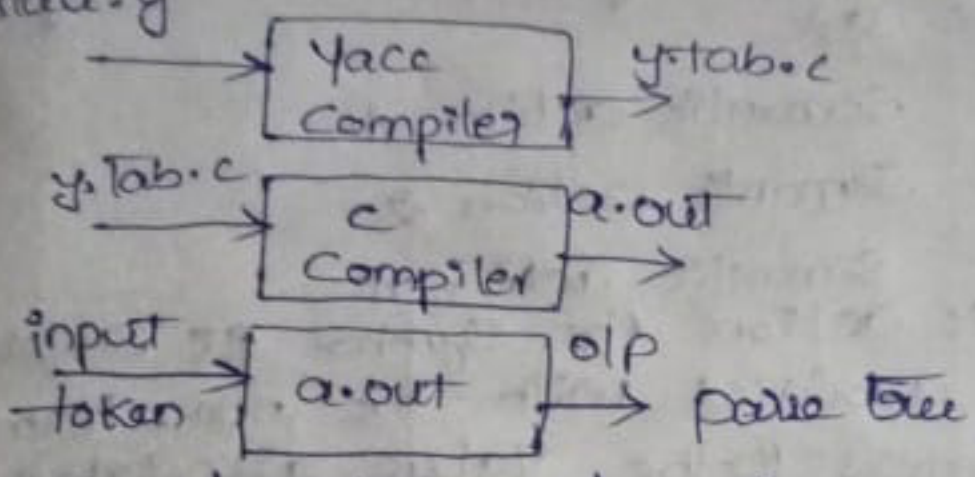
Assignment - 3

Q.a) Briefly explain about the automatic parser generator - YACC.

Ans) yacc means yet another Compiler Compiler

A translator can be constructed using YACC

translate.y



First a file translate.y translated into a C program called y.tab.c using yacc compiler. By compiling y.tab.c with C Compiler the desired object program a.out will generate. If you give tokens as input to the a.out then it will produce parse tree as a output.

Structure of YACC program:-

=> Declarative section

=> Translation rule section.

=> Supporting C routines (or) auxiliary function section.

=> Declarative section:-

=> There are two subsections in the declarations part of yacc program but both are optional.

=> In the first subsection we put ordinary C declarations in between %{ %} here we place declarations of any temporaries or procedures of the second and third sections that is the header files, declaratives of variables and constants.

```
%{
  #include <type.h>
%}
```

The translation rule section:-
=> In the part of YACC specification after the first %% pair we put the translation rules. Each rule consists of a grammar production and associated semantic actions.
=> A set of productions that we have been writting.
head → <body1> / <body2> / - - - / <bodyn> can be written in YACC as
head : body1 Semantic action 1
| body2 Semantic action 2
| bodyn Semantic action n
=> In a semantic action the symbol $$ refers to the attribute value associated with the non-terminal of the head $$; refers to the value associated with the ith grammar symbol of the body.
The supporting C routines or auxiliary functions:-
The third part of yacc specifications consist of supporting C routines.
The lexical analyzer by the name yylex function must be provided in the section.

5.a) Construct CLR parsing table for the grammar.

S → AA
A → aA
A → b

a) Step 1
Given grammar

S → AA
A → aA
A → b

Step 1:-
S' → S
S → AA
A → aA
A → b

Step 1: • at right side of the productions.

$S' \rightarrow .S$
$S \rightarrow .AA$
$A \rightarrow .aA$
$A \rightarrow .b$

Step 3:

$S' \rightarrow .S, \$$
$S \rightarrow .aA, \$$
$A \rightarrow .b, a/b$ } I_0

Goto (I_0, S)
$S \rightarrow S., \$$ } I_1

Goto (I_0, A)
$S \rightarrow A.A, \$$
$A \rightarrow .aA, \$$
$A \rightarrow .d; \$$ } I_2

Goto (I_0, a)
$A \rightarrow a.A, a/b$
$A \rightarrow .aA, a/b$
$A \rightarrow .b, a/b$ } I_3

