms can handle most LR(1) grammars. The basic idea is to are take all translation actions to the right end of the production. The key observation here is that when a bottom-up parser reduces by the rule A!XY, it removes X and Y from the top of the stack and replaces them by A. Since the synthesized attributes of X is on the top of the stack, they thus can be used to compute attributes of Y. In real implementations, information contained deeper in the stack can be floated up by using special markers to mark the production we are currently in. Markers can also be used to perform error checking and other intermediate semantic actions.

TYPE CHECKING

Multiple Choice Type Questions

1. Which of the following is not true about dynamic type checking? [WBUT 2006, 2007, 2008, 2009, 2011]
   a) Type checking is done during the execution
   b) It increases the cost of execution
   c) All the type errors are detected
   d) None of these

Answer: (d)

2. Which one of the following is not true about dynamic checking? [WBUT 2010]
   a) It increases the cost of execution
   b) Type checking is done during execution
   c) All the type error are detected
   d) None of these

Answer: (d)

Short Answer Type Questions

1. Consider the following C declarations: [WBUT 2006]

```
type def struct {
    int a, b;
} CELL, *PCELL;

CELL f[100];

PCELL b(x, y) int x; CELL y; {K}
```

Write type expressions for the types f and b.

Answer:

The required type expressions are as follows:

$$f.type = array(100, (int \times a) \times (int \times b))$$

$$b.type = int \times (int \times a) \times (int \times b) \rightarrow \uparrow (int \times a) \times (int \times b)$$

2. What is type checking? Differentiate between Dynamic and Static Type checking. [WBUT 2010]

Answer:

For many programming languages, the compiler must check that the source program follows *Type Rules* of the language, i.e., both the syntactic and semantic conventions of the source language. This is called **static checking**. Type checking is targeted towards:

Compiler can detect meaningless or invalid code.

Provide useful compile-time information like memory alignment that can result in more efficient machine instructions.

Types can serve as a form of documentation.

Types allow programmers to think about programs at a higher level than the bit or byte.

A programming language is said to use static typing when type checking is performed during compile-time as opposed to run-time. Examples of languages that use static typing include C, C++, C#, Java, Fortran, Pascal, and Haskell.

A programming language is said to use dynamic typing when type checking is performed at run-time (also known as "late-binding") as opposed to at compile-time. Examples of languages that use dynamic typing include Javascript, Lisp, Perl, PHP, Python, Ruby, and Smalltalk.

RUNTIME ENVIRONMENT

Multiple Choice Type Questions

1. Symbol table can be used for [WBUT 2010]
   a) checking type compatibility b) suppressing duplicate error messages
   c) storage allocation d) all of these

Answer: (d)

Short Answer Type Questions

1. What is activation record? Explain clearly the components of an activation record. [WBUT 2012]

Answer: *Refer to Question of No. 1 of Long Answer Type Questions.*

Long Answer Type Questions

1. What is an activation record? [WBUT 2006, 2007, 2008]
When and why are those records used? [WBUT 2006, 2008]
List different fields of an activation record and state the purpose of those fields. [WBUT 2006, 2007, 2008]

Answer:

1st Part:

An Activation Record (also known as a "stack frame" of a "call stack") is a data structure containing the variables belonging to one particular scope (e.g. a procedure body), as well as links to other activation records.

2nd Part:

These variables may be the actual parameters passed on to the procedure or variables declared in the scope of the procedure. Activation records are usually created on the call stack on entry to a block and destroyed on exit.

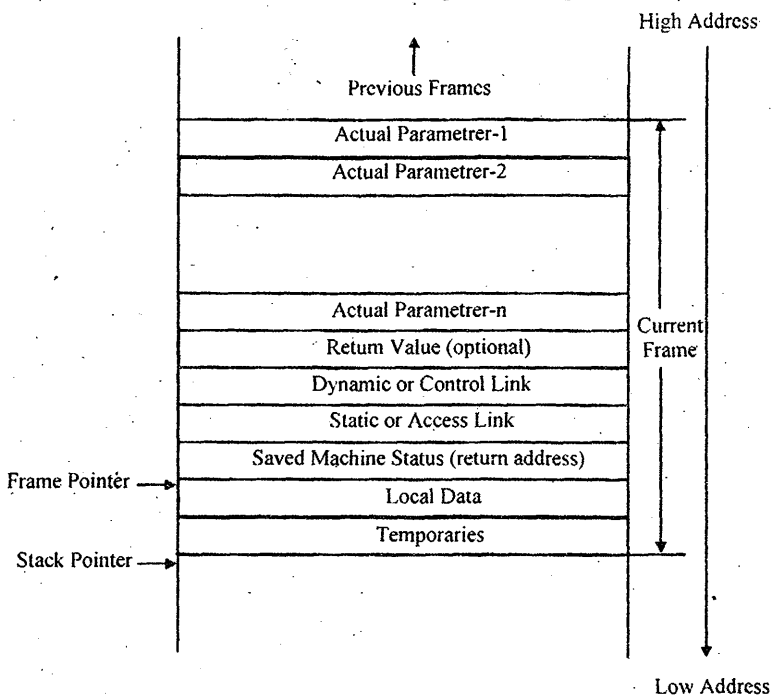
Variables in the current scope are accessed via the frame pointer which points to the current activation record. Variables in an outer scope are accessed by following chains of links between activation records. There are two kinds of link— the static link and the dynamic link.

Given below is a summary of functions of the activation record, depending on the language, operating system, and machine environment, etc.:

- Local data storage
- Parameter passing
- Evaluation stack (for example, temporary variables) for arithmetic or logical operations
- Pointer to current instance in object-oriented languages (e.g., this in C++)
- Enclosing subroutine context
- Other return states besides the return address — environment and/or machine specific

3rd Part:

A typical organization of an activation record is given in figure below:



Before the run-time environment creates the frame for the callee, it needs to allocate space for the callee's arguments. These arguments belong to the caller's frame, not to the callee's frame. There is a frame pointer (called FP) that points to the beginning of the frame. The stack pointer points to the first available byte in the stack immediately after the end of the current frame (the most recent frame).

There are many variations to this theme. Some compilers use displays to store the static chain instead of using static links in the stack. A display is a register block where each pointer points to a consecutive frame in the static chain of the current frame. This allows a very fast access to the variables of deeply nested functions.

Another important variation is to pass arguments to procedures using registers.

When a procedure starts running, the frame pointer and the stack pointer contain the same address. While the procedure is active, the frame pointer, points at the top of the parameters and return address. The stack pointer may be changed its value while a procedure is active but the frame pointer does not change its value while a procedure is active. The variables will always be the same distance from the unchanging frame pointer.

2. Write Short Note on Activation record **[WBUT 2010, 2011]**

Answer: ***Refer to Question No. 1 of Long Answer Type Questions.***

INTERMEDIATE CODE GENERATION

Multiple Choice Type Questions

1. Which of the following is not an intermediate code form? [WBUT 2010, 2012]
   a) Postfix notation
   b) Syntax trees
   c) Three address codes
   d) Quadruples

Answer: (c)

Short Answer Type Questions

1. Translate the expression -(a+b)*(c+d)+(a+b+c) into [WBUT 2006]
   i) quadruples ii) Triples iii) indirect triples.

Answer:

Quadruple:

	op	Res	arg1	arg2
(101)	t1	+	a	b
(102)	t2	uniminus	t1	
(103)	t3	+	c	d
(104)	t4	*	t2	t3
(105)	t5	+	a	b
(106)	t6	+	t5	c
(107)	t7	+	t4	t6

Triple:

	op	arg1	arg2
(101)	+	a	b
(102)	uniminus	(101)	
(103)	+	c	d
(104)	*	(102)	(103)
(105)	+	a	b
(106)	+	(105)	c
(107)	+	(104)	(106)

Indirect Triple:

	statement
(1)	(101)
(2)	(102)
(3)	(103)
(4)	(104)
(5)	(105)
(6)	(106)
(7)	(107)

	op	arg1	arg2
(101)	+	a	b
(102)	uniminus	(101)	
(103)	+	c	d
(104)	*	(102)	(103)
(105)	+	a	b
(106)	+	(105)	c
(107)	+	(104)	(106)

2. State the utility of 3-address instruction and DAG. **[WBUT 2007]**

Answer:

The utility of DAG-s are mostly in eliminating common subexpressions in a basic block during code optimization. While it is possible to have syntax directed translation schemes for expressions that generate DAGs directly, doing the same for control flow statements is difficult. On the other hand, syntax of most programming languages have straightforward translation schemes to 3-address codes. The set of 3-address codes thus generated can then be broken up into basic blocks and DAG-s can be algorithmically constructed for each basic block for carrying out optimization.

3. Generate the 3 address code for the following program fragment **[WBUT 2007]**

while ((A < C) and (B > D)) do
if (A < 1) then C = C + 1
else
while (A < = D) do
A = A + 3.

Answer:

101: if A < C goto 103
102: goto 105 ===> "Next" of this sentence
103: if B > D goto 105
104: goto 105 ===> "Next" of this sentence
105: if A < 1 goto 107
106: goto 110
107: t1 := C + 1
108: C := t1
109: goto 101
110: if A <= D goto 112
111: goto 101
112: t2 := A + 3
113: A := t2

114: goto 110
115: goto 101

4. Translate the arithmetic expression a * -(b + c) into: **[WBUT 2008]**
a) Three-address code
b) Postfix notation
Answer:
a) The three address code for the arithmetic expression a * -(b + c) is:
t1 = b + c
t2 = -t1
t3 = a * t2

b) The postfix expression for the arithmetic expression a * -(b + c) is:
abc+-*

5. Explain the following terms with the example given below: **[WBUT 2008]**
a) Quadruples
b) Triples
c) Indirect Triples
Example: a := b * (c + d / b) - (e * f)
Answer:
A three-address code, i.e., a collection of several three-address statements, can be represented in three forms — quadruples, triples and indirect triples.
The set of three-address codes for the example is:
t1 := d / b
t2 := c + t1
t3 := b * t2
t4 := - t3
t5 := e * f
t6 := t4 - t5
a := t6
A quadruple is a record structure with four fields — op, arg1, arg2, and result. The op field contains an internal code for the operator. The three address statement **x := y op z** is represented by placing y in arg 1, z in arg 2 and x in result, Statements with unary operators like **x := - y** or **x := y** do not use arg2. Conditional and unconditional jumps put the target label in result. The quadruples for the example code is:

	op	Res	arg1	arg2
(101)	t1	/	d	b
(102)	t2	+	c	t1
(103)	t3	*	b	t2
(104)	t4	uniminus	t3	
(105)	t5	*	e	f
(106)	t6	-	t4	t5
(107)	a	:=	t6	

If three-address statements are represented by records with only three fields – op, arg1 and arg2, we call them triples. The fields arg1 and arg2, for the arguments of op, are either pointers to the symbol table (for programmer defined names or constants) or pointers into the triple structure (for temporary values). The triples for the given code is:

	op	arg1	arg2
(101)	/	d	b
(102)	+	c	(101)
(103)	*	b	(102)
(104)	uniminus	(103)	
(105)	*	e	f
(106)	-	(104)	(105)
(107)	:=	a	(106)

Another implementation of three-address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves.
This implementation is naturally called indirect triples. The indirect triples for the given code is:

	statement
(1)	(101)
(2)	(102)
(3)	(103)
(4)	(104)
(5)	(105)
(6)	(106)
(7)	(107)

	op	arg1	arg2
(101)	/	d	b
(102)	+	c	(101)
(103)	*	b	(102)
(104)	uniminus	(103)	
(105)	*	e	f
(106)	-	(104)	(105)
(107)	:=	a	(106)

Long Answer Type Questions

1. a) Distinguish between quadruples, triples and indirect triples for the expression:
a = b * - c + b * - c **[WBUT 2007, 2009]**
b) Translate the arithmetic expression a * - (b + c / d) into: **[WBUT 2007, 2012]**
i. Postfix notation
ii. 3-address code.

c) Generate machine code for the following instruction:

X = a / - (b * c) - d

Assume 3 registers are available. **[WBUT 2007]**

Answer:

a) Quadruple:

	res	op	arg1	arg2
(101)	t1	uniminus	c	
(102)	t2	*	b	t1
(103)	t3	+	t2	t2

Triple:

	op	arg1	arg2
(101)	uniminus	c	
(102)	*	b	(101)
(103)	+	(102)	(102)

Indirect Triple:

	Statement
(1)	(101)
(2)	(102)
(3)	(103)

	op	arg1	arg2
(101)	uniminus	c	
(102)	*	b	(101)
(103)	+	(102)	(102)

b) (i) The Postfix notation for the expression a * - (b + c / d) is **abcd/+-*.**

(ii) The 3-Address code for the expression a * - (b + c / d) is:

t1 = c / d

t2 = b + t1

t3 = -t2

t4 = a * t3

c) The machine code for X = a / - (b * c) - d is:

MOV R1, a

MOV R2, b

MUL R2, c

NEG R2

DIV R1, R2
MOV R3, d
SUB R1, R3
MOV X, R1

2. a) What are the differences among Quadruples, Triples and Indirect Triples? **[WBUT 2009]**

OR,

Differentiate Quadruple, Triples and Indirect triples with example. **[WBUT 2010]**

b) Generate machine code for the followings instruction: **[WBUT 2009]**

v = a + (b* c) – d.

