# ASSIGNMENT 2 - MST2

# Compiler Design

**MST-2**

**1a.** Define closure and go-to operation.

**Closure** is a fundamental concept in formal language theory and computer science. It refers to the process of generating a set of all possible productions that can be derived from a given grammar. In other words, it is the operation of adding all possible productions that can be derived from a grammar to that grammar itself. Closure is an essential step in constructing the LR(1) parser table for a given grammar.

To understand closure, we need to first understand what an item is. An item is a production rule with a dot (.) placed at some position in the right-hand side of the rule. For example, consider the following production rule:

A → BCD

The item set for this rule would be:

A → .BCD

Closure operation takes an item set as input and generates a new set of items by adding all possible productions that can be derived from the grammar. The closure operation is defined as follows:

Given a set of items I, the closure of I denoted as CLOSURE(I), is defined as follows:

1. Add all items in I to CLOSURE(I).
2. For each item A → α.Bβ in CLOSURE(I), where B is a non-terminal symbol, add all items B → .γ to CLOSURE(I), where γ is any string of terminals and non-terminals.
3. Repeat step 2 until no new items can be added to CLOSURE(I).

The closure operation is used in constructing LR(1) parser tables, which are used to parse deterministic context-free languages.

**Go-to** operation is another fundamental concept in formal language theory and computer science. It refers to the process of computing the next state of a parser given the current state and an input symbol. Go-to operation is used in constructing LR(1) parser tables.

Given a LR(1) parser table, the go-to function GOTO(I, X) is defined as follows:

1. Let I be a set of items and X be a terminal or non-terminal symbol.
2. The go-to function GOTO(I, X) returns the closure of the set of items that can be obtained by shifting the dot over X in any item A → α.Xβ in I.

The go-to operation is used in constructing LR(1) parser tables, which are used to parse deterministic context-free languages.

1b. Differentiate between synthesized and inherited attributes.

**Inherited attributes** are properties of a non-terminal symbol in a grammar that are determined by its parent nodes in the parse tree. When a non-terminal symbol is expanded during parsing, its inherited attributes are passed down from its parent nodes to its child nodes. Inherited attributes can be used to propagate information from higher-level constructs to lower-level constructs. For example, consider a grammar rule for arithmetic expressions:
```
expr -> expr + term | term
```
Here, the value of the `expr` non-terminal depends on the values of its child nodes (i.e., the `expr` and `term` symbols). We can define an inherited attribute `val` for `expr` that represents the value of the expression. The value of `val` for an `expr` node is determined by the values of its child nodes and whether it is being added to another expression or not.

**Synthesized attributes**, on the other hand, are properties of a non-terminal symbol that are determined by its child nodes in the parse tree. When a non-terminal symbol is expanded during parsing, its synthesized attributes are computed based on the values of its child nodes. Synthesized attributes can be used to compute information about higher-level constructs from information about lower-level constructs. For example, consider a grammar rule for arithmetic expressions:
```
term -> term * factor | factor
```
Here, the value of the `term` non-terminal depends on the values of its child nodes (i.e., the `term` and `factor` symbols). We can define a synthesized attribute `val` for `term` that represents the value of the term. The value of `val` for a `term` node is computed based on the values of its child nodes and whether it is being multiplied by another term or not.

In summary, inherited attributes are passed down from parent nodes to child nodes, while synthesized attributes are computed based on the values of child nodes. Both types of attributes are used in semantic analysis to compute information about a program's structure and meaning.

**1c**. List the methods of error recovery in syntax analysis.

In syntax analysis, error recovery is the process of detecting and correcting errors in the input program. There are several methods of error recovery in syntax analysis, including:

1. **Panic Mode Recovery:** This is the simplest and most commonly used method of error recovery in syntax analysis. In this method, when an error is detected, the parser discards input symbols until it finds a symbol that can be used as a synchronization point to continue parsing. This synchronization point is usually a delimiter such as a semicolon or a closing brace.

2. **Phrase-Level Recovery:** In this method, the parser tries to identify the smallest phrase that contains the error and then discards the entire phrase. The parser then resumes parsing from the point immediately after the discarded phrase. This method is more complex than panic mode recovery but can recover from errors more accurately.