Goto (I_0, d)
$A \rightarrow b., a/b$ } I_4

Goto (I_2, A)
$S \rightarrow aA. , \$$ } I_5

Goto (I_2, a)
$A \rightarrow A.A, \$$
$A \rightarrow .aA, \$$
$A \rightarrow .b, \$$ } I_6

Goto (I_2, b)
$A \rightarrow b., \$$ } I_7

Goto (I_3, A)
$A \rightarrow aA., a/b$ } I_8

Goto (I_3, a)
$A \rightarrow aA, a/b$
$A \rightarrow .aA, a/b$
$A \rightarrow .b, a/b$ } I_3 (same)

Goto (I_3, b)
$A \rightarrow b., a/b.$

Goto (I_6, A)
$A \rightarrow aA., \$$ } I_9

Goto (I_6, a)
$A \rightarrow a.c, \$$
$A \rightarrow .aA, \$$
$A \rightarrow .b, \$$

Goto (I_6, b)
$A \rightarrow b., \$$

step4 :- construct the CLR parse table:

	Action			Goto	
	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1	~~S6~~		Accept		
I_2	S6	S7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			
I_5			R_1		
I_6	S6	S7			9
I_7			R_3		
I_8					
I_9			R_2		

4.a) Comparison of SLR, CLR and LALR parsers:

a) Feature	SLR parser (Simple LR)	CLR parser (Canonical LR)	LALR (Look ahead LR)
parsing Table size	smallest	Largest	Medium
Grammar Handling	limited	Most powerful	Nearly as powerful as CLR but compact.
Basic qd decisions	use FOLLOW set qd reductions.	Uses look-ahead symbols to make precise decisions	Uses merged look-ahead symbols. Similar to CLR but optimized.
Conflict (Shift-Reduce, Reduce-Reduce)	More conflict due to reliance on FOLLOW set.	Least conflict because of look-ahead symbols	May introduce reduce-reduce conflict when merging states.

5. // S → A → Aa / b
b explain equiv

Error Detection	Delayed (error detected later)	Delayed (similar to SLR)	Similar to CLR, not always immediate.
Time and space complexity	Low (fast but limited)	High (slow due to large tables but powerful)	Medium (optimized for efficiency)
Ease of implementation	Easiest (simplest to build)	Most Complex (large tables make it hard)	Easier than CLR but slightly more complex than SLR
Used in	Simple parsers and educational tools	Strong theoretical Compilers	Most real-world Compilers (YACC etc)

4.b) Why we need LR parser and explain the working of LR parser?

a) LR parsers are essential in compiler design because they provide a robust and efficient way to analyze the syntax of programming languages. They are bottom-up parsers, meaning they construct the parse tree from the input tokens up to the start symbol, and they can handle a wide variety of grammars.

Here a more detailed explaination of why LR parsers are important:-

1. Robustness and Efficiency.
2. Handling Complex Languages
3. Implementation.
4. Error Detection
5. Widely Used

Working of LR parser:-

1. Initialization:
- An empty stack is initialized, and a special end-of-input marker ($) is placed at the bottom.
- The start symbol of the grammar is placed on top of the stack.

2. Input Reading:-
The input string is read from left-to right, Symbol by Symbol

3. Shift and Reduce Operations:-

Shift:-
The current Input Symbol is moved from the input string to the top of the stack.

Reduce:-
If the Symbols on top of the Stack match the right-hand side of a production rule, they are replaced with the left-hand side non-terminal.

5.b) Briefly explain equivalence of type expression and type conversion?

a) Type expression equivalence determines when two type descriptions are considered the same, while type conversion changes the data type of an expression.

Type expression equivalence:-

⇒ Structural Equivalence:-
Two types are structurally equivalent if they have the same underlying structure, regardless of the name used for the type. For example, if two types are both defined as "array of integer", they would be structurally equivalent even if one type was named "int-array" and the other was named "integer-list".

Name Equivalence:-
Two types are name equivalent if they have the same name regardless of their underlying structure. For example, if a type called, "my-type" is defined as "array of integer", & another type also named "my-time" is defined elsewhere, they would be name equivalent.