Answer:

a) The difference between triples and quadruples may be regarded as a matter of how much indirection is present in the representation. When we ultimately produce target code, each name, temporary or programmer-defined, will be assigned some runtime memory location. This location will be placed in the symbol-table entry for the datum. Using the quadruple notation, a three-address statement defining or using a temporary can immediately access the location for that temporary via the symbol table. A more important benefit of quadruples appears in an optimizing compiler, where statements are often moved around. Using the quadruple notation, the symbol table interposes an extra degree of indirection between the computation of a value and its use. If we move a statement computing x, the statements using x require no change. However, in the triples declaration. moving a statement that defines a temporary value requires us to change all references to that statement in the arg1 and arg2 arrays. This problem makes triples difficult to use in an optimizing compiler. Indirect triples present no such problem. A statement can be moved by reordering the statement list. Since pointers to temporary values refer to the right-side arg1 and arg2 arrays, which are not changed, none of those pointers need be changed. Thus, indirect triples look very much like quadruples as far as their utility is concerned. The two notations require about the same amount of space and they are equally efficient for reordering of codes with ordinary triples. Allocation of storage to those temporaries needing it must be deferred in the code generation phase. However, indirect triples can save some space compared with quadruples if the same temporary value is used more than once. The reason is that two or more entries in the statement array can point to the same line of the op-arg1-arg2 structure.

b)

MOV R1,a
MOV R2,b
MUL R2,c
ADD R1,R2

SUB R1,d
MOV v,R1

3. Translate the expression a = -(a+ d) * (c+ d) + (a + b + c) into
a) Quardruple
b) Triple
c) Indirect Triple **[WBUT 2010, 2011]**

Answer

The quadruples set for the given code is:

	Op	*arg1*	*arg2*	*result*
101	+	a	d	t_1
102	uniminus	t_1		t_2
103	+	c	d	t_3
104	*	t_2	t_3	t_4
105	+	a	b	t_5
106	+	t_5	c	t_6
107	+	t_4	t_6	t_7
108	assign	a	t_7	

The Triples set for the given code is:

	Op	*arg1*	*arg2*
101	+	a	d
102	uniminus	*(101)*	
103	+	c	d
104	*	*(102)*	*(103)*
105	+	a	b
106	+	*(105)*	*c*
107	+	*(104)*	*(106)*
108	assign	a	*(107)*

For indirect triples, the triples set as above would be accompanied with another level of indirection through a statement table like:

	statement
1	*(101)*
2	*(102)*
3	*(103)*

4	*(104)*
5	*(105)*
6	*(106)*
7	*(107)*
8	*(108)*

4. a) Distinguish between quadruples, triples and indirect triples for the expression.

$x = y * -z + y * -z$ **[WBUT 2012]**

Answer:

Quadruple:

	res	op	arg1	arg2
(101)	t1	uniminus	z	
(102)	t2	*	y	t1
(103)	t3	+	t2	t2

Triple:

	op	arg1	arg2
(101)	uniminus	z	
(102)	*	y	(101)
(103)	+	(102)	(102)

Indirect Triple:

	Statement
(1)	(101)
(2)	(102)
(3)	(103)

	op	arg1	arg2
(101)	uniminus	z	
(102)	*	y	(101)
(103)	+	(102)	(102)

b) While the three-address code for the following C program:

```
main ( )
    {
        int x = 1;
        int y[20];
        while (x ≤ 20)
         y[x] = 0;
    }
```

[WBUT 2012]

Answer:

100 : if (x<=20) goto 102
101 : goto 150
102 : T1=addr(y)
103 : T1[20]=0
104 : goto 100
105 :

5. Write short note on Backpatching. **[WBUT 2007]**

Answer:

The problem in generating three-address codes in a single pass is that we may not know the labels that control must go to at the time jump statements are generated. So to get around this problem a series of branching statements with the targets of the jumps temporarily left unspecified is generated.

BackPatching is the operation of putting the appropriate target address instead of labels when the proper address of the label is determined.

Backpatching Algorithms perform three types of operations:

- makelist(i). This creates a new list containing only i, an index into the array of quadruples (i.e., the address of a three-address code) and returns pointer to the list it has made.
- merge(i; j). This concatenates the lists pointed at by i and j and returns a pointer to the concatenated list.
- backpatch(p; i). This inserts i as the target address for each of the statements on the list pointed to by p.

An example translation for the rule

$E \rightarrow E^{(1)}$ or $ME^{(2)}$ is

{

backpatch(E1.falselist,M.quad)

E.truelist = merge(E1.truelist,E2.truelist)

E.falselist = E2.falselist}

CODE OPTIMIZATION

Multiple Choice Type Questions

1. Which of the following is not a loop optimization? [WBUT 2006, 2008, 2009, 2011]

a) Induction variable elimination b) Loop unrolling

c) Loop jamming d) Loop heading

Answer: (d)

2. A basic block can be analyzed by [WBUT 2010, 2012]

a) DAG b) Flow graph c) Graph with cycles d) None of these

Answer: (a)

3. The method which merges the bodies of two loops is [WBUT 2010]

a) loop unrolling b) loop ramming

c) constant folding d) none of these

Answer: (b)

4. Optimization(s) connected with $x:=x+0$ is/are [WBUT 2012]

a) peephole and algebraic b) reduction in strength and algebraic

c) peephole only d) loop and peephole

Answer: (c)

Long Answer Type Questions

1. a) What is Peephole optimization? [WBUT 2006, 2008, 2009]

b) Consider some interblock code optimization without any data-flow analysis by treating each extended basic block as if it is a basic block.

Give algorithms to do the following optimizations within an extended basic block. In each case, indicate what effect on other extended basic blocks a change within one extended basic block can have. [WBUT 2006, 2008, 2009]

i. Common subexpression elimination.

ii. Constant folding.

iii. Copy propagation.

Answer:

a) Algorithmic code-generations strategy as in syntax-directed translation, often produce target code that contains redundant instructions and constructs that are not optimal. 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 peephole need not be contiguous, although some implementations do require this. Program transformations that are characteristic of peephole optimization are:

- Redundant instructions elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

Redundant Loads and Stores

In the instructions sequence

MOV R0, a

MOV a, R0

it is possible to delete the second instruction because whenever that is executed, the previous instruction will ensure that the value of a is already in register R0. However, If there had been a label with the second instruction, one could not be sure that it will always executed immediately after the previous one and so may not remove it.

Unreachable Code

Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1.In C, the source code might look like:

```
#define debug 0
...
if (debug) {
hspace15ptprint debugging information
}
```

In the intermediate representations the if-statement may be translated as:

```
if debug =1 goto L1
goto L2
L1: print debugging information
L2:
```

One obvious peephole optimization is to eliminate jumps over jumps. Thus, no matter what the value of debug, the code above can be replaced by:

```
if debug !=  1 goto L2
print debugging information
L2:
```

As "debug" evaluates to the constant 0, the code becomes:
if 0 != 1 goto L2
print debugging information
L2:
As the argument of the "if" in the first statement evaluates to a constant 'true', it can be replaced by goto L2. Then all the statement that print debugging information are manifestly unreachable and can be eliminated one at a time.

Flow-of-Control Optimizations
The intermediate codes sometimes generate jumps to jumps or jumps to conditional jumps (or vice versa). Such unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations: We can replace the jump sequence
goto L1
...
L1 : gotoL2
by the sequence
goto L2
...
L1: goto L2

If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump.
Similarly, the sequence

if a < b goto L1
...
L1: goto L2
can be replaced by
if a < b goto L2
...
L1: goto L2

Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

goto L1
...
L1:if a < b goto L2
L3:

may be replaced by

L1:if a < b goto L2
goto L3
...
L3:

While the number of instructions in the last two examples is the same, we sometimes ski the unconditional jump in the second one, but never in the first one. Thus the secon example is superior in execution time.

Algebraic Simplification

A large number of algebraic simplification that can be attempted through peephol optimization. Only a few algebraic identities occur frequently enough that it is wort considering implementing them .For example, statements such as

x := x + 0

or

x := x * 1

are often produced by straightforward intermediate code-generation algorithms, and the can be eliminated easily through peephole optimization. Common sub-expressions ma also be eliminated during peephole optimization. Its possible that a large amount of dea (useless) code may exist in the program. Eliminating these will definitely optimize th code Reduction in strength is also used wherever possible.

Use of Machine Idioms

The target machine may have hardware instructions to implement certain specifi operations efficiently. Detecting situations that permit the use of these instructions ca reduce execution time significantly. For example, some machines have auto increme and auto-decrement addressing modes. These add or subtract one from an operand befor or after using its value. The use of these modes greatly improves the quality of code whe pushing or popping a stack, as in parameter passing. These modes can also be used i code for statements like i : = i + 1.

b) A extended basic block is a sequence of blocks B1;B2; : : : ;Bk such that for i _ i < k Bi is the only predecessor of Bi+1 and B1 does not have a unique predeccor.

We note that an extended basic block is basically a tree of basic blocks rooted at B where an edge is a control-flow edge. We start with B1 and its DAG as being the curren DAG.

We can "visit" all the basic blocks in the tree in a "depth-first" manner. As we mov down a node (say Bi, we modify the current DAG by "adding" the DAG for Bi to th

current DAG. On the other hand, if we move up, from Bi we "subtract" the DAG for Bi from the current DAG. This can be achieved if we keep a stack of DAG-s. When we move down to Bi, we merge the DAG of Bi with the current DAG and push it into the stack. Conversely, when we move up, we simply pop out the current DAG from the stack.

We carry out usual method of optimization using the current DAG (by assuming that it is the only DAG of the current basic block) after moving down to a new node Bi by carrying out common subexpression elimination, constant folding and copy propagation only for the codes in Bi. This ensures that no basic block is used more than once for code generation.

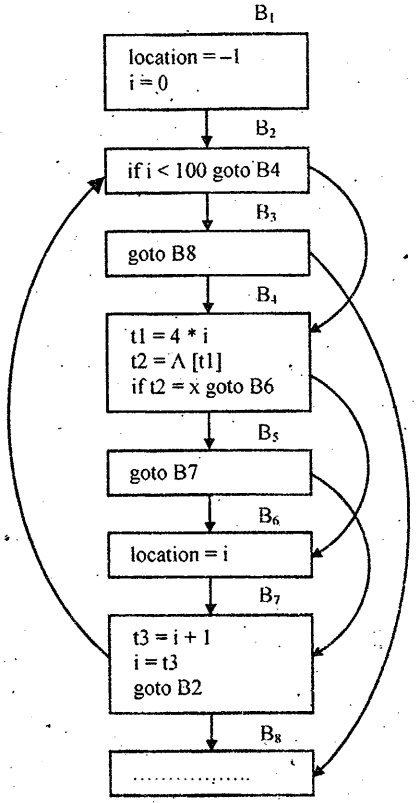
In all the cases, a change within one extended basic block will have no effect on other extended basic blocks because no two extended basic block shares any basic block. Hence, changes would remain localized. However, forming data-flow equations may become intractable.

2. Draw the flow graph for the following code: **[WBUT 2010, 2011]**

i) location = - 1
ii) i = 0
iii) i < 100 goto 5
iv) goto 13
v) $t_1 = 4i$
vi) $t_2 = A [t_1]$
vii) if $t_2 = x$ goto 9
viii) goto 10
ix) location = i
x) $t_3 = i + 1$
xi) $i = t_3$
xii) goto 3
xiii).....

Answer:

We get the ``block leaders" at addresses 1 (since this is the first statement), 3 (since this a target of the goto statement at 12), 4 (immediately after a conditional goto), 5 (immediately after a goto), 8 (immediately after a conditional goto), 9 (immediately after a goto), 10 (since this a target of the goto statement at 8) and 13 (immediately after a goto). From this, we get the basic blocks and the Flow Graph as:



3. Write short note on the following:

a) Issues in the design of a code generator **[WBUT 2006]**

b) Peephole Optimization **[WBUT 2007, 2010, 2011]**

c) Code optimization **[WBUT 2008]**

d) Loop optimization **[WBUT 2009]**

Answer:

a) Issues in the design of a code generator:

While the details are dependent on the target language and the operating system, certain issues are inherent in almost all code generation problems.

