2021

1.a What is the difference between a compiler and interpreter?

Answer:

Compiler	Interpreter
Compilers scan the entire program in one go.	The program is translated one line at a time.
As and when scanning is performed, all the errors are shown in the end together, not line by line.	One line of code is scanned, and errors encountered are shown.
Compilers convert the source code to object code.	Interpreters do not convert the source code into object code.
The execution time of compiler is less, hence it is preferred.	It is not preferred due to its slow speed.
The compiled code is platform dependent.	The interpreted code is platform independent.
Programming languages that use compilers include C, C++, C#, etc.	Interpreters are used for Python, Ruby, Perl, etc.

1.b Suppose a source program contains the assignment statement,

position = initial + rate * 60

Explain how this statement is processed and finally translated at different phases of a traditional compiler.

Answer:

The given statement

position = initial + rate * 60

Lexical Analysis:

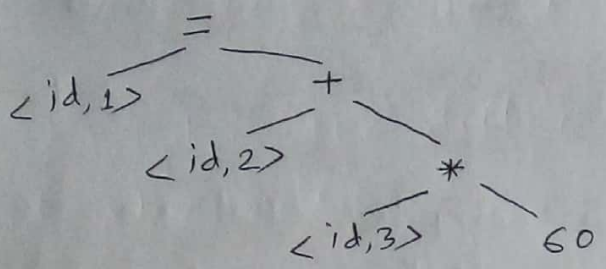
position, initial and rate are lexeme that would be mapped into tokens <id, 1>, <id, 2> and <id, 3> respectively, where id refers to identifier, and the following numbers point to the symbol table entry.

	name	type
1	position	float
2	intial	float
3	rate	float

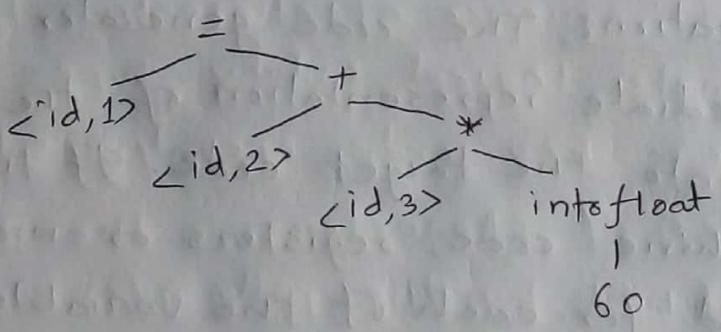
After the lexical analysis the assignment statement is represented as the sequence of tokens.

<id, 1> <=> <id, 2> <+> <id, 3> <*> <60>

Syntax Analysis: It uses the sequence of tokens produced by the lexical analyser to create a tree-like intermediate representation that depicts the grammatical structure of the stream of token. Where interior node represents an operation and the children of the node represent the arguments of the operation. A syntax tree for the given stream of tokens is



Semantic Analysis: The semantic analyser uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. An important part of semantic analysis is type checking. Here the identifiers are assumed to be float. Here the lexeme 60 is an integer. The multiplication (*) operation is performed between rate and 60 that is float and integer. So, an extra node is added for type convertion 'intofloat'



Intermediate Code Generation:

Intermediate code can be represented in the form of three-address-code which consists of a sequence of assembly-like instructions with three operands per instruction. Hence our assignment statement takes the following form.

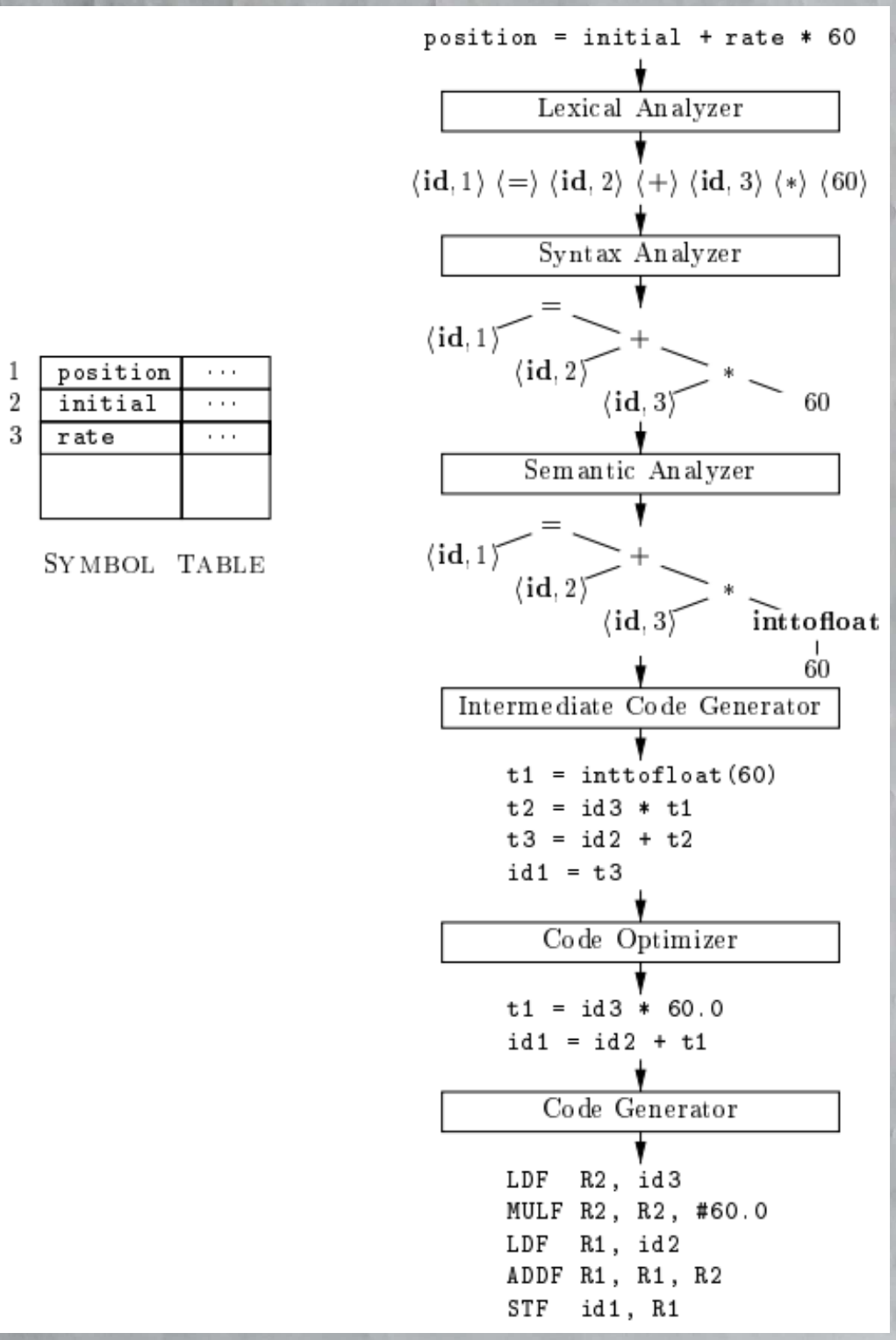
```
t1 = intofloat(60)
t2 = id3 * t1
t3 = id2 * t2
id1 = t3
```

Intermediate code optimization: The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. In our example the conversion of 60 from integer to floating point can be done once and for all at compile time. We can simply replace 60 by the floating point number 60.0. Moreover, t3 is used only once to transmit its value to id1, so it also can be optimized.