3. **Error Productions:** In this method, the grammar of the input language is modified to include special productions that can handle common errors. For example, if a missing semicolon is a common error in the input language, an error production can be added to handle this error by inserting a semicolon at the appropriate point in the parse tree.

4. **Global Correction:** In this method, the parser attempts to correct errors globally by making assumptions about what was intended by the programmer. For example, if a variable name is misspelled, the parser may assume that it was intended to be another variable with a similar name and continue parsing.

5. **Backtracking:** In this method, when an error is detected, the parser backtracks to a previous state and tries an alternative path through the parse tree. This method can be very effective but can also be very expensive in terms of time and memory.

In conclusion, there are several methods of error recovery in syntax analysis, each with its own advantages and disadvantages. The choice of which method to use depends on factors such as the complexity of the input language, the frequency and types of errors expected, and the desired trade-off between accuracy and efficiency.

**1d**. Differentiate between parse tree and syntax tree with an example.

A parse tree, also known as a concrete syntax tree, is a tree-like data structure that represents the syntactic structure of a string according to some formal grammar. It shows how the input string can be broken down into smaller pieces until each piece conforms to a grammar rule. Each node in the parse tree represents either a terminal symbol or a non-terminal symbol in the grammar. Terminal symbols are the basic building blocks of the language, such as keywords, identifiers, and operators, while non-terminal symbols represent syntactic categories like expressions, statements, and declarations.

For example, consider the following expression in C++:

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

**2.** Construct LR (0) parsing table for the grammar: S→AA, A →aA/b.

To construct the LR(0) parsing table for the given grammar, we need to follow the below steps:

Step 1: Augment the grammar by adding a new start symbol and production rule as follows:

S' --> S
S --> AA
A --> aA/b

Step 2: Find the set of LR(0) items for each production rule. The set of LR(0) items for a production rule is obtained by considering all possible positions of the dot (.) in the right-hand side of the production rule. For example, the set of LR(0) items for the production rule S --> AA are:

S --> .AA
S --> A.A
A --> .aA
A --> .b

Step 3: Construct the LR(0) automaton by using the sets of LR(0) items obtained in step 2. The LR(0) automaton is a directed graph with nodes representing sets of LR(0) items and edges representing transitions between these sets. The initial state is the set of LR(0) items containing the augmented production rule with the dot at the beginning.

Step 4: Fill in the entries of the LR(0) parsing table using the LR(0) automaton constructed in step 3. Each entry of the parsing table corresponds to a state and a terminal or nonterminal symbol. If there is a shift operation from state i to state j when input symbol X is read, then we put an entry in row i and column X that says "shift j". If there is a reduce operation by production rule R when input symbol X is read, then we put an entry in row i and column X that says "reduce R".

The LR(0) parsing table for the given grammar is as follows:

| State | a | b | $ | A |
|-------|------|------|------|------|
| 0 | s2 | s3 | | s1 |
| 1 | | | acc | |
| 2 | s2 | s3 | | s4 |
| 3 | r2 | r2 | r2 | r2 |
| 4 | | | r1 | |

In the parsing table, each row represents a state of the LR(0) automaton, and each column represents a terminal or nonterminal symbol. The entry in row i and column X represents the action to take when input symbol X is read in state i. The symbols "s" and "r" stand for shift and reduce operations, respectively. The number after "s" or "r" indicates the state or production rule to shift or reduce to.

To parse a given input string using the LR(0) parsing table, we start with an empty stack and the initial state (state 0). Then, we read symbols from the input string one by one and perform the corresponding actions according to the parsing table. If the action is a shift operation, we push the input symbol onto the stack and transition to the next state. If the action is a reduce operation by production rule R, we pop the right-hand side of R from the stack and push the left-hand side of R onto the stack. We then transition to the state indicated by the entry in the parsing table for the current state and left-hand side of R.

**3.** Explain YACC and the different part of a YACC input file.

YACC, which stands for "Yet Another Compiler-Compiler," is a tool that generates a parser based on a formal grammar specification. It is commonly used in the development of programming languages and other applications that require parsing of complex input.

A YACC input file typically consists of three parts:

1. **Definitions**: This section contains declarations for any tokens (i.e., lexical elements) that will be used in the grammar. These are typically defined using regular expressions or literal strings. In addition, this section may also include any necessary C code

that will be included in the generated parser.