They are:

Input to Code Generator: What intermediate form of code forms the input to the code generator — postfix code; three-address code (as quadruples or triples or indirect triples), syntax trees, DAG-s, etc. It is assume that prior machine code generation the front end has scanned, parsed, and translated the source program into the intermediate representation, so the values of names appearing in the intermediate language can be represented by quantities that the target machine can directly manipulate.

Also, the necessary type checking must have taken place and type conversion operators have been inserted wherever necessary and obvious semantic errors (e.g., attempting to

index an array by a floating point number) have already been detected. The code generation phase can therefore proceed on the assumption that its input is free of errors.

Target Code: The output of the code generator is the target code which may take on a variety of forms like absolute machine language, relocatable machine language, or assembly language.

Memory Management: Mapping of 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. While machine code is being generated, labels in the intermediate representation statements have to be converted to addresses of instructions. This process is analogous to the "backpatching" method.

Instruction Selection: The nature of the instruction set of the target machine determines the difficulty of instruction selection. The uniformity and completeness of the instruction set are important factors. If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling. Instruction speeds and machine idioms are other important factors. A target machine with a rich instruction set may provide several ways of implementing a given operation. Since the cost differences between different implementations may be significant, a naive translation of the intermediate code may lead to correct, but unacceptably inefficient target code.

Register Allocation: Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilization of register is particularly important in generating good code. The use of registers is often subdivided into two subproblems—register allocation, where the set of variables that will reside in registers at a point in the program are selected and register assignment, during which the specific register that a variable will reside in is picked. Finding an optimal assignment of registers to variables is difficult, even with single register values. [Choice of Evaluation Order] 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. Picking a best order is another difficult problem.

b) Peephole Optimization:

Refer to Question No. 1(a) of Long Answer Type Questions.

c) Code optimization:

In the context of a program written in a high level language and then compiled by a suitable compiler, the final machine executable code, leaves enough scope for optimization. There can be two dimensions to the optimization:

- We may try to optimize the code such that it runs faster.
- We may want the compiled code to occupy as less memory as possible. In many cases, reduction in size amounts to reduction in run-time (since less number of instructions gets executed) but quite often, the two dimensions are often antagonistic to each other.

There can be several levels at which the code can be optimized:

Design Level: Here the designer (human being) selects the best possible algorithms and tries to implement that by writing good quality code.

Source Code Level: Here the programmer (human being) tries to achieve best possible coding quality by avoiding obvious slowdowns.

Compile Level: Here, an optimizing compiler tries to ensure that the executable program is optimized at least as much as the compiler can predict.

Assembly Level: A human programmer writing code using an assembly language designed for a particular hardware platform will normally produce the most efficient code since the programmer can take advantage of the full repertoire of machine instructions.

Compiler level code optimization involves the application of rules and algorithms (by the compiler) on the generated code at different level to transform the same to a code that is faster, smaller, more efficient, and so on. Code optimization is therefore the process of transformation of the generated non-optimal code to an optimal one.

d) Loop optimization:

Loops, especially the inner loops are where 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 that increases the amount of code outside that loop. Three techniques are important for loop optimization:

- Code Motion. Moving "loop invariant" code outside a loop.
- Induction-variable Elimination. Eliminating all but one variable that change in step within a loop.
- Reduction in Strength. Replacing an expensive operation by a cheaper one, such as a multiplication by an addition.

Code Motion

An expression that yields the same result independent of the number of times a loop is executed is called a loop-invariant computation.

In code motion, the transformation takes 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)
```

Code motion will result in the equivalent of

```
t = limit - 2;
while (i <= t)
```

Induction Variable Elimination and Reduction in Strength

An induction variable is a variable that gets increased or decreased by a fixed amount on every iteration of a loop, or is a linear function of another induction variable. For example, in the following loop, i and j are induction variables:

```
for (i = 0; i < 10; ++i) {
        j = 17 * i;
}
```

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. Strength reduction is an optimization technique where a costly operation is replaced with an equivalent but less expensive operation. Some examples are (assume 'j' is an integer and 'x' is a floating point variable):

- "j * 16" can be replaced by "j << 4"
- "j / 16" can be replaced by "j >> 4"
- "j * 2" can be replaced by "j + j"
- "2.0 * x" can be replaced by "x + x"
- Even "j * 15" can be replaced by "(j << 4) - j"

Induction variable elimination quite often goes hand in hand with strength reduction.

QUESTION 2006

Group – A
(Multiple Choice Type Questions)

1. Choose the correct alternatives for the following:

i) Cross-compiler is a compiler
- a) which is written in a language that is different from the source language
- b) that generates object code for its host machine
- c) which is written in a language that is same as the source language
- ✓d) that runs on one machine but produces object code for another machine

ii) YACC builds up
- a) SLR parsing table
- ✓b) canonical LR parsing table
- c) LALR parsing table
- d) none of these

iii) Given a grammar $G=(\{E\},\{id,+\},P,E)$; where, P is given by $E \to E+E, E \to id$. Then FOLLOW (E) will contain
- a) $\{\$\}$
- b) $\{+\}$
- ✓c) $\{\$,+\}$
- d) $\{\$,id,+\}$

iv) Which of the following expressions has no 1-value?
- a) $a[i+1]$
- b) a
- ✓c) 3
- d) $*a$

v) An annotated parse tree is
- ✓a) a parse tree with attribute values shown at the parse tree nodes
- b) a parse tree with values of only some attributes shown at parse tree nodes
- c) a parse tree without attribute values shown at parse tree nodes
- d) a parse tree with grammar symbols shown at parse tree nodes

vi) Which of the following is not true about dynamic type checking?
- a) Type checking is done during the execution
- b) It increases the cost of execution
- c) All the type errors are detected
- ✓d) None of these

vii) Consider the statement "$fi(x>=10)$", where '*if*' has been misspelled. The error is detected by the compiler in the phase
- a) lexical analysis
- b) syntax analysis
- ✓c) semantic analysis
- d) syntactic analysis

viii) Which of the following is not a loop optimization?
- a) Induction variable elimination
- b) Loop unrolling
- c) Loop jamming
- ✓d) Loop heading

ix) A dangling reference is a
- ✓a) pointer pointing to storage which is freed
- b) pointer pointing to nothing
- c) pointer pointing to storage which is still in use
- d) pointer pointing to uninitialized storage

x) If a grammar is LALR (1) then it is necessarily
a) SLR (1) b) LR (1) c) LL (1) ✓d) none of these

Group – B
(Short Answer Type Questions)

2. Give the NFA for the following Regular Expression $\left(0^* 1^+\right)^* 0^*$. Then find a DFA for the same language.

See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 1.

3. When does a lexical analyzer report an error? Write some error recovery actions taken by a lexical analyzer when it finds an error.

See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 2.

4. What is 'handle'? Consider the grammar: $E \rightarrow E+n \mid E X n \mid n$. For a sentence $n+n \times n$, write the handles in the right-sentential form of the reduction. What is predictive parsing?

See Topic: PARSING AND CONTEXT FREE GRAMMAR, Short Answer Type Question No. 1.

5. Consider the following *C* declarations:

```
type def struct {
    int a, b;
} CELL, *PCELL;

CELL f[100];

PCELL b(x, y) int x; CELL y; {K}
```

Write type expressions for the types *f* and *b*.

See Topic: TYPE CHECKING, Short Answer Type Question No. 1.

6. Translate the expression:

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

i) quadruples
ii) triples
iii) indirect triples.

See Topic: INTERMEDIATE CODE GENERATION, Short Answer Type Question No. 1.

Group – C
(Long Answer Type Questions)

7. a) Define an operator grammar.

b) Given a grammar $G = \left(\{E, T, F\}, \{id, +, *, (,)\}, P, E\right)$; where, P is given by

$$E \to E+T \mid T, T \to T*F \mid F, F \to (E) \mid id$$

Construct the SLR (1 parsing table for G).

a) **See Topic: OPERATOR PRECEDENCE PARSING, Short Answer Type Question No. 1.**

b) **See Topic: BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY, Long Answer Type Question No. 1.**

8. Which parser is used for the implementation of recursive descent parsing? Draw a model diagram for that parser. Construct the parsing table for the grammar

$$E \to TE'$$
$$E' \to +TE' \mid \in$$
$$T \to FT'$$
$$T' \to *FT' \mid \in$$
$$T \to (E) \mid id$$

With the help of parsing table and other components how the string '$id + id * id$' is parsed?

See Topic: TOP-DOWN PARSING, Long Answer Type Question No. 1.

9. a) Construct the DAG for the following basic block.

$$d := b * c$$
$$e := a + b$$
$$b := b * c$$
$$a := e - d$$

b) What is Peephole optimization?

c) Consider some interblock code optimization without any data-flow analysis by treating each extended basic block as if it is a basic block. Give algorithms to do the following optimizations within an extended basic block. In each case, indicate what effect on other extended basic blocks a change within one extended basic block can have.

i) Common subexpression elimination.
ii) Constant folding.
iii) Copy propagation.

a) **See Topic: SYNTAX DIRECTED TRANSLATION, Short Answer Type Question No. 1.**

b) & c) **See Topic: CODE OPTIMIZATION, Long Answer Type Question No. 1.**

10. a) What is an activation record? When and why are those records used? List different fields of an activation record and state the purpose of those fields.

b) What do you understand by terminal table and literal table?

a) See Topic: RUNTIME ENVIRONMENT, Long Answer Type Question No. 1.
b) See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 3.

11. Write short notes on the following:
    a) Symbol table organization
    b) Issues in the design of a code generator
    c) YACC

a) See Topic: INTRODUCTION TO COMPILER, Long Answer Type Question No. 3(a).
b) See Topic: CODE OPTIMIZATION, Long Answer Type Question No. 3(a).
c) See Topic: BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY, Long Answer Type Question No. 6(a).

QUESTION 2007

Group –A
(Multiple Choice Type Questions)

1. Choose the correct alternatives for the following:

i) A grammar in which every production rule of the form $x \rightarrow a$ is known as
    a) LL (0) b) LL (1) c) Context Free ✓d) regular

ii) Which data structure is mainly used during shift-reduce parsing?
    a) Pointers b) Arrays ✓c) Stacks d) Queues

iii) Semantic analysis is applied to determine
    a) the argument types b) the type of intermediate results
    ✓c) both (a) & (c) d) none of these

iv) A language L allows declaration of arrays whose sizes are not known during compilation. It is required to make efficient use of memory. Which one of the following is true?
    a) A compiler using static memory allocation technique can be written for L
    b) A compiler cannot be written for L, an interpreter must be used
    ✓c) A compiler using dynamic memory allocation technique can be written for L
    d) None of these

v) If x is a terminal then FIRST (x) is
    a) ε ✓b) {x} c) x^* d) none of these

vi) YACC builds up
    a) SLR parsing table b) LALR parsing table
    ✓c) canonical LR parsing table d) none of these

vii) Which of the following is not true about dynamic type checking?
a) It increases the cost of execution b) Type checking is done during the execution
c) All the type errors are detected ✓d) None of these

viii) An annotated parse tree is
a) a parse tree with values of only some attributes shown at parse tree nodes
✓b) a parse tree with attribute values shown at the parse tree nodes
c) a parse tree without attribute values shown at the parse tree nodes
d) a parse tree with grammar symbols shown at the parse tree nodes

ix) If a grammar is LALR (1) then it is necessarily
a) LL (1) b) SLR (1) c) LR (1) ✓d) none of these

x) In a block structured language if procedure A invokes B with nested depths and n_B respectively, then
✓a) $n_B - n_A \leq 1$ b) $n_B - n_A \geq 1$ c) $n_B - n_A = 1$ d) none of these

Group – B
(Short Answer Type Questions)

2. What is activation record? Explain clearly the components of an activation record.
See Topic: **RUNTIME ENVIRONMENT, Short Answer Type Question No. 1.**

3. State the utility of 3-address instruction and DAG.
See Topic: **INTERMEDIATE CODE GENERATION, Short Answer Type Question No. 2.**

4. Generate the 3 address code for the following program fragment