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Target Code Generation: The code generator takes as input an intermediate representation of the source code and maps it into the target code. If the target code is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. For example, use using registers R1 and R2, the intermediate code in might get translated into the machine code.

LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1



1.c Distinguish between single-pass and multi-pass compiler.

Answer:

A single-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 immediate representations in steps between source code and machine code, and which reprocesses the entire compilation unit in each sequential pass.

2.a Define token and lexeme. What are the functions of lexical analyser?

Answer:

Token: Token is a sequence of characters that can be treated as a single logical entity. Types of tokens are: Identifiers, keywords, operators, etc.

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

Functions of lexical analyzer:

1: It reads the input source program and produces as output, a sequence of tokens that the parser uses for syntax analysis.

2: Comments and white spaces (like tab, blank, new, line) donot influence code generation. The lexical analyser strips out the comments and white spaces

in the source program.
3: The lexical analyser keeps track of the newline characters, so that it can output the line number with associated error messages, in case of errors in the source program.

2.b Explain the necessity of regular expression and context free grammar in designing of a compiler.

Answer:

Regular expressions and context-free grammers are essential tools in the design and implementation of compilers. They play distinct roles in different phases of the compilation process.

Necessity of Regular Expression:

1. Lexical Analysis (Scanning): Regular Expressions (RE) are used to define the patterns of tokens in the source code. Tokens are the smallest units in a programming language. Lexical analyzers use RE to recognize and tokenize the input source code. Each token is described by a RE that captures its syntactic structure.
2. Automated tool generation: REs are often used in tools like Lex (Lexical Analyser Generator) to automatically generate lexical analyzers. This automation simplifies the process of building a lexer for a specific programing Language.

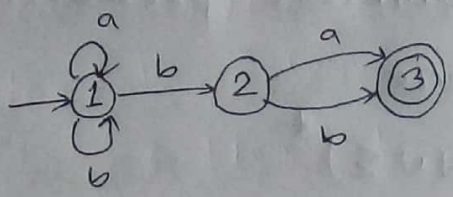
Necessity of Context Free Grammars (CFG):

1. **Syntax Analysis (Parsing):** CFGs are used to define the syntactic structure of a programming language. They describe how different language constructs are formed using a set of production rules. The parser uses the CFG to analyze the hierarchical structure of the source code and build a parse tree or an abstract syntax tree (AST).
2. **Automated Tool Generation:** Parser generators (e.g., Yacc/Bison) use CFGs to automatically generate parsers for a given language. This automation makes it easier to implement parsers and ensures consistency between the language specification and the parser.
3. **Error Detection:** CFGs are useful for error detection during the parsing phase. If the input source code does not conform to the grammar rules, the parser can detect syntax errors and report them to the user.

2.c Construct a lexical analyser (i.e. DFA) for the RE $(a|b)^*b(a|b)$.

Answer:

NFA:



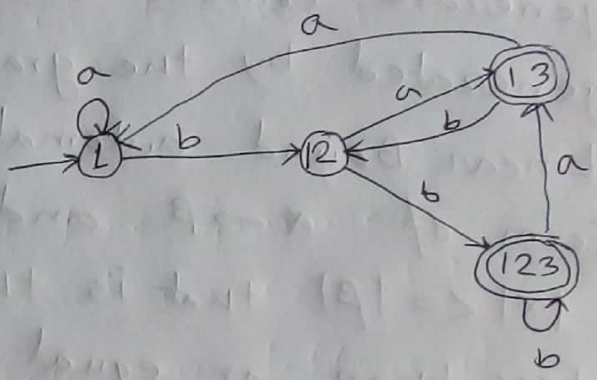
Transition Table:

	a	b
→ 1	{1}	{1, 2}
2	{3}	{3}
*3	ϕ	ϕ

Transition Table of DFA

	a	b
→ 1	{1}	{1, 2}
12	13	123
*13	1	12
*123	13	123

DFA:



3.a Write down formal definition of grammar. Discuss Chomsky hierarchy of formal gramars.

Answer:

A Grammar is formally defined as the tuple (N, Σ, P, s) where

N = set of non-terminal symbols

Σ = set of terminals

s = starting symbol

P = production rules

Chomsky hierarchy of formal Grammar:

Type 0: Unrestricted Grammar: Type-0 grammars include all formal grammar. Type-0 grammar languages are recognized by turing machine. These languages are also known as the Recursively

Enumerable languages. Grammar production in the form of $\alpha \rightarrow \beta$ where

$$\alpha \in (N \cup \Sigma)^* N (N \cup \Sigma)^*$$

$$\beta \in (N \cup \Sigma)^*$$

Type 1: Context Sensitive Grammar: Type 1 grammars generate context sensitive languages. The language generated by the grammar is recognized by the Linear Bound automata. Grammar production in the form of $\alpha \rightarrow \beta$ and $\alpha, \beta \in (N \cup \Sigma)^+$ and $|\alpha| <= |\beta|$ that is the count of symbol in α is less than or equal to β.

Type 2: Context Free Gramar: It generates context free language, which is recognized by a Pushdown automata. Grammar Production in the form of $\alpha \rightarrow \beta$ and $|\alpha| = 1$, that is, the left hand side of poduction have only one variable.

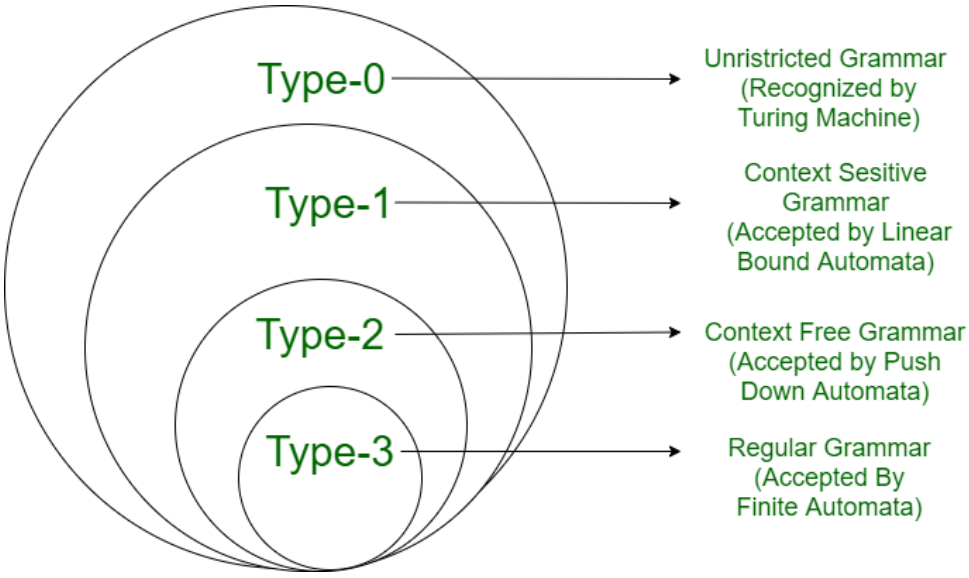
$$\alpha \in N \text{ and } \beta \in (N \cup \Sigma)^*$$

Type 3: Regular Grammar: It generates regular languages that can be recognized by finite-state Machine. Type-3 is the most restricted form of Grammar. The form of the Grammar is

$N \rightarrow N\Sigma^*/\Sigma^*$ (left-Regular G.)