Type Conversion:-
⇒ implicit Conversion:-
This occurs automatically by the compiler during runtime, without explicit instructions from the programmer. For instance, converting an integer to a float might be done implicitly.
⇒ Explicit Conversion:-
This is done by the grammar using a special operator or function to change the type of an expression or variable. For example, inc, (float) i, explicitly converts the integer variable i to a float
⇒ purpose:-
Type conversion is often used to allow code to work with different data types together or to prepare data for specific operations that require a particular type.

2-b) Differences between S-attributes and L-attributed grammars. Discuss their significance in syntax directed translation?

a)	S-attributed SDT	L-attributed SDT
1.	Uses only Synthesized attribute	1. Uses both Synthesized and inherited attribute. inherited attribut can inherit values from either parent or left sibling only.
2.	Semantic actions are placed at right end of production. $A \rightarrow BCD$ $A \rightarrow BCD\{A\}$	2. Semantic actions are placed anywhere on right hand side of production. $A \rightarrow BCD$ $A \rightarrow \{SA\}\ BCD$ $A \rightarrow B\{SA\}\ CD$ $A \rightarrow BC\{SA\}\ D$ $A \rightarrow BCD\{SA\}$

3. Attributes are evaluated bottom up parsing.	3. Attributes are evaluated by travelling parse tree depth first and left to right.
4. No strict rules on attribute dependencies.	4. Some rules limit where attributes can get their values from, especially for inherited attributes.
5. In arithmetic expression, the value of an expression depends on the values of its parts, so it's S-attributed.	5. In type checking, the type of an expression might depend on its context, so it often L-attributed.

Significance in SDT :-

1. Semantic Rules Attachment :

Attributes allow attaching semantic to grammar rules - Critical for tasks like type checking, intermediate code generation etc.

2. Compiler phases :-

S-attributed grammars are often used during code generation stages in bottom-up parsers.

L-attributed grammars are often used in semantic analysis and symbol-table management during top down parsing.

3. Modularity & Maintainability :-

Attribute grammars modularize semantic actions cleanly, aiding in better design and maintenance of compilers.

3. a) Write about function and operator overloading?

An overloaded symbol has different meanings depending on its context. Overloading is resolved when a unique meaning is determined for each occurrence of a name. In this section, we restrict attention to overloading that can be resolved by looking only at the arguments of a function, as in Java.

Example:- The + operator in Java denotes either String Concatenation or addition, depending on the types of its operands. User defined functions can be overloaded as well as in.

```
Void err() {...}
Void err(String S) {...}
```

Note that we can choose between these two versions of a function err by looking at their arguments.

The following is a type-synthesis rule for overloaded functions:

If f can have type $S_i \to t_i$, for $1 \le i \le n$, where $S_i \neq S_j$ for $i \neq j$ and x has type S_k, for some $1 \le k \le n$ then expression f(x) has type t_k.

The value-number method can be applied to type expressions to resolve overloading based on argument types, efficiently.

1) Construct SLR and LALR parsing table for the gramma

$S \to L=R \mid R$
$L \to *R \mid id$
$R \to L$

a) We'll first argument the gramma by adding a new start Symbol:

$S' \to S$

Step 1:- FIRST and FOLLOW Sets

FIRST Sets:-

step4 :- construct the CLR parse table:

	Action			Goto	
	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1	~~S6~~		Accept		
I_2	S6	S7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			
I_5			R_1		
I_6	S6	S7			9
I_7			R_3		
I_8					
I_9			R_2		

4.a) Comparison of SLR, CLR and LALR parsers:

a) Feature	SLR parser (Simple LR)	CLR parser (Canonical LR)	LALR (Look ahead LR)
parsing Table size	smallest	Largest	Medium
Grammar Handling	limited	Most powerful	Nearly as powerful as CLR but compact.
Basic qd decisions	use FOLLOW set qd reductions.	Uses look-ahead symbols to make precise decisions	Uses merged look-ahead symbols. Similar to CLR but optimized.
Conflict (Shift-Reduce, Reduce-Reduce)	More conflict due to reliance on FOLLOW set.	Least conflict because of look-ahead symbols	May introduce reduce-reduce conflict when merging states.