```
While ((A < C) and (B > D)) do
If (A < 1) then C = C + 1
else
while (A < = D) do
A = A + 3.
```

See Topic: **INTERMEDIATE CODE GENERATION, Short Answer Type Question No. 3.**

5. With the help of a block diagram, show each phase including symbol table and error handler of a compiler.
See Topic: **INTRODUCTION TO COMPILER, Short Answer Type Question No. 1.**

6. What is lookahead operator? Give an example. With the help of the lookahead concept show how identifiers can be distinguished from keywords.
See Topic: **LEXICAL ANALYSIS, Short Answer Type Question No. 4.**

Group – C
(Long Answer Type Questions)

7. a) Consider the (very artificial) language, over the alphabet of letters and digits and the dollar ($), having the following three kinds of tokens : numbers, consisting of one or more consecutive digits; short identifiers, consisting of a single letter and long identifiers; consisting of one or more letters followed by a single $.

i) Write down a regular expression for reach of the three token patterns.
ii) Draw DFA for each expression.
iii) Combine the DFA's into a common NFA with a common start state and distinct final states for each token.
iv) Convert your combined NFA into a DFA using whatever algorithm you like.
v) Suppose you use this DFA to perform lexical analysis, backtracking to the most recently encountered final state when necessary. Give an example of an input that will require backtracking and re-reading exactly 5 characters.

b) Given a grammar:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id$

Which is a set of valid items for a viable prefix E +.

a) See Topic: LEXICAL ANALYSIS, Long Answer Type Question No. 1.

b) See Topic: BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY, Short Answer Type Question No. 1.

8. a) Eliminate left recursion from the following grammar:

$E \rightarrow E + T \mid T$

$T \rightarrow TF \mid F$

$F \rightarrow F^* \mid a \mid b$

b) Construct SLR parsing table for the above grammar.

a) See Topic: TOP-DOWN PARSING, Short Answer Type Question No. 1.

b) See Topic: BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY, Long Answer Type Question No. 2.

9. a) Describe two situations where operator precedence parser can discover errors.

b) Prove that the following grammar is ambiguous:

$S \rightarrow AB$

$A \rightarrow aa \mid a$

$B \rightarrow ab \mid b$

c) What is shift reduce parsing? What is handle? Give examples of handle. Show an illustration of shift reduce parsing for a suitable grammar and for each reduction indicate the corresponding handle.

d) What is LEX? Write a short note on LEX.

a) See Topic: OPERATOR PRECEDENCE PARSING, Short Answer Type Question No. 2.

b) See Topic: PARSING AND CONTEXT FREE GRAMMAR, Short Answer Type Question No. 2.

c) See Topic: BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY, Long Answer Type Question No. 3.

d) **See Topic: LEXICAL ANALYSIS, Long Answer Type Question No. 2(a).**

10. a) Distinguish between quadruples, triples and indirect triples for the expression
a = b * – c + b * – c

b) Translate the arithmetic expression
a * – (b + c / d) into
i) Syntax tree
ii) Post-fix notation
iii) 3 – address code.

c) Generate machine code for the following instruction:
X = a / – (b * c) – d.
Assume 3 registers are available.

a), b) & c) See Topic: INTERMEDIATE CODE GENERATION, Long Answer Type Question No. 1.

b) i) See Topic: SYNTAX DIRECTED TRANSLATION, Short Answer Type Question No. 2.

11. Write short notes on the following:
a) YACC
b) Symbol table management
c) Peephole optimization
d) Backpatching
e) Thompson's construction rule

a) See Topic: LEXICAL ANALYSIS, Long Answer Type Question No. 2(b).

b) See Topic: INTRODUCTION TO COMPILER, Long Answer Type Question No. 3(a).

c) See Topic: CODE OPTIMIZATION, Long Answer Type Question No. 3(b).

d) See Topic: INTERMEDIATE CODE GENERATION, Long Answer Type Question No. 5.

e) See Topic: LEXICAL ANALYSIS, Long Answer Type Question No. 2(c).

QUESTION 2008

Group – A
(Multiple Choice Type Questions)

1. Choose the correct alternatives for the following:

i) Which data structure is mainly used during shift-reduce parsing?
a) Stack b) Queue ✓c) Array d) Pointer

ii) If all productions in a grammar G = (V, T, S, P) are of the form A → xB or A → x A, B ∈ V and x ∈ T*, then it is called
a) contex-sensitive grammar b) non-linear grammar
✓c) right-linear grammar d) left-linear grammar

iii) The edges in a flow graph whose heads dominate their tails are called
✓a) Back edges b) Front edges c) Flow edges d) None of these

iv) The regular expression 0* (10*)* denotes the same set as

a) (1 * 0) * 1 * ✓b) 0 + (0 + 10)* c) (0 + 1) * 10 (0 + 1)* d) none of these

v) If x is a terminal, then FIRST (x) is

a) ε ✓b) {x} c) x^* d) none of these

vi) The role of the preprocessor is

a) produce output data b) produce output to compilers

✓c) produce input to compilers d) none of these

vii) Which of the following is not true about dynamic type checking?

a) It increases the cost of execution b) Type checking is done during the execution

c) All the type errors are detected ✓d) None of these

viii) A dangling reference is a

✓a) pointer pointing to storage which is still in use

b) pointer pointing to storage which is freed

c) pointer pointing to nothing

d) pointer pointing to uninitialized storage

ix) Which of the following is not a loop optimization?

a) Loop unrolling b) Loop jamming

✓c) Loop heading d) Induction variable elimination

x) If a grammar is LALR (1) then it is necessarily

a) LL (1) b) SLR (1) c) LR (1) ✓d) None of these

Group – B
(Short Answer Type Questions)

2. Consider the context-free grammar:

$S \rightarrow SS + | SS^* | a$

a) Show how the string aa+a* can be generated by this grammar.

b) Construct a parse tree for this string.

c) What language is generated by this grammar?

See Topic: PARSING AND CONTEXT FREE GRAMMAR, Short Answer Type Question No. 3.

3. Consider the following left-linear grammar:

$S \rightarrow Sab | Aa$

$A \rightarrow Abb | bb$

Find out an equivalent right-linear grammar.

See Topic: PARSING AND CONTEXT FREE GRAMMAR, Short Answer Type Question No. 4.

4. Translate the arithmetic expression a* – (b + c) into
a) Syntax tree
b) Three-address code
c) Postfix notation.

a) See Topic: SYNTAX DIRECTED TRANSLATION, Short Answer Type Question No. 3.

b) & c) See Topic: INTERMEDIATE CODE GENERATION, Short Answer Type Question No. 4.

5. Give the NFA for the following Regular Expression. Then find a DFA for the same language.
(a|b) * abb

See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 5.

6. What is handle?
Consider the grammar E → E + E|E * E|id
Find the handles of the right sentential forms of reduction for the string id + id * id.

See Topic: PARSING AND CONTEXT FREE GRAMMAR, Short Answer Type Question No. 1.

Group – C
(Long Answer Type Questions)

7. Explain the following terms with examples:
a) Quadruples
b) Triples
c) Indirected triples
ex : a : = – b * (c + d / b) – (e * f)

See Topic: INTERMEDIATE CODE GENERATION, Short Answer Type Question No. 5.

8. Design an LL (1) parsing table for the following grammar:
S → aAcd | BcAe
A → b | ε
B → C f | d
C → f e
With the help of the parsing table show how the string "fefcbe" is parsed.

See Topic: TOP-DOWN PARSING, Long Answer Type Question No. 2.

9. a) Consider the following Grammar:
1) E → TE′
2) E′→ + TE′|ε
3) T→ FT′
4) T′→ * FT′|ε
5) F → (E)|id
i) Obtain the FIRST and FOLLOW sets for the above grammar.
ii) Construct a Predictive Parsing table for the above grammar.
b) Explain the predictive Parser's action by describing the moves it would make on an input id + id * id$.

See Topic: TOP-DOWN PARSING, Long Answer Type Question No. 3.

10. a) What is Peephole optimization?
b) What is an activation record? When and why are those records used? List different fields of an activation record and state the purpose of those fields.
c) Construct the DAG for the following basic block:

d: = b*c
e: = a + b
b: = b*c
a: = e – d

a) See Topic: CODE OPTIMIZATION, Long Answer Type Question No. 1(a).
b) See Topic: RUNTIME ENVIRONMENT, Long Answer Type Question No. 1.
c) See Topic: SYNTAX DIRECTED TRANSLATION, Short Answer Type Question No. 1.

11. a) What do you understand by terminal table and literal table?
b) Consider some interblock code optimization without any data-flow analysis by treating each extended basic block as if it is a basic block. Give algorithms to do the following optimization within an extended basic block. In each case, indicate what effect on other extended basic blocks a change within one extended basic block can have.

i) Common sub-expression elimination
ii) Constant folding
iii) Copy propagation

a) See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 3.
b) See Topic: CODE OPTIMIZATION, Long Answer Type Question No. 1(b).

12. Write short notes on the following:

a) Cross compiler
b) Code optimization
c) Left factoring
d) Context free grammar
e) Inherited attributes

a) See Topic: INTRODUCTION TO COMPILER, Long Answer Type Question No. 3(b).
b) See Topic: CODE OPTIMIZATION, Long Answer Type Question No. 3(c).
c) See Topic: PARSING AND CONTEXT FREE GRAMMAR, Long Answer Type Question No. 3(a).
d) See Topic: PARSING AND CONTEXT FREE GRAMMAR, Long Answer Type Question No. 3(b).
e) See Topic: PARSING AND CONTEXT FREE GRAMMAR, Long Answer Type Question No. 3(c).

QUESTION 2009

Group – A
(Multiple Choice Type Questions)

1. Choose the correct alternatives for the following:
i) The regular expression $(a \mid b)^* abb$ denotes

a) all possible combinations of *a*'s and *b*'s
b) set of all strings endings with *abb*

c) set of all strings starting with *a* and ending with *abb*
✓d) none of these

ii) An inherited attributes is the one whose initial value at a parse tree node is defined in ter
a) attributes at the parent and / or siblings of that node
b) attributes at children nodes only
✓c) attributes at both children nodes and parent and / or siblings of that node
d) none of these

iii) The intersection of a regular language and a context free language is
a) always a regular language
✓b) always a context free language
c) always a context sensitive language
d) none of these

iv) If *I* is a set of valid items for a viable prefix γ, then GOTO (*I* , *X*) is a set of items that are the viable prefix:
✓a) δ b) γ c) Prefix of γ d) None c

v) Shift-reduce parsers are
a) Top-down parsers
✓b) Bottom-up parsers
c) May be top-down or bottom-up parsers
d) None of these

vi) In a programming language, an identifier is permitted to be a letter followed by any nu letters or digits. If L and D denote the set of letters and digits respectively, which of the expressions defines an identifier?
a) $(L \cup D)^+$ ✓b) $L.(L \cup D)^*$ c) $(L.D)^*$ d) $L.(L.D)$

vii) The following productions of a regular grammar generates a language L.
S→aS | bS | a | b
The regular expression for L is
a) $a+b$ b) $(a+b)^*$ ✓c) $(a+b)(a+b)^*$ d) $(aa+b$

viii) Which of the following is not a loop optimization?
a) Induction variable elimination
b) Loop jamming
c) Loop unrolling
✓d) Loop heading

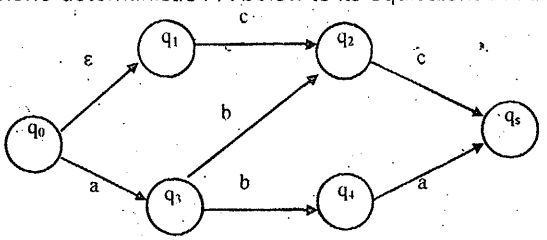
ix) Which of the following is not true about dynamic type checking?
a) It increases the cost of execution
b) Type checking is done during the execution
c) All the type errors are detected
✓d) None of these

x) An annotated parse tree is a parse tree
a) with values of only some attributes shown at parse tree nodes
✓b) with attribute values shown at the parse tree node

c) without attribute values shown at the parse tree nodes
d) with grammar symbols shown at the parse tree nodes

Group – B
(Short Answer Type Questions)

2. Convert the none-deterministic FA below to its equivalent DFA.



See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 6.

3. Consider the following lexically nested C code:
int a, b;
int foo( ) {int a, c;}
int bar( ) { int a, b; / * HERE*/}
a) How can symbol tables represent the state of each scope at the point marked HERE? Draw a diagram.
b) What symbols are visible / not visible at point HERE?
See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 7.

4. Consider the context-free grammar
$S \rightarrow SS + | SS^* | a$
a) Show how the string aa + a* can be generated by this grammar.
b) Construct a parse tree for this string.
c) What Language does this grammar generate? Justify your answer.
See Topic: PARSING AND CONTEXT FREE GRAMMAR, Short Answer Type Question No. 3.

5. a) How does Lexical Analyzer help in the process of compilation? Explain it with an example.
b) Consider the following conditional statement:
if (x > 3) then y = 5 else y = 10
From the above statement how many tokens are possible and what are those?
See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 8.

6. What is look ahead operator? Give an example. With the help of the look ahead concept show how identifiers can be distinguished from keywords.
See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 4.

Group – C
(Long Answer Type Questions)

7. a) Explain the different phases of a compiler, showing the output of each phase, using the example of the following statement:

Position: = initial + rate * 60

b) Compare compiler and interpreter.

See Topic: INTRODUCTION TO COMPILER, Long Answer Type Question No. 4.

8. a) Construct SLR parsing table for following grammar:

$S \rightarrow AS \mid b$

$A \rightarrow SA \mid a$

b) What is an operator grammar? Give an example.

a) See Topic: BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY, Long Answer Type Question No. 4.

b) See Topic: OPERATOR PRECEDENCE PARSING, Short Answer Type Question No. 1.

9. a) Translate the following expression:

$a = b^* - c + b^* - c$

into

i) Quadruples

ii) Triples

iii) Indirect Triples.

b) What are the differences among Quadruples, Triples and Indirect Triples?

c) Generate machine code for the followings instruction:

$v = a + (b^* c) - d$.

a) See Topic: INTERMEDIATE CODE GENERATION, Long Answer Type Question No. 1(a).

b) & c) See Topic: INTERMEDIATE CODE GENERATION, Long Answer Type Question No. 2.