$N \rightarrow \Sigma^* N/\Sigma^*$ (right-Regular G.)

* extended.


Type-0
Unristricted Grammar
(Recognized by
Turing Machine)
Type-1
Context Sesitive
Grammar
(Accepted by Linear
Bound Automata)
Type-2
Context Free Grammar
(Accepted by Push
Down Automata)
Type-3
Regular Grammar
(Accepted By
Finite Automata)

3.b Define normal Chomsky form (CNF). Convert following CFG into CNF:

i) $S \to ASaBA$ ii) $A \to B|S$ iii) $B \to b|\varepsilon$

Answer:

Chomsky Normal Form is defined using the following three conditions.

1. If there is an epsilon(ε) production, it must be generated by start symbol, $S \to \varepsilon$
2. A non-terminal can generate two non-terminals, $A \to BC$
3. A non-terminal can generate a terminal, $A \to a$

Also the start symbol can't be at the right side of production.

i) $S \to ASaBA$

Applying starting symbol rule:

$R_1 = S' \to S$

$S \to ASaBA$

Applying unit production rule:

$R_2 = S' \to ASaBA$

Applying Mixed Rule:

$R_3 = S' \to ASXBA$

$X \to a$

Applying Long production Rule

$R_4 = S' \to AW$

$W \to SZ$

$Z \to XY$

$Y \to BA$

$X \to a$

R = S → ASaBA
A → B|S
B → b|ε

Removing start symbol (S) from RHS:

R_1: S′ → S
S → ASaBA
A → B|S
B → b|ε

Removing null production: (A → ε, B → ε)

R_2: S′ → S
S → ASaBA|SaBA|ASaB|ASaA|SaB|SaA|ASa|Sa
A → B|S
B → b

Removing unit production: (S′ → S, A → B, A → S)

R_3: S′ → ASaBA|SaBA|ASaB|ASaA|SaB|SaA|ASa|Sa
S → ASaBA|SaBA|ASaB|ASaA|SaB|SaA|ASa|Sa
A → b|ASaBA|....
B → b

Removing Mixed productions:

R_4: S′ → ASXBA|SXBA|ASXB|ASXA|SXB|SXA|ASX|SX
S → ....
A → b|....
B → b
X → a

Removing long productions:

R_5: S′ → AW|ZA|AZ|AU|YB|YA|AY|SX
S → ....
A → b|....

$W \rightarrow ZA$

$Z \rightarrow YB$

[redacted]

$U \rightarrow YA$

$Y \rightarrow SX$

4.a Define ambiguity of CFG. Using disambiguation rule, make the grammar unambiguous:

$E \rightarrow (E) \mid E-E \mid E*E \mid E+E \mid id$

Answer:

A CFG is said to be ambiguous if it produces more than one parse tree for the same sentence. Ambiguous grammars produce more than one leftmost derivation or more than one rightmost derivation.

Removal of associativity and precedence ambiguity

$E \rightarrow E+T \mid E-T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Removal of left recursion

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid -TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

4.b Define LL(1) grammar. Convert the above grammar into LL(1). Construct a predictive parsing table using the grammar.

Answer:

A Grammar is said to be LL(1) if it is,

(i) unambiguous (ii) non-left recursive

(iii) in $A \to \alpha | \beta$, for strings α and β, they can't have same prefix.

Removal of Left Recursions makes it LL(1)

*Note: that step should be written here

	First	Follow
$E \to TE'$	{(, id}	{$,)}
$E' \to +TE' \mid -TE' \mid \varepsilon$	{+, −, ε}	{$,)}
$T \to FT'$	{(, id}	{+, −, $,)}
$T' \to *FT' \mid \varepsilon$	{*, ε}	{+, −, $,)}
$F \to (E) \mid id$	{(, id}	{*, +, −, $,)}

	id	+	−	*	(	)	$
E	$E \to TE'$				$E \to TE'$		
E'		$E' \to +TE'$	$E' \to -TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$				$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$				$F \to (E)$		

5.a What is compiler? Define the types of compilers.

Answer:

Compiler is a translator which is used to convert programs in high-level language to low-level language. It translates the entire program and also reports the errors in source program encountered during the translation.

Types of compilers

Single-pass Compiler: Processes the source code in a single pass, from lexical analysis to target code generation. It is memory efficient but may lack some advanced optimizing opportunities.

Multipass Compiler: Makes multiple passes over the source code, allowing for more extensive analysis and optimization. Multipass compilers often produce more efficient code but may require more memory.

Cross Compiler: These are the compilers that run on one machine and generate code for another machine.

Source to source/ trans compiler: These compilers convert the source code of one programming language to another programming language.

5.b Briefly discuss the functional components of a compiler.

Answer:

Lexical Analyser: The first functional component of a compiler is called the lexical analyzer or scanner or Lexer. It reads the source program and converts the sequence of characters into a sequence of tokens. So, it is also called tokenizer.

Syntax Analyzer: Also called parser, reads the sequence of tokens, produced by lexer and converts it into a tree-like intermediate representation called parse tree. It checks if the tree conforms the Grammar of the language.

Semantic Analyser: It uses the parse tree and symbol table to check the source program for semantic consistency with the language definition. For example: type checking.

Intermediate Code generator: It received input from its predecessor phase, semantic analyser, in the form of an annotated syntax tree. That is further converted to one or more intermediate representations, which can have a variety of form. Three-address-code is one of them.

Intermediate Code Optimization: The machine independent code optimization phase attempts to improve the intermediate code so that better code will result.

Target Code Generator: It takes as input an intermediate representation of the source program and maps it into the target language.

Target Code Optimization: In this phase the target code is transformed into a more efficient target code. The code optimization phase considers better usage of registers, usage of machine-specific idioms or features of the processor like specialized instructions, pipelineings, and so on to make the generated target code more efficient.

5.c What is front end and back end of a compiler? Why the compilers' functional components (phases) should be divided?