5. // S → A → Aa / b
b explain equiv

Error Detection	Delayed (error detected later)	Delayed (similar to SLR)	Similar to CLR, not always immediate.
Time and space complexity	Low (fast but limited)	High (slow due to large tables but powerful)	Medium (optimized for efficiency)
Ease of implementation	Easiest (simplest to build)	Most Complex (large tables make it hard)	Easier than CLR but slightly more complex than SLR
Used in	Simple parsers and educational tools	Strong theoretical Compilers	Most real-world Compilers (YACC etc)

4.b) Why we need LR parser and explain the working of LR parser?

a) LR parsers are essential in compiler design because they provide a robust and efficient way to analyze the syntax of programming languages. They are bottom-up parsers, meaning they construct the parse tree from the input tokens up to the start symbol, and they can handle a wide variety of grammars.

Here a more detailed explaination of why LR parsers are important:-

1. Robustness and Efficiency.
2. Handling Complex Languages
3. Implementation.
4. Error Detection
5. Widely Used

Working of LR parser:-

1. Initialization:
- An empty stack is initialized, and a special end-of-input marker ($) is placed at the bottom.
- The start symbol of the grammar is placed on top of the stack.

2. Input Reading:-
The input string is read from left-to right, Symbol by Symbol

3. Shift and Reduce Operations:-

Shift:-
The current Input Symbol is moved from the input string to the top of the stack.

Reduce:-
If the Symbols on top of the Stack match the right-hand side of a production rule, they are replaced with the left-hand side non-terminal.

5.b) Briefly explain equivalence of type expression and type conversion?

a) Type expression equivalence determines when two type descriptions are considered the same, while type conversion changes the data type of an expression.

Type expression equivalence:-

⇒ Structural Equivalence:-
Two types are structurally equivalent if they have the same underlying structure, regardless of the name used for the type. For example, if two types are both defined as "array of integer", they would be structurally equivalent even if one type was named "int-array" and the other was named "integer-list".

Name Equivalence:-
Two types are name equivalent if they have the same name regardless of their underlying structure. For example, if a type called, "my-type" is defined as "array of integer", & another type also named "my-time" is defined elsewhere, they would be name equivalent.

Type Conversion:-
⇒ implicit Conversion:-
This occurs automatically by the compiler during runtime, without explicit instructions from the programmer. For instance, converting an integer to a float might be done implicitly.
⇒ Explicit Conversion:-
This is done by the grammar using a special operator or function to change the type of an expression or variable. For example, inc, (float) i, explicitly converts the integer variable i to a float
⇒ purpose:-
Type conversion is often used to allow code to work with different data types together or to prepare data for specific operations that require a particular type.

2-b) Differences between S-attributes and L-attributed grammars. Discuss their significance in syntax directed translation?

a)	S-attributed SDT	L-attributed SDT
1.	Uses only Synthesized attribute	1. Uses both Synthesized and inherited attribute. inherited attribut can inherit values from either parent or left sibling only.
2.	Semantic actions are placed at right end of production. $A \rightarrow BCD$ $A \rightarrow BCD\{A\}$	2. Semantic actions are placed anywhere on right hand side of production. $A \rightarrow BCD$ $A \rightarrow \{SA\}\ BCD$ $A \rightarrow B\{SA\}\ CD$ $A \rightarrow BC\{SA\}\ D$ $A \rightarrow BCD\{SA\}$

3. Attributes are evaluated bottom up parsing.	3. Attributes are evaluated by travelling parse tree depth first and left to right.
4. No strict rules on attribute dependencies.	4. Some rules limit where attributes can get their values from, especially for inherited attributes.
5. In arithmetic expression, the value of an expression depends on the values of its parts, so it's S-attributed.	5. In type checking, the type of an expression might depend on its context, so it often L-attributed.

Significance in SDT :-

1. Semantic Rules Attachment :

Attributes allow attaching semantic to grammar rules - Critical for tasks like type checking, intermediate code generation etc.

2. Compiler phases :-

S-attributed grammars are often used during code generation stages in bottom-up parsers.

L-attributed grammars are often used in semantic analysis and symbol-table management during top down parsing.