10. a) Construct the DAG for the following basic block:

d : = b* c

e : = a + b

b : = b * c

a : = e – d

b) What is peephole optimization?

c) Consider some interblock code optimization without any data flow analysis by treating each extended basic block as if it is a basic block. Give algorithms to do the following optimizations within an extended basic block. In each case, indicate what effect on other extended basic blocks a change within one extended basic block can have.

i) Common sub-expression elimination

ii) Constant folding

iii) Copy propagation.

a) See Topic: SYNTAX DIRECTED TRANSLATION, Short Answer Type Question No. 1.

b) & c) See Topic: CODE OPTIMIZATION, Long Answer Type Question No. 1.

11. Write short notes on the following:
    a) Loop optimization
    b) Dependency graph
    c) Input buffering
    d) YACC
    e) Symbol Table
    f) L-attributed definitions
    g) LEX

a) See Topic: CODE OPTIMIZATION, Long Answer Type Question No. 3(d).
b) See Topic: SYNTAX DIRECTED TRANSLATION, Long Answer Type Question No. 3(a).
c) See Topic: SYNTAX DIRECTED TRANSLATION, Long Answer Type Question No. 3(b).
d) See Topic: LEXICAL ANALYSIS, Long Answer Type Question No. 2(b).
e) See Topic: INTRODUCTION TO COMPILER, Long Answer Type Question No. 3(a).
f) See Topic: SYNTAX DIRECTED TRANSLATION, Long Answer Type Question No. 3(c).
g) See Topic: LEXICAL ANALYSIS, Long Answer Type Question No. 2(a).

QUESTION 2010

Group – A
(Multiple Choice Type Questions)

1. Choose the correct alternatives for the following:

i) Symbol table can be used for
    a) checking type compatibility
    b) suppressing duplicate error messages
    c) storage allocation
    ✓d) all of these

ii) Which data structure is mainly used during shift-reduce parsing?
    a) Pointers b) Arrays ✓c) Stacks d) Queues

iii) Which of the following is not an intermediate code form?
    a) Postfix notation b) Syntax trees ✓c) Three address codes d) Quadruples

iv) If x is a terminal then FIRST (x) is
    a) ε ✓b) { x } c) x d) none of these

v) Which one of the following error will not be detected by the compiler?
    a) Lexical error b) Syntactic error c) Semantic error ✓d) Logical error

vi) The grammar E $\rightarrow$E+E$\left|E^{*}E\right|\alpha$ is
    ✓a) ambiguous
    b) unambiguous
    c) not given sufficient information
    d) none of these

vii) YACC builds up
a) SLR parsing table ✓b) LALR parsing table
c) canonical LR parsing table d) none of these

viii) If a grammar is in LALR (1) then it is necessarily
a) LL (1) b) SLR (1) c) LR (1) ✓d) none of these

ix) Which one of the following is not true about dynamic checking?
a) It increases the cost of execution b) Type checking is done during execution
c) All the type error are detected ✓d) None of these

x) A basic block can be analyzed by
✓a) DAG b) Flow graph c) Graph with cycles d) None of these

xi) The method which merges the bodies of two loops is
a) loop unrolling ✓b) loop ramming c) constant folding d) none of these

xii) A top down parser generates
✓a) leftmost – derivation b) rightmost – derivation
c) leftmost derivation in reverse d) rightmost derivation in reverse

Group – B
(Short Answer Type Questions)

2. How the following statement is translated via the different phases of compilation?
Position : = initial + rate *70.
See Topic: INTRODUCTION TO COMPILER, Long Answer Type Question No. 1(a).

3. Convert the following NFA into its equivalent DFA:
The set of all strings with 0 and 1, beginning with 1 & ending with 00.
See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 9.

4. Explain inherited attribute and synthesized attribute for Syntax directed translation with suitable example.
See Topic: SYNTAX DIRECTED TRANSLATION, Short Answer Type Question No. 4.

5. What is type checking? Differentiate between Dynamic and Static Type checking.
See Topic: TYPE CHECKING, Short Answer Type Question No. 2.

6. Differentiate Quadruple, Triples and Indirect triples with example.
See Topic: INTERMEDIATE CODE GENERATION, Long Answer Type Question No. 2(a).

Group – C
(Long Answer Type Questions)

7. a) What are the analysis phase and synthesis phase of an assemble?

b) Suppose a robot can be instructed to move one step east, north, west or south from its current position. A sequence of such instruction is generated by the following grammar:

Seq → Seq_1 instr | begin

Instr → east | north | west | south

i) Construct a syntax directed definition to translate an instruction sequence into a robot position.

ii) Draw a parse tree for: begin west south.

a) See Topic: INTRODUCTION TO COMPILER, Short Answer Type Question No. 2.

b) See Topic: SYNTAX DIRECTED TRANSLATION, Long Answer Type Question No. 1.

8. Construct a predictive parsing table for the grammar:

S → iEtSS′ | *a*

S′ → eS | ε

E → b

Here S is star symbol & S′ are non-terminals & i, t, a, e, b are terminals.

Explain the steps in brief.

See Topic: PARSING AND CONTEXT FREE GRAMMAR, Long Answer Type Question No. 1.

9. Construct DFA directly from [not by generating NFA] the regular expression

L = (a | b)* ab

What are the main contributions of Syntax Directed Translation in Compiler? Design a Dependency Graph and Direct acyclic graph for the string

a+ a*(b – c)+ (b – c) * d

1st Part: See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 10.

2nd Part: SYNTAX DIRECTED TRANSLATION, Long Answer Type Question No. 2.

10. Translate the expression a = – (a + d) * (c+ d) + (a + b + c) into

a) Quardruple

b) Triple

c) Indirect Triple

Draw the flow graph for the following code:

i) location = - 1

ii) i = 0

iii) i < 100 goto 5

iv) goto 13

v) $t_1 = 4i$

vi) $t_2 = A [t_1]$

vii) if $t_2 = x$ goto 9

viii) goto 10

ix) location = i

x) $t_3 = i + 1$

xi) $i = t_3$
xii) goto 3
xiii).....

What do you understand by terminal table and literal table?

1st Part: See Topic: INTERMEDIATE CODE GENERATION, Long Answer Type Question No. 3.
2nd Part: See Topic: CODE OPTIMIZATION, Long Answer Type Question No. 2.
3rd Part: See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 3.

11. Write short notes on the following:
a) LEX and YAAC
b) Activation record
c) Symbol Table
d) Peephole optimization
e) Cross compiler.

a) See Topic: LEXICAL ANALYSIS, Long Answer Type Question No. 2(d).
b) See Topic: RUNTIME ENVIRONMENT, Long Answer Type Question No. 2.
c) See Topic: INTRODUCTION TO COMPILER, Long Answer Type Question No. 3(a).
d) See Topic: CODE OPTIMIZATION, Long Answer Type Question No. 3(b).
e) See Topic: INTRODUCTION TO COMPILER, Long Answer Type Question No. 3(b).

QUESTION 2011

Group – A
(Multiple Choice Type Questions)

1. Choose the correct alternatives for the following:

i) First pos of a.(dot) node with leaves c1 and c2 is

a) $firstpos(c1) \cup firstpos(c2)$

b) $firstpos(c1) \cap firstpos(c2)$

✓c) $if\,(nullable(c1))\,firstpos(c1) \cup firstpos(c2)\,else\,firstpos(c1)$

d) $if\,(nullable(c2))\,firstpos(c1) \cup firstpos(c2)\,else\,firstpos(c1)$

ii) Parse tree is generated in the phase of

✓a) Syntax Analysis
b) Semantic Analysis
c) Code Optimization
d) Intermediate Code Generation

iii) $FIRST(\alpha\beta)$ is

a) $FIRST(\alpha)$

b) $FIRST(\alpha) \cup FIRST(\beta)$

✓ c) $FIRST(\alpha) \cup FIRST(\beta)\,if\,FIRST(\alpha)$ contains ε else $FIRST(\alpha)$

d) none of these

iv) A given grammar is not LL (1) if the parsing table of a grammar may contain
a) any blank filed
b) any e-entry
c) duplicate entry of same production
✓d) more than one production rule

v) White spaces and tabs are removed in
✓a) Lexical analysis
b) Syntax analysis
c) Semantic analysis
d) all of these

vi) Left factoring guarantees
✓a) not occurring of backtracking
b) cycle free parse tree
c) error free target code
d) correct LL(1) parsing table

vii) A parse tree showing the values of attributes at each node is called in particular
a) Syntax tree
✓b) Annotated parse tree
c) Syntax Direct parse tree
d) Direct Acyclic graph

viii) Which of the following is not true for Dynamic Type Checking
a) It increases the cost of execution
b) Type checking is done during the execution
c) All the type errors are detected
✓d) None of these

ix) Which of the following is not a loop optimization?
a) Induction variable elimination
b) Loop jamming
c) Loop unrolling
✓d) Loop heading

x) YACC builds up
a) SLR parsing table
✓b) LALR parsing table
c) canonical LR parsing table
d) none of these

Group – B
(Short Answer Type Questions)

2. Describe analysis phase of a compiler in respect of the following example.
Position = initial + rate * 60
See Topic: INTRODUCTION TO COMPILER, Short Answer Type Question No. 3.

3. Describe with diagram the working process of lexical Analyzer.
See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 11.

4. What is error handling? Describe the Panic Mode and Phrase level error recovery technique with example.
See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 12.

5. What do you understand by L-attributed definitions? Illustrate with an example.

See Topic: SYNTAX DIRECTED TRANSLATION, Short Answer Type Question No. 5.

6. What is recursive descent parsing? Describe the drawbacks of recursive descent parsing for generating the string '*abc*' from the grammar:

$S \rightarrow aBc$

$B \rightarrow bc \mid b$

See Topic: TOP-DOWN PARSING, Short Answer Type Question No. 2.

Group – C
(Long Answer Type Questions)

7. a) Describe with a block diagram the parsing technique of LL(1) parser. b) Parser the string 'abba' using LL(1) parser where the parsing table is given below:

	a	*b*	$
S	$S \rightarrow aBa$		
B	$B \rightarrow \varepsilon$	$B \rightarrow bB$	

c) Check whether the following grammar is LL(1) or not.

$S \rightarrow iCtSE \mid a$

$E \rightarrow eS \mid \varepsilon$

$C \rightarrow b$

See Topic: TOP-DOWN PARSING, Long Answer Type Question No. 4.

8. a) Describe LR parsing with block diagram.

b) What are the main advantages of LR parsing?

c) Construct SLR parsing table for the grammer given below:

$S \rightarrow Cb$

$C \rightarrow bC / d$

See Topic: BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY, Long Answer Type Question No. 5.

9. a) Construct DFA directly from [Not by generating NFA] the regular expression $L = (a \mid b)^* ab$

b) What are the main contributions of Syntax Directed Translation in Compiler? Design a Dependency Graph and Direct Acyclic Graph for the string: $a + a^*(b - c) + (b - c)^* d$.

a) See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 10.

b) See Topic: SYNTAX DIRECTED TRANSLATION, Long Answer Type Question No. 2.

10. a) Translate the expression $a = -(a + b)^*(c + d) + (a + b + c)$ into

i) Quadruple

ii) Triple

iii) Indirect Triple

iv) 3-address code.

b) Draw the flow graph for the following code:

i) location = –1

ii) $i = 0$

iii) $i < 100$ goto 5

iv) goto 13

v) $t_1 = 4i$

vi) $t_2 = A[t_1]$

vii) if $t_2 = x$ goto 9

viii) goto 10

ix) location $\doteq i$

x) $t_3 = i + 1$

xi) $i = t_3$

xii) goto 3

xiii) .....

c) What do you understand by terminal table and literal table?

a) See Topic: INTERMEDIATE CODE GENERATION, Long Answer Type Question No. 3.

b) See Topic: CODE OPTIMIZATION, Long Answer Type Question No. 2.

c) See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 3.

11. Write short notes on the following:

a) LEX and YAAC

b) Activation Record

c) Symbol Table

d) Peephole optimization

e) Input Buffering

a) See Topic: LEXICAL ANALYSIS, Long Answer Type Question No. 2(d).

b) See Topic: RUNTIME ENVIRONMENT, Long Answer Type Question No. 2.

c) See Topic: INTRODUCTION TO COMPILER, Long Answer Type Question No. 3(a).

d) See Topic: CODE OPTIMIZATION, Long Answer Type Question No. 3(b).

e) See Topic: SYNTAX DIRECTED TRANSLATION, Long Answer Type Question No. 3(b).

QUESTION 2012

Group – A
(Multiple Choice Type Questions)

1. Choose the correct alternatives for the following:

i) Which data structure is mainly used during Shift-Reduce parsing?

✓a) stack b) queue c) array d) pointer

ii) If x is a terminal, then FIRST (x) is

a) ε ✓b) $\{x\}$ c) $x*$ d) none of these

iii) Which of the following is not an intermediate code form?

a) Postfix Notation b) Syntax Trees
✓c) Three-Address Codes d) Quadruples

iv) Which one of the following errors will not be detected by the compiler?

a) Lexical error b) Syntactic error
c) Semantic error ✓d) Logical error

v) A basic block can be analyzed by

✓a) DAG b) flow graph
c) graph with cycles d) none of these

vi) A Top down parser generates

✓a) left-most derivation b) right-most derivation
c) left-most derivation in reverse d) right-most derivation in reverse

vii) YACC builds up

a) SLR parsing table ✓b) LALR parsing table
c) canonical LR parsing table d) none of these

viii) If the attributes of the parent node depends on its children, then its attributes are called

a) TAC b) synthesized
✓c) inherited d) directed

ix) Which is used to keep track of currently active activations?

✓a) control stack b) activation c) execution d) symbol

x) Optimization(s) connected with $x:=x+0$ is/are

a) peephole and algebraic b) reduction in strength and algebraic
✓c) peephole only d) loop and peephole

Group – B
(Short Answer Type Questions)

2. What is a cross-compiler? Create a cross-compiler for SML (Sensor Mark-up Language) using Java compiler, written in ATOM-450, producing code in ATOM-450 and a SML language producing code for XML written in Java.

See Topic: INTRODUCTION TO COMPILER, Short Answer Type Question No. 4.

3. Define regular expression. Write the regular expression over alphabet $\{a,b,c\}$ containing at least one 'a' and at least one 'b'. What is dead state? Explain with suitable example.

See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 13.

4. Define grammar. What do you mean by ambiguity in grammar? Illustrate with suitable example. What is the necessity to generate parse tree?

See Topic: SYNTAX DIRECTED TRANSLATION, Short Answer Type Question No. 6.

5. Distinguish between interpreter and compiler. How does lexical analyzer help in the process of compilation? Consider the following statement and find the number of tokens with type and value as applicable:

```
void main ( )
    {
        int x;
        x = 3;
    }
```

1st Part: See Topic: INTRODUCTION TO COMPILER, Long Answer Type Question No. 1(b).
Next Part: See Topic: LEXICAL ANALYSIS, Short Answer Type Question No. 14.

6. What is activation record? Explain clearly the components of an activation record.

See Topic: RUNTIME ENVIRONMENT, Short Answer Type Question No. 1.

Group – C
(Long Answer Type Questions)

7. a) Apply all the phases of compiler and show the corresponding output in every phase for the following code of the source program:

```
while (y ≥ t) y = y – 3;
```

b) What do you mean by passes of compiler? Explain advantages and disadvantages of one-pass and two-pass over each other.

See Topic: INTRODUCTION TO COMPILER, Long Answer Type Question No. 2.

8. a) Define LL(1) grammar. Consider the following grammar:

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

Test whether the grammar is LL(1) or not and construct a predictive parsing table for it.

b) Consider the following Context Free Grammar (CFG) G and reduce the grammar by removing all unit productions. Show each step of removal.

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow C|b$

$C \rightarrow D$

$D \rightarrow E$

$E \rightarrow a$

c) Consider the following grammar G. Show that the grammar is ambiguous by constructing two different leftmost derivations for the sentence '$abab$'.

$S \rightarrow aSbS|bSaS|\varepsilon$

See Topic: **PARSING AND CONTEXT FREE GRAMMAR, Long Answer Type Question No. 2.**

9. a) Consider the following grammar G. Alternate the production so that it may free from backtracking.

Statement → if Expression then Statement else Statement

Statement → if Expression then Statement

b) What is left-recursion? Illustrate with suitable example. Consider the following grammar G. Find out the left recursion and remove it:

$S \rightarrow Bb|a$

$B \rightarrow Bc|Sd|e$

c) What is Operator Precedence Parsing? Discuss about the advantage and disadvantage of Operator Precedence Parsing. Consider the following grammar:

$E \rightarrow TA$

$A \rightarrow +TA|\varepsilon$

$T \rightarrow FB$

$B \rightarrow *FB|\varepsilon$

$F \rightarrow id$

Test whether this grammar is Operator Precedence Grammar or not and slow how the string $w=id+id*id+id$ will be processed by this grammar.

a) See Topic: **PARSING AND CONTEXT FREE GRAMMAR, Short Answer Type Question No. 6.**
b) & c) See Topic: **OPERATOR PRECEDENCE PARSING, Long Answer Type Question No. 1.**

10. a) Distinguish between quadruples, triples and indirect triples for the expression.

$x=y*-z+y*-z$

b) Translate the expression $a*-(b+c/d)$ into

i) Syntax tree.

ii) Post-fix notation

iii) 3-address code.

c) While the three-address code for the following C program:

```
main ( )
  {
      int x = 1;
      int y[20];
      while (x ≤ 20)
       a[x] = 0;
  }
```

a) & c) See Topic: INTERMEDIATE CODE GENERATION, Long Answer Type Question No. 4.

b) See Topic: INTERMEDIATE CODE GENERATION, Long Answer Type Question No. 1(b).

11. Write short notes on the following:
    a) Chomsky classification of grammar
    b) Symbol table
    c) LEX
    d) YACC
    e) Handle pruning

a) See Topic: INTRODUCTION TO COMPILER, Long Answer Type Question No. 3(c).

b) See Topic: INTRODUCTION TO COMPILER, Long Answer Type Question No. 3(a).

c) See Topic: LEXICAL ANALYSIS, Long Answer Type Question No. 2(a).

d) See Topic: LEXICAL ANALYSIS, Long Answer Type Question No. 2(b).

e) See Topic: BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY, Long Answer Type Question No. 6.

COMPILER DESIGN

SOLUTION 2013

Group – A

(Multiple Choice Type Questions)

1. Choose the correct alternatives for the following:

i) Which of the following is the most powerful parser?

a) SLR

b) LALR

c) Canonical LR

d) Operator precedence.

Chapter Name: "BOTTOM-UP PARSING"

Answer: (b)

ii) Inherited attribute is a natural choice in

a) keeping track of variable declaration

b) checking for the correct use of L values and R values

c) both (a) and (b)

d) none of these

Chapter Name: "SYNTAX DIRECTED TRANSLATION"

Answer: (c)

iii) A top down parser generates

a) rightmost derivation

b) rightmost derivation in reverse

c) left most derivation

d) left most derivation in reverse.

Chapter Name: "TOP-DOWN PARSING"

Answer: (c)

iv) In operator precedence parsing, precedence relations are defined

a) for all pair of non-terminals

b) for all pair of terminals

c) to delimit the handle

d) only for a certain pair of terminals.

Chapter Name: "OPERATOR PRECEDENCE PARSING"

Answer: (d)

v) A parser with the valid prefix property is advantageous because it

a) detect errors as soon as possible

b) detect errors as and when they occur

c) limits the amount of erroneous output passed to the next phases

d) all of these phases.

Chapter Name: "BOTTOM UP PARSING"

Answer: (c)

vi) Which of the following is used for grouping of characters into tokens?

a) Parser

b) Code optimization

c) Code generator

d) Lexical analyzer

Chapter Name: "LEXICAL ANALYSIS"

Answer: (d)

vii) Three-address code involves

a) exactly 3 addresses

b) at most 3 addresses

c) no unary operator

d) none of these.

Chapter Name: "INTERMEDIATE CODE GENERATION"

Answer: (b)

viii) ..... or scanning is the process where the stream of characters making up the source program is read from left to right grouped into tokens.

a) Lexical analysis

b) Diversion

c) Modeling

d) None of these.

Chapter Name: "LEXICAL ANALYSIS"

Answer: (a)

ix) The graph that shows basic blocks and their successor relationship is called

a) DAG

b) Flow chart

c) Control graph

d) Hamiltonian graph

Chapter Name: "CODE OPTIMIZATION"

Answer: (b)

x) A compiler that runs on one machine and produces code for a different machine is called

a) Cross compilation

b) One pass compilation

c) 2 pass compilation

d) none of these.

Chapter Name: "INTRODUCTION"

Answer: (a)

Group – B

(Short Answer Type Questions)

2. Explain different stages of compiler with a suitable example. What is Token, Patter and Lexeme?

Answer:

First part: ***Refer to "INTRODUCTION TO COMPILER", Long Answer Type Question (1)***

Second part: Chapter Name: "INTRODUCTION TO COMPILER"

Lexeme: group of characters that form a token

Token: class of lexemes that match a pattern

• Token may have attributes, if more than one lexeme in token

Pattern: typically defined using a regular expression

• REs are simplest language class that's powerful enough

3. Explain left factoring with suitable example.

Answer:

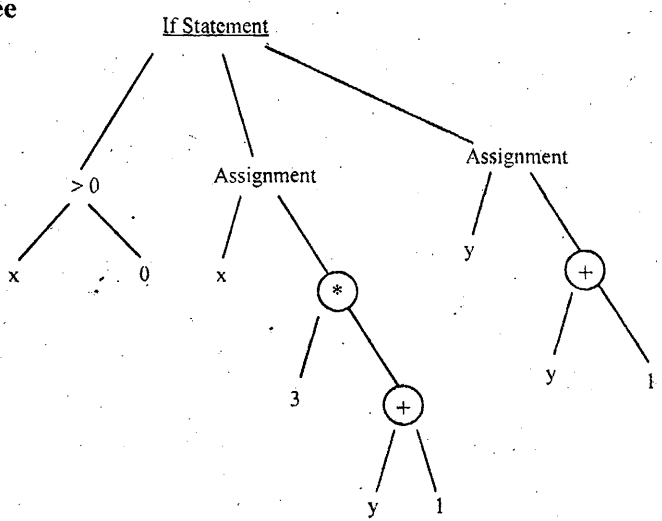
Refer to "PARSING AND CONTEXT FREE GRAMMAR", Long Answer Type Question 4.(a)

4. Draw a Syntax tree and DAG from the following expression. If $x > 0$ **then** $x = 3*(y+1)$ else $y = y+1$

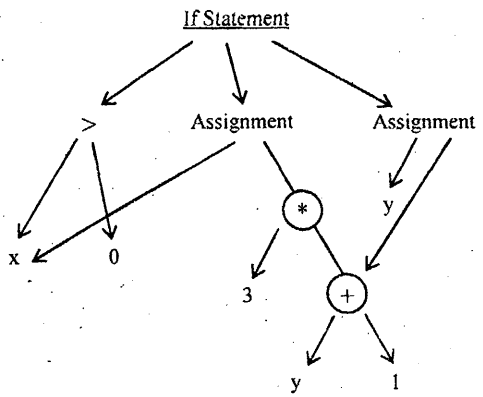
Chapter Name: "SYNTAX DIRECTED TRANSLATION"

Answer:

Syntax Tree



DAG



5. Compare quadruples, triples and indirect triples.

$x = (a+b) * -e/d$

Represent this expression in quadruples, triples and indirect triples form.

Chapter Name: "INTERMEDIATE CODE GENERATION"

Answer:

First Part:

Three address code instructions can be implemented as objects or as records with fields for the operator and the operands. Three such representations are called

Quadruples: A quadruple (or just "quad') has four fields, which we call op, arg,, arg2, and result

Triples: A triple has only three fields, which we call op, arg1, and arg2. the DAG and triple representations of expressions are equivalent

Indirect Triples: consist of a listing of pointers to triples, rather than a listing of triples themselves.

The benefit of Quadruples over Triples can be seen in an optimizing compiler, where instructions are often moved around.

With quadruples, if we move an instruction that computes a temporary t, then the instructions that use t require no change. With triples, the result of an operation is referred to by its position, so moving an instruction may require to change all references to that result. This problem does not occur with indirect triples.

Second Part:

Quadruples

	op	res	Arg1	Arg2
(101)	T1	+	a	b
(102)	T2	/	c	d
(103)	T3	uniminus	T2	
(104)	T4	*	T1	T3

Triples

	op	Arg1	Arg2
(101)	+	a	b
(102)	/	c	d
(103)	uniminus	(102)	
(104)	*	(101)	(103)

Indirect triples

	op	Arg1	Arg2
(101)	+	a	b
(102)	/	c	d
(103)	uniminus	(102)	
(104)	*	(101)	(103)

	Statement
(1)	(101)
(2)	(102)
(3)	(103)
(4)	(104)

6. Convert the regular expression $(a+b)^*abb$ into e-NFA and then corresponding DFA.

Answer:

Refer to "LEXICAL ANALYSIS", Short Answer Type Question no. 5

Group – C

(Long Answer Type Questions)

7. a) What is Basic Block? List out the basic blocks and draw the flow graph for the following code:

1. location = –1
2. i = 0
3. i < 100 goto 5
4. goto 13
5. $t_1 = 4i$
6. $t_2 = A[t_1]$
7. if t_2 = x goto 9
8. goto 10
9. location = i
10. $t_3 = i + 1$
11. $i = t_3$
12. goto 3
13. ..

b) What do you mean by a Handle? Give example.

When a grammar is called ambiguous? Is there any technique to remove ambiguity?

Explain with an example. What is the reduce-reduce conflict in LR parser? What are the various data structures used for symbol table construction?

Answer:

a) *Refer to "CODE OPTIMIZATION", Long Answer Type Question No. 2*

b) First Part:

Refer to "PARSING AND CONTEXT FREE GRAMMAR", short Answer Type Question No. 1

Second part: Chapter Name: "SYNTAX DIRECTED TRANSLATION"

A grammar that produces more than one parse tree for some sentence is said to be ambiguous.

An ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for the same sentence.

There are several ways to remove ambiguity:

(A) Fix grammar so it is not ambiguous. (Not always possible or reasonable or possible.)

(B) Add grouping/precedence rules.

(C) Use semantics: choose parse that makes the most sense.

Grouping/precedence rules are the most common approach in programming language applications.

Invoking semantics is more common in natural language applications.

Fixing the grammar is less often useful in practice, but neat when you can do it.

Third Part: Chapter Name: "BOTTOM UP PARSING"

For any two complete items A $\rightarrow$ a. and B $\rightarrow$ b. in s, Follow(A) and Follow(B) are disjoint (their intersection is the empty set). Violation of this rule is a Reduce-Reduce Conflict.

Last Part: Chapter Name: "INTRODUCTION"

Linked List , Sorted and unsorted list , search tree, binary tree, hash table are the data structures used for symbol table.

8. Generate the three address code for the following code segment (show the semantic actions).