Answer:

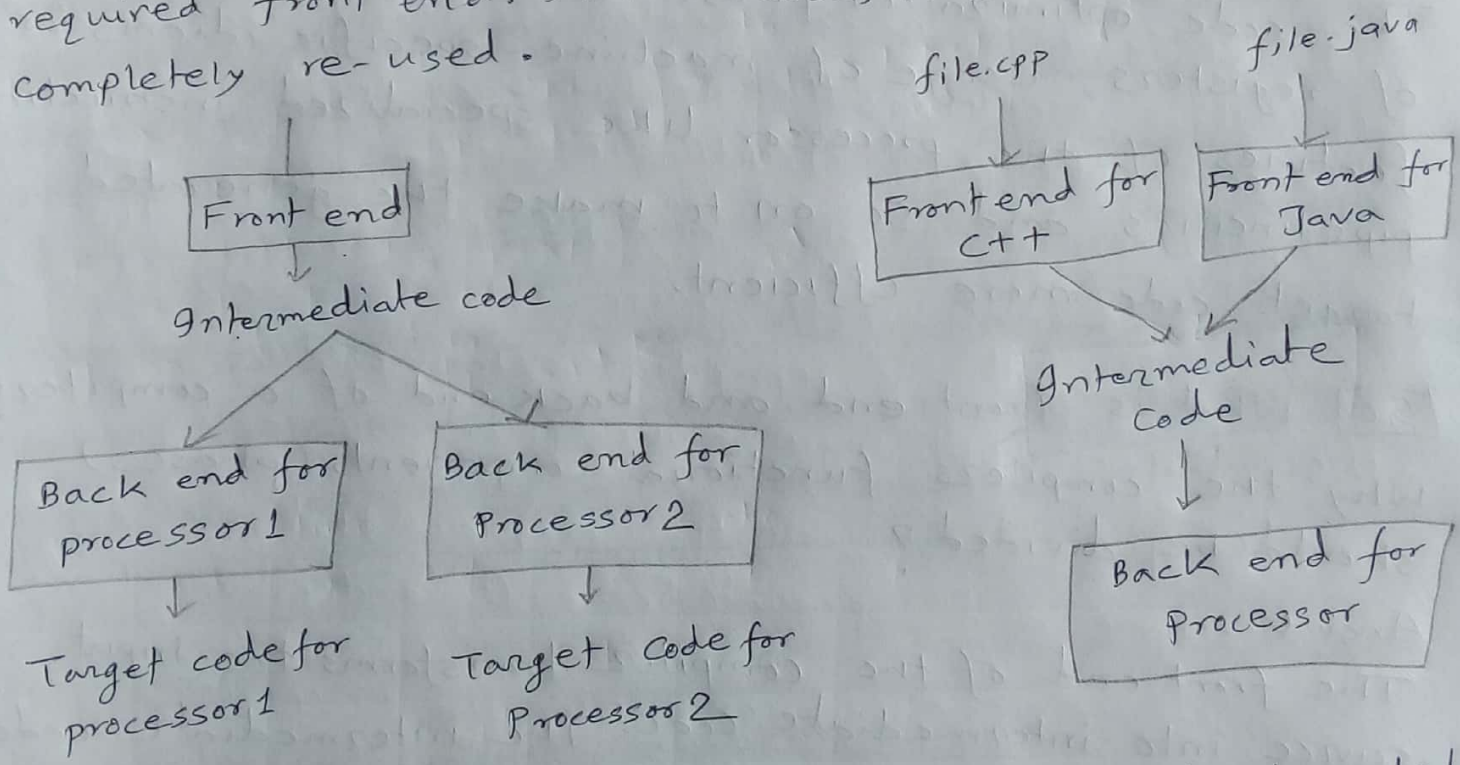
The front end of the compiler transforms the input source into intermediate code. The intermediate code is a machine-independent representation of the input source program.

The back end of the compiler takes the machine independent code and generates the target language program. The backend deals with machine specific details like the registers, number of allowable operators and so on.

The phases of compiler is divided into two ~~stat~~ stages (front end and back end) due to two reasons.

1. The compiler can be extended to support an additional processor by adding the required back end of the compiler. The existing front end is completely re-used in this case.

2. The compiler can be easily extended to support an additional input source language by adding the required front end. In this case, the back end is completely re-used.



6.a What is bottom-up parsing? How it is implemented

Answer:

Bottom-up parsing is a process in which, the parse tree for an input string is constructed beginning at the leaves (the bottom) and working up towards the root (the top). Bottom-up parsing

involves 'reducing' the input string 'w' to the start symbol of the grammar.

Bottom-up parsing is often implemented using a data structure called a parsing table and a control structure know as a state machine. The parsing table is constructed based on the grammar and contains information about the actions to be taken in different states, where each state corresponds to a configuration of the parsing stack and input.

6.b Construct an operator relation table for operator precedence parser for the following grammar:

$E \rightarrow EAE \mid -E \mid id$

$A \rightarrow + \mid *$

Answer:

The Grammar doesn't follow the rules of operator precedence. So, the operator precedence form is as follows

$E \rightarrow E+E \mid E*E \mid -E \mid id$

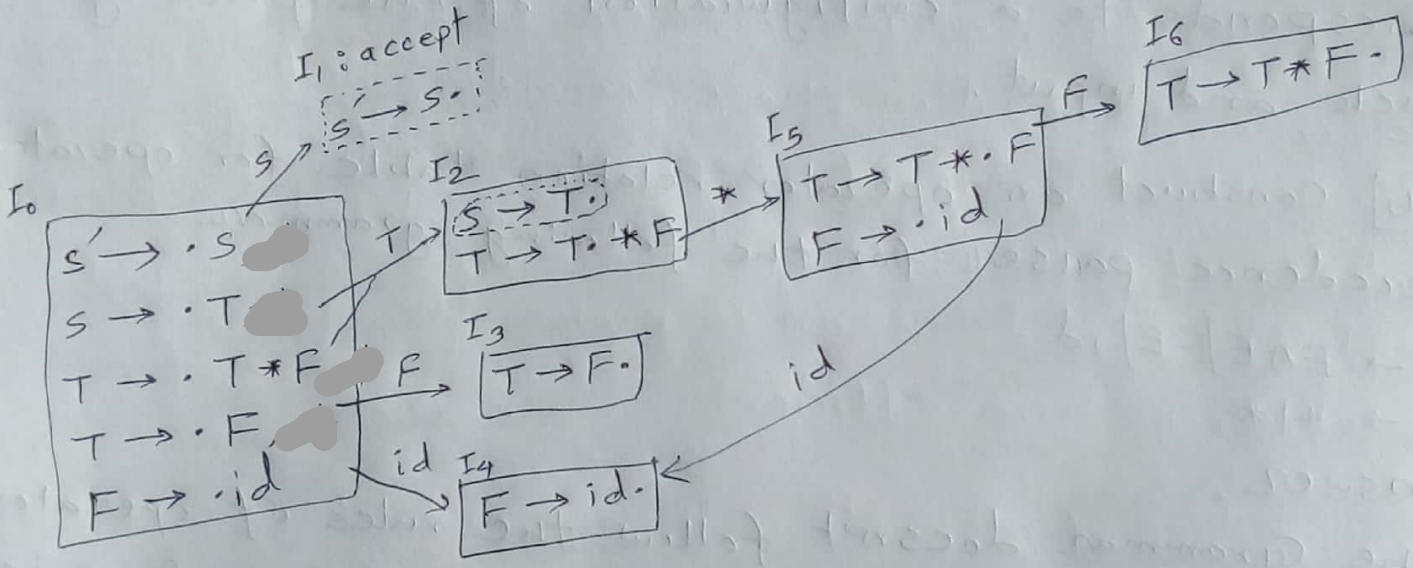
	id	*	+	−	$
id	err	·>	·>	·>	·>
*	<·	·>	·>	·>	·>
+	<·	<·	·>	·>	·>
−	<·	<·	·>	·>	·>
$	<·	<·	<·	<·	accepted