3. Modularity & Maintainability :-

Attribute grammars modularize semantic actions cleanly, aiding in better design and maintenance of compilers.

3. a) Write about function and operator overloading?

An overloaded symbol has different meanings depending on its context. Overloading is resolved when a unique meaning is determined for each occurrence of a name. In this section, we restrict attention to overloading that can be resolved by looking only at the arguments of a function, as in Java.

Example:- The + operator in Java denotes either String Concatenation or addition, depending on the types of its operands. User defined functions can be overloaded as well as in.

```
Void err() {...}
Void err(String S) {...}
```

Note that we can choose between these two versions of a function err by looking at their arguments.

The following is a type-synthesis rule for overloaded functions:

If f can have type $S_i \to t_i$, for $1 \le i \le n$, where $S_i \neq S_j$ for $i \neq j$ and x has type S_k, for some $1 \le k \le n$ then expression f(x) has type t_k.

The value-number method can be applied to type expressions to resolve overloading based on argument types, efficiently.

1) Construct SLR and LALR parsing table for the gramma

$S \to L=R \mid R$
$L \to *R \mid id$
$R \to L$

a) We'll first argument the gramma by adding a new start Symbol:

$S' \to S$

Step 1:- FIRST and FOLLOW Sets

FIRST Sets:-

sions to resolve overloading based on argument types, efficiently.

1) Construct SLR and LALR parsing table for the grammar

S → L=R | R
  → *R | id
L → L

2) We'll first argument the grammar by adding a new start symbol:

S' → S

Step 1:- FIRST and FOLLOW Sets

FIRST Sets :-
FIRST (id) = {id}
FIRST ( ) = { }
FIRST (L) = {*, id}
FIRST (R) = {*, id}
FIRST (S) = {*, id}

FOLLOW Sets :-
FOLLOW (S) = {$}
FOLLOW (L) = {=, $}

FOLLOW(R) = {$, =}

Step 2: I R(0) Items and States (Summary)

I0 (initial Item)

S' → .S
S → .L=R
S → .R
L → .*R
L → .id
R → .L

From here, we construct the DFA using GOTO on terminals and non-terminals (id, *, =, L, R, S), computing Closure for each.

SLR parsing Table:-

State	id	*	=	$	S	L	R
0	S5	S4			1	2	3
1				acc			
2			S6				
3			r2	r2			
4	S5	S4				7	3
5			r4	r4			
6	S5	S4					8
7			r3	r3			
8			r1	r1			

LALR parsing:-

1. FIRST Set:

FIRST (id) = {id}
FIRST (*) = {*}
FIRST (L) = {*, id}
FIRST (R) = {*, id}
FIRST (S) = {*, id}

Follow Set:

FOLLOW (S) = {$}
FOLLOW (L) = {=, $}
FOLLOW (R) = {=, $}

step2: construct canonical LR(1) items
This step involves during building the full LR(1) item sets, which can be large. Instead of listing every detail

1. Creat LR(1) items with lookahead.
2. Build closure and goto functions
3. Merge item sets with same LR(0) cores to get LALR(1) sets.

I0:
[S' → •S, $]
S → •L = R, $
S → •R, $
L → •*, R, =
L → •id, =
R → •L, $

Build, LALR passing table:-

State	id	*	=	$	S	L	R
0	S5	S4			1	2	3
1				acc			
2			S6				
3			r2	r2			
4	S5	S4				7	3
5			r4	r4			
6	S5	S4					8
7			r3	r3			
8			r1	r1			

3.b) Consider the following grammar
S → TL;
T → int | float,
L → L, id | id

TL, id

Stack	input	Action
	int id, id; $	shift int
int	id, id; $	reduced to T → id
T	id, id; $	shift id
T id	, id; $	reduced L → id
TL	, id; $	Shift ,
TL,	id; $	Shift id
TL, id	; $	reduce id by L → id
TL	; $	reduce L → L, id
TL;	$	Shift ;

tom up approach

xt = (a,(a,a))

input string	Action
	shift

Stack	input	Action
S	$	Reduce S → TL;
S'	$	accept.