```
Main ( )
    {
    int a=1, z=0;
    int b[0];
    while (a<=10)
```

```
        b[a]=2*a;
        z=b[a]+a;
    }
```

Chapter Name: "INTERMEDIATE CODE GENERATION"

Answer:

100: if (a<=10) goto 107

101: goto 105

102: T_1 = addr (b)

103: T_2 = 2 * a.

104: $T_1[20]=T_2$

105: z = $T_1[20]$ + a

106: goto 100

107:

9. Consider the following grammar:

S → aABb

A → c | ε

B → d | ε

Prove the above grammar is LL (1).

Draw the parsing table.

Now check whether the string "ab" and "acdb" are the languages of the above grammar.

(Derive each step with the help of a stack.)

Chapter Name: "TOP DOWN PARSING"

Answer:

S → aABb

A → c | ε

B → d | ε

FIRST(S) = FIRST(aABb) = { a }

FIRST(A) = FIRST(c) U FIRST(ε) = { c, ε }

FIRST(B) = FIRST(d) U FIRST(ε) = { d, ε }

Since the right-end marker $ is used to mark the bottom of the stack, $ will initially be immediately below S (the start symbol) on the stack; and hence, $ will be in the FOLLOW(S). Therefore:

$FOLLOW(S) = \{ \$ \}$

Using S → aABb, we get:

$FOLLOW(A) = FIRST(Bb)$
$= FIRST(B) - \{ \in \} \cup FIRST(b)$
$= \{ d, \in \} - \{ \in \} \cup \{ b \} = \{ d, b \}$
$FOLLOW(B) = FIRST(b) = \{ b \}$

Therefore, the parsing table is as shown in Table below.

Table : Production Selections for Parsing Derivations

	a	*b*	*c*	*d*	$
S	$S \rightarrow aABb$				
A		$A \rightarrow \in$	$A \rightarrow c$	$A \rightarrow \in$	
B		$B \rightarrow \in$		$B \rightarrow d$	

As there is only single production in each entry so the grammar is LL(1).

Consider an input string *acdb*. The various steps in the parsing of this string, in terms of the contents of the stack and unspent input, are shown in Table below.

Stack Contents	Unspent Input	Moves
$S	acdb$	Derivation using S → aABb
$bBAa	acdb$	Popping a off the stack and advancing one position in the input
$bBA	cdb$	Derivation using A → c
$bBc	cdb$	Popping c off the stack and advancing one position in the input
$bB	db$	Derivation using B → d
$bd	db$	Popping d off the stack and advancing one position in the input
$b	b$	Popping b off the stack and advancing one position in the input
	$	Announce successful completion of the parsing

Similarly, for the input string "ab", the various steps in the parsing of the string, in terms of the contents of the stack and unspent input, are shown in Table below.

Stack Contents	Unspent Input	Moves
$S	ab$	Derivation using S → aABb
$bBAa	ab$	Popping a off the stack and advancing one position in the input
$bBA	b$	Derivation using A →ε
$bB	b$	Derivation using B → ε
$b	b$	Popping b off the stack and advancing one position in the input
$	$	Announce successful completion of the parsing

10. Consider the following grammar:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id$

Draw a SLR state transition diagram for the above grammar. Also draw the SLR parse table.

Chapter Name: "BOTTOM UP PARSING"

Answer:

The augmented grammar is:

$E' \rightarrow E$

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Now, $s_0 = \text{closure}(E' \rightarrow .E.)$

We start with $s_0 = \{E' \rightarrow .E\}$. Silence the 'dot' is in front of **E**, a non-terminal, items for all **E** productions with the dot at the beginning, will be appended with the s_0. So, s_0 becomes:

$s_0 = \{\mathbf{E'} \rightarrow .\mathbf{E}, \mathbf{E} \rightarrow .\mathbf{E}+\mathbf{T}, \mathbf{E} \rightarrow .\mathbf{T}\}$.

Again, the dot is front of a (new) non-terminal T and all T productions with the dot at the beginning, will be appended to s_0. So, s_0 becomes:

$s_0 = \{\mathbf{E'} \rightarrow .\mathbf{E}, \mathbf{E} \rightarrow .\mathbf{E}+\mathbf{T},\}$.

$E \rightarrow .T, T \rightarrow .T * F, T \rightarrow .F.$

Again, the dot is front of a (new) non-terminal F and all F productions with the dot at the beginning, will be appended to s_0. So, finally, s_0 becomes:

$s_0 = \{\mathbf{E'} \rightarrow .\mathbf{E}, \mathbf{E} \rightarrow .\mathbf{E}+\mathbf{T},.\}$

$E \rightarrow .T, T \rightarrow .T * F, T \rightarrow .F, F \rightarrow .id$

It is possible to have transitions from s_0 on **E, T, F, id** and because, these are the grammar symbols in the items in s_0, where the dot is just before it.

Consider:

$s_1 = \text{goto}(\text{s}_0, \mathbf{E})$.

This is simple. Simply let the dot cross **E** wherever the dot is in front of **E.**

This gives:

$s_1 = \{\mathbf{E'} \rightarrow \mathbf{E}., \mathbf{E} \rightarrow \mathbf{E}.+\mathbf{T}\}$.

Note that in both the items, the dot is in front of terminals. So, the "closure" operation will not add further items.

Similarly,

$s_2 = goto(s_0, \mathbf{T}) = \{\mathbf{E} \rightarrow \mathbf{T}., \mathbf{T} \rightarrow \mathbf{T}.*\mathbf{F}\}$

$s_3 = goto(s_0, \mathbf{F}) = \{\mathbf{T} \rightarrow \mathbf{F}.\}$

$s_4 = goto(s_0, \mathbf{id}) = \{\mathbf{E} \rightarrow \mathbf{id}.\}$

We can proceed similarly to get the following:

$s_5 = goto(s_1, +) = \{\mathbf{E} \rightarrow \mathbf{E}+.\mathbf{T}, \mathbf{T} \rightarrow .\mathbf{T}*\mathbf{F},\}$

$T \to .F, F \to .id.$

$s_6 = goto(s_2, *) = \{\mathbf{T \to T*.F, F \to .id}\}.$

$s_7 = goto(s_5, \mathbf{T}) = \{\mathbf{E \to E+T., T \to T.*F}\}.$

$s_8 = goto(s_6, \mathbf{F}) = \{\mathbf{T \to T*F.}\}.$

Additionally:

$goto(s_5, \mathbf{F}) = s_3.$

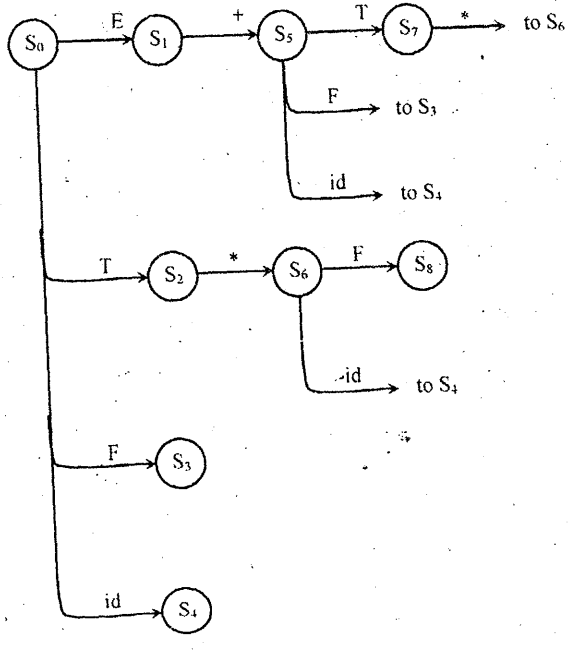
$goto(s_5, \mathbf{id}) = s_4.$

$goto(s_6, \mathbf{id}) = s_4.$

$goto(s_7, *) = s_6.$

You can clearly visualize the DFA and are encouraged to draw a diagram for the same (see Fig).

We have completed **Step-1** of parser table construction process.



The action/goto tables are extracted directly from the above information in the following manner. First, each state in the above table will be a row in the action/goto tables. Filling in the entries for row I is done as follows:

- Action[I,c] = shift to state goto(items in state I, c) for terminal c
- Goto[I,c] = goto(items in state I, c) for non-terminal c
- For production number n: A -> α in state I where '.' occurs at the end, Action[I,a] = reduce production n, for all elements a in Follow(A)
- For the added production(augmented production) with the . at the end in state I, Action[I,$] = accept.

The Parsing Table is as below:

State ← Action → GOTO

	id	+	*	$	E	T	F
0	S_4				1	2	3
1		S_5		acc			
2		R_2	S_6	R_2			
3		R_4	R_4	R_4			
4		R_5	R_5	R_5			
5	S_4					7	3
6	S_4						8
7		R_1	S_6	R_1			
8		R_3	R_3	R_3			

11. Write short notes on any three of the following:

a) LEX

Answer:

Refer to "LEXICAL ANALYSIS", Long Answer Type 4(a).

b) YACC

Answer:

Refer to "LEXICAL ANALYSIS", Long Answer Type 4(c).

c) Peephole optimization

Answer:

Refer to "CODE OPTIMIZATION", Long Answer Type 3(d).

d) Symbol Table

Answer:

Refer to "INTRODUCTION", Long Answer Type 4(a).

e) Cross compiler

Answer:

Refer to "INTRODUCTION", Long Answer Type 4(b).

SOLUTION 2014

Group – A

(Multiple Choice Type Questions)

1. Answer all questions:

(i) Which of the following translation program converts assembly language program to object program

(a) assembler (b) compiler (c) macro-processor (d) linker

Chapter Name: "INTRODUCTION TO COMPILER"

Answer: (a)

(ii) The grammer $E \rightarrow E + E | E * E | a$, is

(a) ambiguous

(b) unambiguous

(c) ambiguous or not depends on the given sentence

(d) none of these

Chapter Name: "PARSING AND CONTEXT FREE GRAMMAR"

Answer: (a)

(iii) Shift reduce parsers are

(a) top-down parsers (b) bottom – up parsers

(c) predictive parsers (d) none of these

Chapter Name: "BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY"

Answer: (b)

(iv) The peep-hole optimization

(a) is applied to a small part of the code

(b) can be used to optimize intermediate code

(c) can be applied to a portion of the code that is not contiguous

(d) all of these

Chapter Name: "CODE OPTIMIZATION"

Answer: (a)

(v) Which of the following uses only synthesized attributes?

(a) s-attributed grammar (b) l-attributed grammar

(c) inherited attribute (d) none of these

Chapter Name: "SYNTAX DIRECTED TRANSLATION"

Answer: (a)

(vi) YACC builds up

(a) SLR parsing table (b) LALR parsing table

(c) canonical LR parsing table (d) none of these

Chapter Name: "BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY"

Answer: (b)

(vii) Which data structure is mainly used during shift-reduce parsing?

(a) pointers (b) arrays (c) stacks (d) queues

Chapter Name: "BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY"

Answer: (c)

(viii) Which of the following expression has no 1-value?

(a) a(l+1) (b) a (c) 7 (d) *a

Chapter Name: "SYNTAX DIRECTED TRANSLATION"

Answer: (c)

(ix) If x is a terminal, the FIRST (x) IS

(a) ε (b) {x} (c) x * (d) none of these

Chapter Name: "TOP-DOWN PARSING"

Answer: (b)

(x) The expression wcw where w belongs to {a, b} * is

(a) regular (b) context free

(c) context sensitive (d) none of these

Chapter Name: "PARSING AND CONTEXT FREE GRAMMAR"

Answer: (b)

Group – B

(Short Answer Type Questions)

2. Using a block diagram indicate the phases of compilation explaining their activities.

Answer:

Refer to "INTRODUCTION TO COMPILER", Short Answer Type Question No. 1.

3. What is a handle? Describe various actions of a shift reduce parser.

Answer:

Refer to "BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY", Long Answer Type Question No. 3.

4. Eliminate left recursion from the following grammar:

$E \rightarrow E + T \mid T$

$T \rightarrow TF \mid F$

$F \rightarrow F * \mid a \mid b$

Answer:

Refer to "TOP-DOWN PARSING", Short Answer Type Question No. 1.

5. Explain Left Factoring with suitable example.

Answer:

Refer to "PARSING AND CONTEXT FREE GRAMMAR", Long Answer Type Question No. 3(a).

6. For the input expression (4*7+1) *2 construct an annotated parse tree.

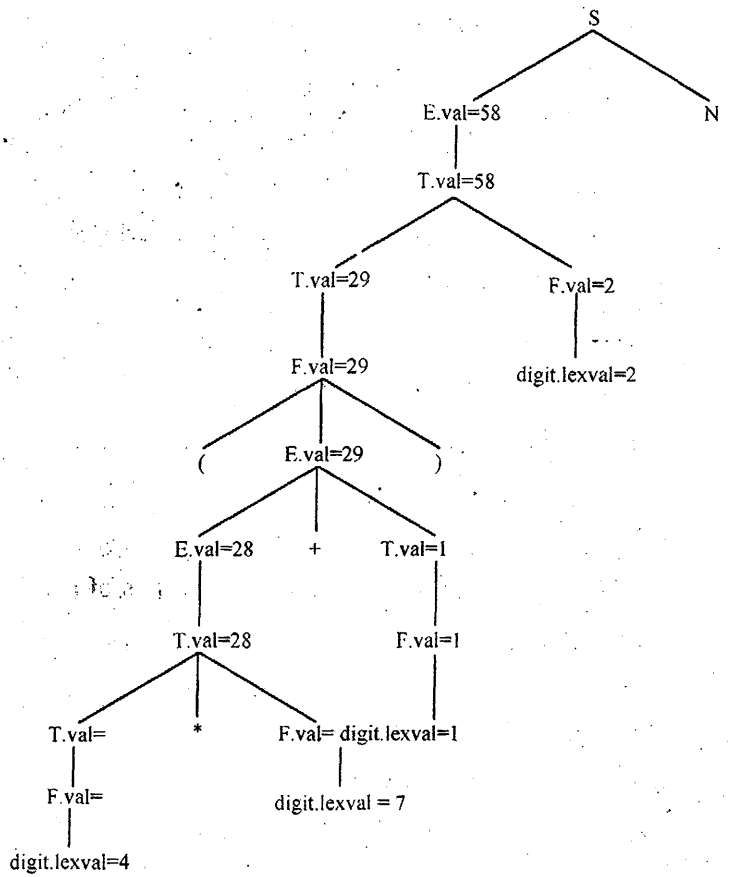
Chapter Name: "SYNTAX DIRECTED TRANSLATION"

Answer:

The syntax directed definition for evaluation of arithmetic expression is given as:

Production Rule	Semantic actions
$S \rightarrow EN$	Print (E.Val)
$E \rightarrow E_1 + T$	E.Val := E_1.Val + T.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
$T \rightarrow T_1 / F$	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
$N \rightarrow :$	-

The annoted parse tree can be drawn as follows:



Group – C

(Long Answer Type Questions)

7. (a) Consider the grammar:

$S \rightarrow aSbS \mid bSaS \mid E$

(i) Show that this grammar is ambiguous by constructing two different left most derivations for the sentence abab.

(ii) Construct the corresponding right most derivations for abab.

(iii) Construct the corresponding parse trees for abab.

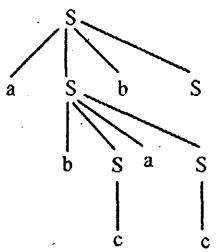
(iv) What language does this grammar generate?

Answer:

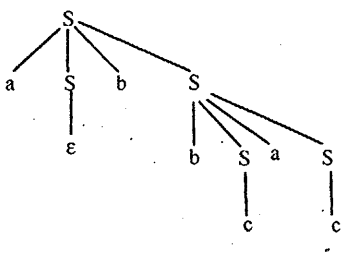
(i) ***Refer to Long Answer Type Question No. 2(c).***

(ii) S → aSbS → aSb → abSaSb → abSab → abab

(iii) Consider a string 'abab'. We can construct parse trees for deriving 'abab'.



Parse tree 1



Parse tree 2

(iv) This grammar generates the language which contains strings of equal number of a's and b's the empty string also.

(b) (i) Show that no left recursive grammar can be LL (1).

(ii) Show that no LL(1) grammar can be ambiguous.

Chapter Name: "PARSING AND CONTEXT FREE GRAMMAR"

Answer:

(i) The production is left-recursive if the leftmost symbol on the right side is the same as the non terminal on the left side. For example,

expr → expr + term.

Left-recursive grammars can never be LL(1), because the left-recursion will lead to an infinite loop.

Consider the grammar

$$A \rightarrow \beta | A\alpha$$

and try to parse the string $\beta\,\alpha$.

- If we removed the left recursion the grammar becomes

$$A \rightarrow \beta A'$$

$A' \rightarrow \alpha A' | \in$

which derives the same string but towards the right instead of the left

- Now parse $\alpha \alpha . \beta$ doesn't match α, therefore try another alternative for A. There are none, so parse fails. With right recursion, we will be matching part of the input string as we go along
- Left recursion indicates we are building the string from right-to-left
- To eliminate left recursion, we turn it into right recursion and build the string left-to-right

(ii) Any LL(1) grammar is unambiguous because by definition there is a at most one left most derivation for any string. LL(1) grammar cannot be ambiguous since our parsing algorithm for LL(1) grammars builds the only possible parse tree for a sentence in a deterministic way. (In general, an ambiguity in a grammar will manifest itself in the fact that there are two productions competing for the same cell in the parse table.)

8. (a) Translate the following expression:

$a = b * -c + b * -c$ **into**

(i) Quadruples

(ii) Triples

(iii) Indirect triples

(b) What are the differences among Quadruples, Triples and Indirect Triples?

(c) Generate machine code for the following instruction:

$V = a + (b * c) - d$

Answer:

a) ***Refer to "INTERMEDIATE CODE GENERATION", Long Answer Type Question No. 1(a).***

b) ***Refer to "INTERMEDIATE CODE GENERATION", Long Answer Type Question No. 2(a).***

c) *Refer to "INTERMEDIATE CODE GENERATION", Long Answer Type Question No. 2(b).*

9. (a) What do you mean by input buffering?

(b) How is input buffering implanted?

Answer:

a) & b) *Refer to "SYNTAX DIRECTED TRANSLATION", Long Answer Type Question No. 3(b).*

(c) What problem can arise implementing input buffering? Give a suitable example.

Chapter Name: "SYNTAX DIRECTED TRANSLATION"

Answer:

In single buffering, a lexeme may cross the buffer boundary which requires the buffer to be refilled, thus overwriting the part of the lexeme that was in the buffer before reloading.

Example: Suppose the last 4 characters of the buffer are 'm', 'y', 'v' and 'a'; the new lexeme starts at 'm' and "myva" is possibly the start of a new identifier "myvar" (say). But the entire lexeme was not found in the buffer, causing it to be reloaded. The freash buffer has 'r' and '=' as the first two characters, clearly meaning that "myvar" was indeed the last lexeme. But while loading th buffer, the "myva" part of the identifier is overwritten.

The above problem is solved by a two buffer scheme. In that case, the problem can arise only if a lexeme is bigger than the size of the buffer which is extremely unlikely.

(d) What is sentinel?

Chapter Name: "SYNTAX DIRECTED TRANSLATION"

Answer:

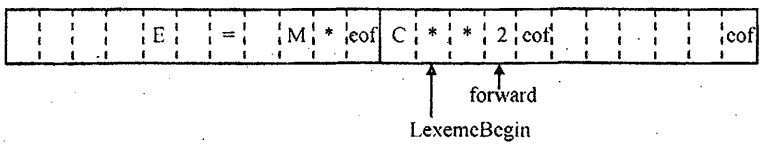
A Sentinel is a special character that cannot be part of the source program. It is added at each buffer end to test normally we use 'eof' as the sentinel. Actually sentinel saves time checking for the ends of buffers.

(e) What is its use?

Chapter Name: "SYNTAX DIRECTED TRANSLATION"

Answer:

This is used for speeding-up the lexical analyzer. It retains the role of entire input end. When it appears other than at the end of a buffer it means that the input is at an end



10. (a) What do you understand by L-attributed definition? Give example.

Answer:

Refer to "SYNTAX DIRECTED TRANSLATION", Short Answer Type Question No. 5.

(b) Describe with diagram the working process of Lexical Analyzer.

Answer:

Refer to "LEXICAL ANALYSIS", Short Answer Type Question No. 8(a).

(c) Describe LR passing with block diagram.

Answer:

Refer to "BOTTOM-UP PARSING AND LR PARSER GENERATION THEORY", Long Answer Type Question No. 5(a).

11. Write the short notes any *three* of the following:

(a) What is activation record? Explain clearly the components of an activation record.

Answer:

Refer to "RUNTIME ENVIRONMENT", Long Answer Type Question No. 2.

(b) YACC

Answer:

Refer to "LEXICAL ANALYSIS", Long Answer Type Question No. 2(b).

(c) Back patching

Answer:

Refer to "INTERMEDIATE CODE GENERATION", Long Answer Type Question No. 5.

(d) Thompson's construction

Answer:

Refer to "LEXICAL ANALYSIS", Long Answer Type Question No. 2(c).

(e) Constant folding and copy propagation.

Chapter Name: "CODE OPTIMIZATION"

Answer:

Constant folding and constant propagation are related compiler optimizations used by many modern compilers. An advanced form of constant propagation known as sparse conditional constant propagation can more accurately propagate constants and simultaneously remove dead code.

Constant folding is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime. Terms in constant expressions are typically simple literals, such as the integer literal 2, but they may also be variables whose values are known at compile time. Consider the statement:

```
i = 320 * 200 * 32;
```

Most modern compilers would not actually generate two multiply instructions and a store for this statement. Instead, they identify constructs such as these and substitute the computed values at compile time (in this case, 2,048,000). The resulting code would load the computed value and store it rather than loading and multiplying several values.

Constant folding can even use arithmetic identities. The value of 0 * x is zero even if the compiler does not know the value of x.

Constant folding may apply to more than just numbers. Concatenation of string literals and constant strings can be constant folded. Code such as "abc" + "def" may be replaced with "abcdef".

Constant folding can be done in a compiler's front end on the IR tree that represents the high-level source language, before it is translated into three-address code, or in the back end, as an adjunct to constant propagation.

Constant propagation is the process of substituting the values of known constants in expressions at compile time. Such constants include those defined above, as well as intrinsic functions applied to constant values.

Consider the following pseudocode:

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

Propagating x yields:

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

Continuing to propagate yields the following (which would likely be further optimized by dead code elimination of both x and y.)

```
int x = 14;
int y = 0;
return 0;
```

Constant propagation is implemented in compilers using reaching definition analysis results. If all a variable's reaching definitions are the same assignment, which assigns a same constant to the variable, then the variable has a constant value and can be replaced with the constant.

Constant propagation can also cause conditional branches to simplify to one or more unconditional statements, when the conditional expression can be evaluated to true or false at compile time to determine the only possible outcome.