6.c Check following grammar SLR(1) or not:

$S \to T,\ T \to T * F \mid F,\ F \to id$

Answer:

	Follow
$S \to T$ ①	{$}
$T \to T * F$ ② | F ③	{$, *}
$F \to id$ ④	{$, *}



States	Action			Goto		
	id	*	$	S	T	F
I_0	S_4			1	2	3
I_1			accept			
I_2		S_5	r_1			
I_3		r_3	r_3			
I_4		r_4	r_4			
I_5	S_4					6
I_6		r_2	r_2			

No confict, hence SLR(1).

7.a Explain syntax-directed translation (SDT) scheme.

Answer:

The main idea behind SDT is that the semantics or the meaning of the program is closely tied to its syntax. SDT involves:

1. Identifying attributes of the grammar symbols in the CFG.
2. Specifying semantic rules or attribute equations relating the attributes and associate them with the productions.
3. Evaluating semantic rules to cause valueable side-effects like insertion of information into the symbol table, semantic checking, issuing of a error message, generation of intermediate code, and so on.

7.b Write down SDT for following CFG:
$S \rightarrow id = E$, $E \rightarrow E-T \mid T$, $T \rightarrow T/F \mid F$, $F \rightarrow id$ using the required SDT produce three-address code for the statement "$x = a - b/c$".

Answer:

$S \rightarrow id = E$ { gen (id.name = E.place); }

$E \rightarrow E-T$ { E.place = new temp(); gen(E.place = E.place - T.place); }

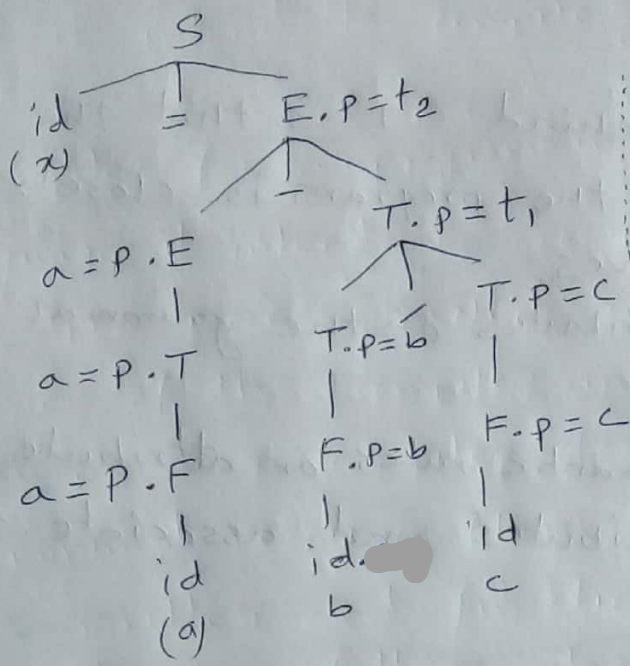
$\mid T$ { E.place = T.place; }

$T \rightarrow T/F$ { T.place = new temp(); gen(T.place = T.place / F.place); }

$\mid F$ { T.place = F.place; }

$F \rightarrow id$ { F.place = id.name }

three address code for $x = a - b/c$ using SDT



$t_1 = b/c$

$t_2 = a - t_1$

$x = t_2$

Three address code

here p. stands for place

7.c) Write down a postfix notation for the infix statement "if a then if c-d then a+c else a*c else a+b".

Answer:

if a then
    if c-d then
        a+c
    else
        a*c
else
    a+b

cd-ac+ac*?

acd-ac+ac*? ab+?

2020

1.a Figure 1(a) shows four different ways to translate source code in a high level language into machine code. Write which kind of translator can be used in each case. What are the positive sides of each translator?

Answer:

1.b How can be a compiler written in the source language that it intends to compile?

Answer:

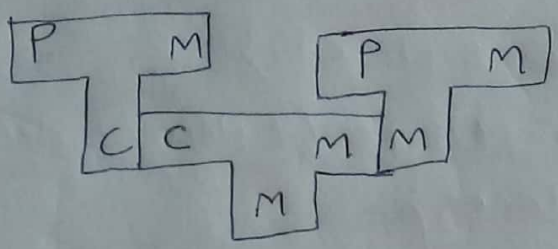
Writing a compiler for a programming language in the same language it intends to compile is known as bootstrapping. The process involves implementing a minimal version of the compiler in a lower-level language or an existing compiler for a different language. Once this minimal version, often called a "bootstrapping compiler" is written and functional, it can be used to compile the full compiler written in the source language.

Lets create a compiler for Pascal (Pa) using bootstrapping method:

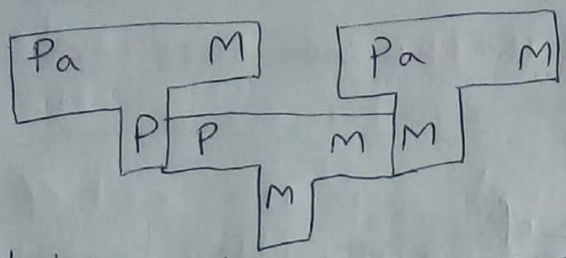
First take a subset of the language Pascal(Pa), let it be P. Now the minimal version ~~is~~ of the compiler will be written in C language, for P.

Then the 'T' diagram will be

M: machine code



Now, we have got the minimal version of the compiler. Now the entire compiler can be written using the source language.



Now this compiler can compile the full compiler.
minimal

2.a Define pattern with example:

Answer:

Pattern: A set of strings in the input for which the same token is produced. This set of strings is described by a rule called a pattern associated with the token.

For an example in c language

l = Letter
d = digit

int number;

Lexeme	Token	pattern
int	keyword	int
number	identifier	[a-zA-Z_][a-zA-Z0-9_]* or (l|_)(l|d|_)*
;	semicolon	;

2.b state the role of lexical analyser with a neat diagram. Identify the lexemes and their corresponding tokens in the following statement:

printf("simple Interest = %f\n", si);

Answer:

The role of lexical analyser is to read the source code of a program and break it into a sequence of tokens.