2. **Grammar Rules**: This section defines the actual grammar rules for the language being parsed. Each rule consists of a left-hand side (LHS) non-terminal symbol, followed by a colon and a list of right-hand side (RHS) symbols that make up the production. The RHS symbols can be either terminal or non-terminal symbols.

3. **C Code**: This section contains any additional C code that is required for the parser to function correctly. This can include functions for handling semantic actions associated with grammar rules, as well as any necessary global variables or data structures.

In summary, YACC is a tool used to generate parsers based on formal grammar specifications. Its input file typically consists of definitions for tokens, grammar rules, and C code.

**4.** Explain the concept of the Three Address code. Discuss the implementation of the three address statements with an example. Write three codes for the expression: - (a * b) + (c + d) – (a + b + c + d).

The Concept of Three Address Code:

Three Address Code (TAC) is a low-level code representation used in compilers to convert high-level programming languages into machine code. It is a sequence of instructions that can be executed by a computer's CPU. The name "three address code" comes from the fact that each instruction in the code has at most three operands, which can be either constants, variables, or memory addresses.

The main purpose of TAC is to simplify the process of code generation and optimization. It allows compilers to generate efficient machine code by breaking down complex expressions into simpler ones. TAC is also useful for performing optimizations such as constant folding, common subexpression elimination, and dead code elimination.

Implementation of Three Address Statements:

A three address statement consists of an operator and up to three operands. The operator can be any arithmetic, logical, or relational operator. The operands can be either constants, variables, or memory addresses. Here is an example of a three address statement:

```
t1 = a + b
```

In this example, `t1` is a temporary variable that holds the result of adding `a` and `b`. The resulting TAC instruction would look like this:

```
ADD a b t1
```

This instruction tells the CPU to add the values stored in `a` and `b`, and store the result in `t1`.

Example of Three Address Code:

Here are three codes for the expression: - `(a * b) + (c + d) – (a + b + c + d)`.

Code 1:
```
t1 = a * b
t2 = c + d
t3 = t1 + t2
t4 = a + b
t5 = c + d
t6 = t4 + t5
t7 = t3 - t6
```

Code 2:
```
t1 = a * b
t2 = c + d
t3 = a + b
t4 = c + d
t5 = t1 + t2
t6 = t3 + t4
t7 = t5 - t6
```

Code 3:
```
t1 = a * b
t2 = c + d
t3 = a + b + c + d
t4 = t1 + t2
t5 = t3 - t4
```
All three codes represent the same expression `(a * b) + (c + d) – (a + b + c + d)`. The difference lies in the order of operations and the use of temporary variables.

# Compiler Design

**Syllabus-2**

## 1. LL (1) Parser:

LL(1) parser is a type of top-down parsing technique that is used to analyze and validate the syntax of a given context-free grammar. This type of parser is called LL(1) because it reads input from left to right, constructs a leftmost derivation, and uses one lookahead symbol to make parsing decisions. The number 1 in LL(1) represents the number of lookahead symbols that the parser uses to make parsing decisions.

The LL(1) parser works by constructing a parse tree for the input string. It starts with the start symbol of the grammar and attempts to derive the input string by applying production rules. At each step, it looks ahead at the next symbol in the input stream and selects the appropriate production rule based on that lookahead symbol. If there are multiple possible production rules for a given lookahead symbol, then the parser uses a set of rules called the FIRST and FOLLOW sets to determine which rule to apply.

The FIRST set of a non-terminal symbol is the set of terminal symbols that can appear as the first symbol in any string that can be derived from that non-terminal. The FOLLOW set of a non-terminal symbol is the set of terminal symbols that can appear immediately after that non-terminal in any valid derivation. By using these sets, the LL(1) parser can determine which production rule to apply based on the current non-terminal and lookahead symbol.

One example of an LL(1) grammar is:

```
S -> A | B
A -> aBa | ε
B -> bAb | ε
```

This grammar generates strings that consist of any combination of `a`s and `b`s, where each `a` is surrounded by `b`s and vice versa. The empty string `ε` is also allowed.