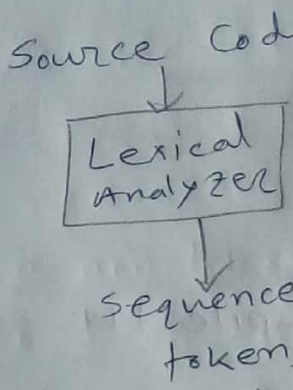


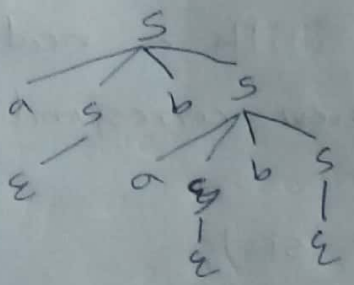
figure: Neat diagram

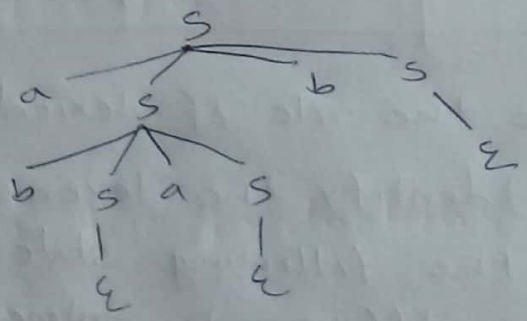
Lexeme	Token
printf	identifier
(	Punctuator
"simple....."	String Literal
,	punctuator
si	identifier
)	punctuator
;	Terminator or punctuator

2.c Consider the CFG $S \rightarrow aSbS \mid bSaS \mid \varepsilon$. Check whether the grammar is ambiguous or not. Why?

Answer:

W = abab





two parse tree: Ambiguous.

3.a] How can you eliminate left-recursion from a grammar? Eliminate left recursion from the following grammar i) $S \to Aa|b$ ii) $S \to Ac|Sd|\varepsilon$.

Answer:

A Grammar is said to be left recursive if it has a non-terminal A such that there is a derivation $A \to Aa$. Top-down parsing methods cannot handle left recursive grammars. Hence left recursion can be eliminated as follows:

If there is a production $A \to A\alpha|\beta$ it can be replaced with a sequence of two productions

$$A \to \beta A'$$
$$A' \to \alpha A'|\varepsilon$$

i) $S \to Aa|b$

$A \to Ac|Sd|\varepsilon$ [correction]

Here, $A \to Sd$ is an indirect left recursion

Rewritting the grammar

$S \to Aa|b$

$A \to Ac|Aad|bd|\varepsilon$

Now Revome left recursion

$S \to Aa|b$

$A \to bdA'|A'$

$A' \to cA'|adA'|\varepsilon$

3.b What do you mean by FIRST and FOLLOW sets? Compute the FIRST of all the non-terminals

Answer:

A first set of a non-terminal A is the set of all the terminals that A can begin with.

A follow set of a non-terminal A is the set of all the terminals that can follow A.

The FOLLOW set would never contain ε, since it is not a valid input token. This is in contrast from the FIRST set, which can contain ε.

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

4.a Sometimes left factoring is needed; why? The following grammar abstracts the "dangling-else" problem:

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

Answer:

If a grammar contains two productions of form

$S \rightarrow a\alpha$ and $S \rightarrow a\beta$

it is not suitable for top down parsing without backtracking. Troubles of this form can sometimes be removed from the grammar by a technique called the left factoring.

In the left factoring, we replace

$S \rightarrow \alpha\alpha \mid \alpha\beta$ by

$S \rightarrow \alpha S'$

$S' \rightarrow \alpha \mid \beta$

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b$

4.b Write an algorithm for predictive parsing table.

Answer:

Input: Grammar G

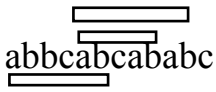
Output: Parsing Table M

Method:

1. For each production $A \rightarrow \alpha$ of G, do steps 2 & 3.
2. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to M[A, a].
3. If ε is in FIRST(α), add $A \rightarrow \alpha$ to M[A, b] for each terminal b in FOLLOW(A). If ε is in FIRST(α) and $ is in FOLLOW(A), add $A \rightarrow \alpha$ to M[A, $].
4. Make each undefined entry of M be error.

4.c What are the rules of type checking? Briefly illustrate how can type conversions happen?

Problem 3: Construct a Syntax-Directed Translation scheme that takes strings of a's, b's and c's as input and produces as output the number of substrings in the input string that correspond to the pattern a(a|b)*c+(a|b)*b. For example the translation of the input string "abbcabcababc" is "3".



Your solution should include:

a) A context-free grammar that generates all strings of a's, b's and c's
b) Semantic attributes for the grammar symbols
c) For each production of the grammar a set of rules for evaluation of the semantic attributes
d) Justification that your solution is correct.

Solution:

a) The context-free grammar can be as simple as the one shown below which is essentially a Regular Grammar G = {{a,b,c}, {S}, S, P} for all the strings over the alphabet {a,b,c} with P as the set of productions given below.

$S \rightarrow S\ a$
$S \rightarrow S\ b$
$S \rightarrow S\ c$
$S \rightarrow a$
$S \rightarrow b$
$S \rightarrow c$

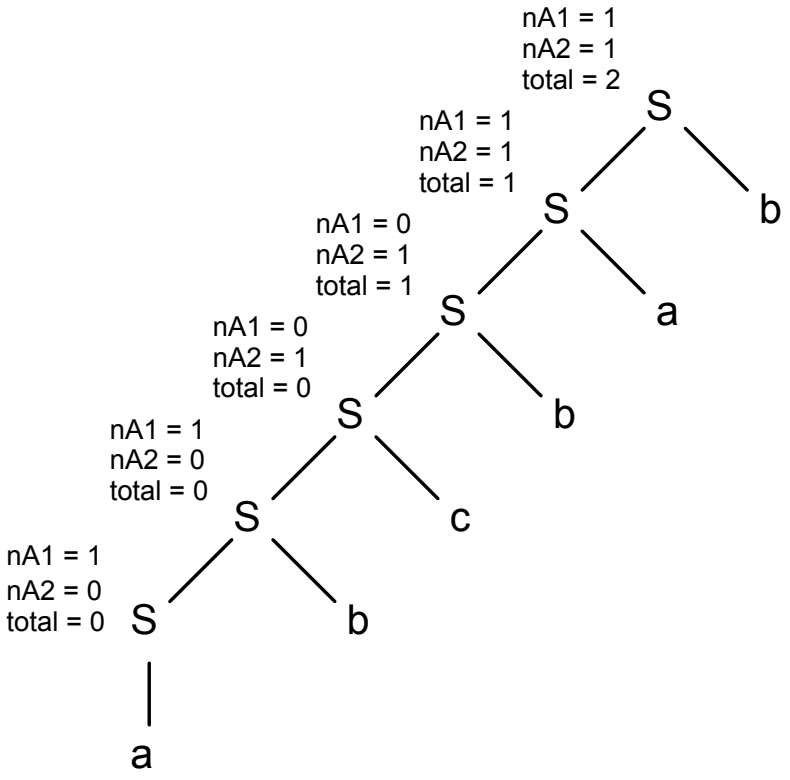
b) Given the grammar above any string will be parsed and have a parse tree that is left-skewed, i.e., all the branches of the tree are to the left as the grammar is clearly left-recursive. We define three **synthesized** attributes for the non-terminal symbol S, namely nA1, nA2 and total. The idea of these attributes is that in the first attribute we will capture the number of a's to the left of a given "c" character, the second attribute, nA2, the number of a's to the right of a given "c" character and the last attributed, total, will accumulate the number of substrings.

We need to count the number of a's to the left of a "c"c charecter and to the right of that character so that we can then add the value of nA1 to a running total for each occurrence of a b character to the right of "c" which recording the value of a's to the right of "c" so that when we find a new "c" we copy the value of the "a's" that were to the right of the first "c" and which are now to the left of the second "c".

c) As such a set of rules is as follows, here written as semantic actions given that their order of evaluation is done using a bottom-up depth-first search traversal.

$S_1 \rightarrow S_2\ a$	{ $S_1.nA1 = S_2.nA1 + 1$; $S_1.nA2 = S_2.nA2$; $S_1.total = S_2.total$; }
$S_1 \rightarrow S_2\ b$	{ $S_1.nA1 = S_2.nA1$; $S_1.nA2 = S_2.nA2$; $S_1.total = S_2.total + S_2.nA2$; }
$S_1 \rightarrow S_2\ c$	{ $S_1.nA1 = 0$; $S_1.nA2 = S_2.nA1$; $S_1.total = S_2.total$; }
$S_1 \rightarrow a$	{ $S_1.nA1 = 1$; $S_1.nA2 = 0$; $S_1.total = 0$; }
$S_1 \rightarrow b$	{ $S_1.nA1 = 0$; $S_1.nA2 = 0$; $S_1.total = 0$; }
$S_1 \rightarrow c$	{ $S_1.nA1 = 0$; $S_1.nA2 = 0$; $S_1.total = 0$; }

d)



We have two rolling counters for the number of a's one keeping track of the number of a's in the current section of the input string (the sections are delimited by "c" or sequences of "c"s) and the other just saves the number of c's from the previous section. In each section we accumulate the number of a's in the previous section for each occurrence of the "b" characters in the current section. At the end of each section we reset one of the counters and save the other counter for the next section.

5.b Differentiate between S-attributed SDT and L-attributed SDT with suitable examples.

Answer:

S-attributed SDT	L-attributed SDT
A SDT that uses only synthesized attribute is called S-attributed SDT.	A SDT that uses both synthesized and inherited attributes is called as L-attributed SDT but each inherited attribute is restricted to inherit from parent or left siblings only.
Uses bottom up parsing	Uses top down parsing
Semantic rules are always written at the leftmost position in RHS.	Semantic rules can be applied anywhere at RHS.
$E \rightarrow E_1 + T \{ E.val = E_1.val + T.val \}$	$E \rightarrow TE' \{ E.val = E'.inh \}$ $E' \rightarrow +TE' \{ E'.inh = E.val + T.val \}$ $\mid \varepsilon \{ E'.inh = E.val \}$

5.c Explain Common sub expression elimination with an example.

Answer:

The expession or sub-expression that has been appeared and computed before and appears again during the computation of the code is the common sub-expression. Elimination of that sub-expression is known as Common sub-expression Elimination.

Example:

```
a = b+c          a = b+c
b = a-d    ⟶     b = a-d
c = b+c          c = a
```

6.a Why is a symbol table needed? Write down the purpose and operations of the symbol table.

Answer:

A symbol table is needed to store information with regard to the identifiers used in the input source program. Each entry in the symbol table corresponds to a symbol (identifier) and contains details like the name of the symbol, the data type, size (the amount of memory required) and ~~contains details like the name of the symbo~~ so on. Typically, an entry is made into the symbol table at the analysis phase. The entry is updated and looked up in different phases of the compiler.

Purpose:

1. It is used to store the name of all entities in a structured form at one place.
2. It is used to verify if a variable has been declared.
3. It is used to determine the scope of a name.
4. It is used to implement type checking by verifying assignments and expressions in the source code are semantically correct.

operation:

allocate: To allocate a new empty symbol table

free: To remove all entries and free storage of symbol table.

lookup: To search for a name and return a pointer to its entry.

insert: To insert a name in a symbol table and return a pointer to its entry

set_attribute: To associate an attribute with a given array.

get_attribute: To get an attribute associated with a given array.

6.b Write the Basic blocks and draw control flow graph of the source code.

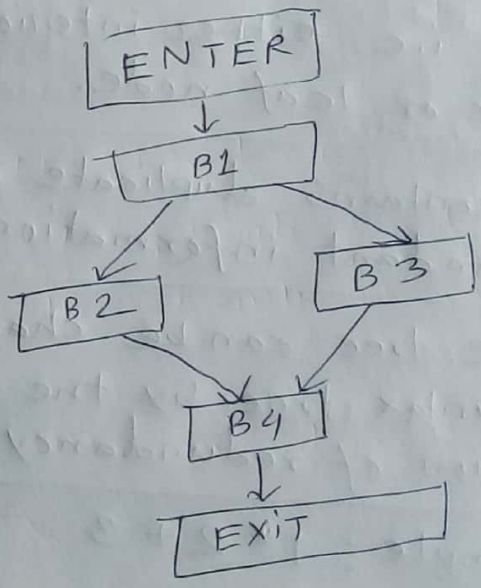
```
w=0;
y=0;
if(x>z){
    y=x;
    x++;
}else{
    y=z;
    z++;
}
w=x+z
```

B1:
```
w=0;
y=0;
if(x>z)
```

B2:
```
y=x;
x++;
```

B3:
```
y=z;
z++;
```

B4:
```
w=x+z;
```



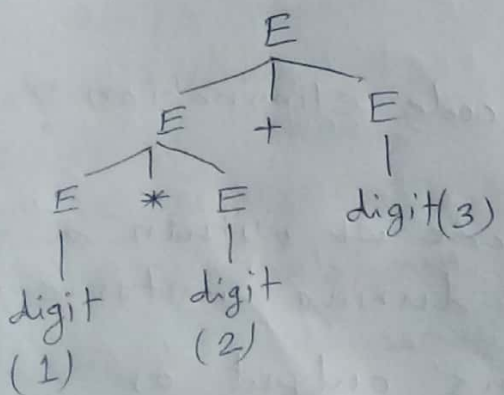
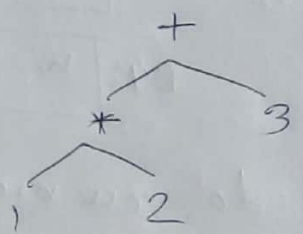
6.c What do you mean by dead code elimination?

Answer:

Dead code refers to blocks of code or within a program that is never executed during runtime or has no impact on the programs' output or behavior.

Identifying and removing dead code is essential for improving program efficiency, reducing complexity, and enhancing maintainability. Identification of dead code is performed by control flow analysis or data flow analysis. The identified dead code is eliminated during generation of executable code.

7.a Write down differences between the parse tree and syntax tree with proper examples.

Parse Tree	Syntax Tree
It can contain operators and operands at any node of the tree, i.e, either interior node or leaf node.	It contains operands at leaf node and operators as interior nodes of tree.
It contains duplicate or redundant information.	It does not include any redundant data.
Parse tree can be changed to syntax tree by the elimination of redundancy.	Syntax tree can't be changed to parse tree.
Example: 1 * 2 + 3 	Example: 1 * 2 + 3 

7. b) Draw the syntax tree and parse tree of the expression $(A+B/C)/(A-C/F)*F+(H*Y*Z)$

Answer:

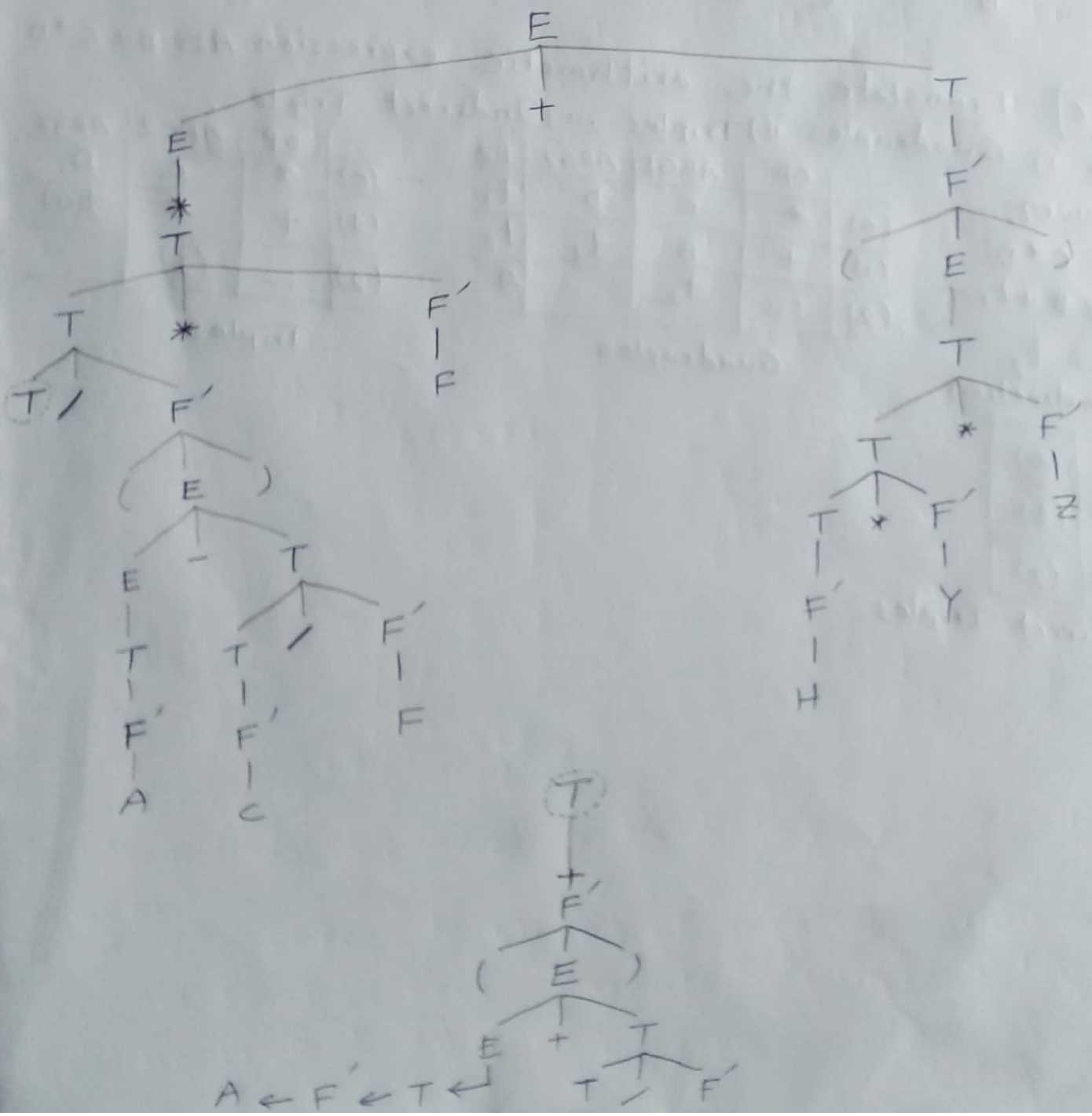
Let the grammar be

$E \rightarrow E+T \mid E-T \mid T$

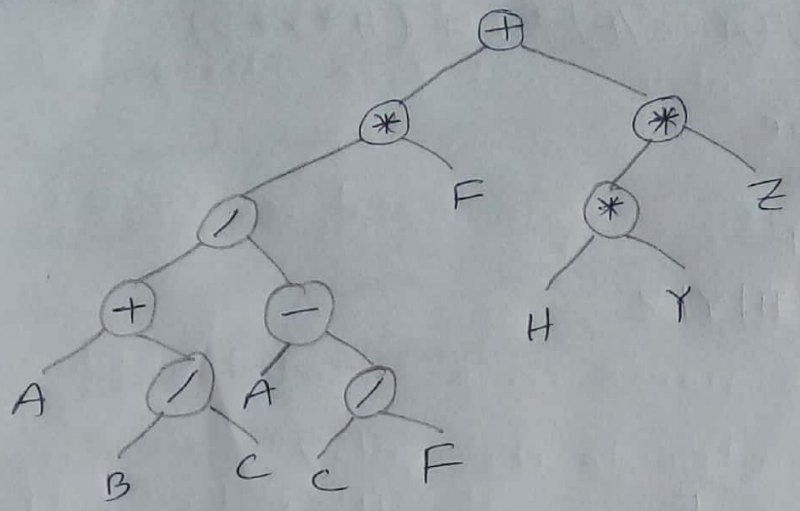
$T \rightarrow T*F' \mid T/F' \mid F'$

$F' \rightarrow (E) \mid A \mid B \mid C \mid F \mid H \mid Y \mid Z$

parse Tree



Syntax tree



7. C Translate the arithmetic expression A := B + C * D into i) Quadruples ii) Triples iii) Indirect Triples.

Answer:

$t_1 := C * D$

$t_2 := B + t_1$

$A := t_2$

	op	ARG1	ARG2	RZ
(0)	*	C	D	t_1
(1)	+	B	t_1	t_2
(2)	:=	t_2		A

Quadruples

	op	ARG1	ARG2
(0)	*	C	D
(1)	+	B	(0)
(2)	:=	(1)	

Triples

instruction	
100	(0)
101	(1)
102	(2)

Indirect Triples.