To parse an input string using this grammar, the LL(1) parser would start with the start symbol `S` and lookahead at the first symbol in the input stream. If the lookahead symbol is `a`, then the parser selects the production rule `S -> A`, since `A` can derive strings that start with `a`. The parser then applies the production rule `A -> aBa` and derives the string `aBa`. The lookahead symbol is now `b`, so the parser selects the production rule `B -> bAb` and derives the string `abAba`. Finally, the parser applies the production rule `A -> ε` to derive the string `aba`, which is a valid derivation of the input string.

**2. Shift reduce parser** :

A shift-reduce parser is a type of bottom-up parsing algorithm that operates by shifting tokens onto a stack and then reducing them to larger constituents according to a set of production rules. This type of parser is commonly used in computer science for compiling programming languages, as well as in natural language processing for analyzing human language.

The shift-reduce parser begins by placing the input string onto a stack and initializing a state machine. The parser then reads the input one token at a time and performs either a shift or reduce operation based on the current state of the machine and the grammar rules.

A shift operation involves moving the next input token onto the top of the stack, while a reduce operation involves popping one or more items off the stack and replacing them with a larger constituent based on a grammar rule. The parser continues to shift and reduce tokens until it reaches the end of the input string and reduces all remaining items on the stack to a single constituent, typically the start symbol of the grammar.

**3. LR Parser**

LR parser is a type of bottom-up parser that can handle a large class of context-free grammars. The LR stands for Left-to-right Rightmost derivation, which means that it scans the input from left to right and constructs a rightmost derivation in reverse order. The LR parser uses a deterministic finite automaton called the LR(0) automaton to parse the input string. It reads the input symbols one by one and shifts them onto a stack until it can reduce them to a non-terminal symbol using a production rule.

**4. LR (0) Parser:**

LR(0) parser is a type of bottom-up parsing technique used in computer science to analyze and process input strings according to a given grammar. It is called an LR(0) parser because it uses a left-to-right scan of the input string, constructs a rightmost derivation of the input string, and uses zero lookahead symbols to make parsing decisions.

The LR(0) parser works by building a deterministic finite automaton (DFA) called an LR(0) automaton from the given grammar. The states of the automaton represent sets of items, where each item is a production rule with a dot at some position indicating how much of the rule has been parsed so far. The transitions between states are determined by shifting the dot to the right or reducing the rule if all its symbols have been parsed.

**5. SLR Parsing table**

SLR Parsing Table, also known as Simple LR Parsing Table, is a table-based parsing technique used for analyzing the syntax of a programming language. It is a bottom-up parsing method that uses a stack to keep track of the grammar rules and the input symbols.

The SLR Parsing Table is generated by constructing an LR(0) automaton for the given grammar. The automaton has states that represent the set of items, where each item is a production rule with a dot at some position in the right-hand side of the rule. The dot represents the current position of the parser in the production rule. The automaton transitions from one state to another based on the next input symbol.

**6. Intermediate code generation**

Intermediate code generation is a process in compiler design that involves the translation of source code into an intermediate representation or code. This intermediate code is designed to be machine-independent and can be easily translated into the target machine code. Intermediate code generation is an important step in the compilation process as it helps in optimizing the code and making it more efficient.

There are different types of intermediate code generation techniques used in compiler design, including:
1. Three-Address Code (TAC): TAC is a type of intermediate code that represents each statement in three parts - operator, operand 1, and operand 2. For example, the TAC representation of the expression "a = b + c" would be "t1 = b + c" and "a = t1". Here, t1 is a temporary variable used to hold the result of the addition operation.

2. Quadruples: Quadruples are another type of intermediate code that represent each statement using four fields - operator,

operand 1, operand 2, and result. For example, the quadruple representation of the expression "a = b + c" would be "+ b c t1" and "= t1 a _".

3. Abstract Syntax Tree (AST): AST is a hierarchical representation of the source code that captures its structure and semantics. It is often used as an intermediate representation in compilers for object-oriented programming languages such as Java and C++.

**7. Syntax directed definition and translations**

A syntax-directed definition (SDD) is a formalism used in computer science to specify the attributes of the nodes in a parse tree. It associates a set of attributes with each grammar symbol and specifies how these attributes are computed from the attributes of the symbol's children in the parse tree. SDDs are used to define programming language semantics, code generation, and optimization.

There are two types of SDD translations: inherited and synthesized. Inherited translations propagate information from parent nodes to child nodes, while synthesized translations propagate information from child nodes to parent nodes.

An example of an inherited translation is type checking in a programming language. The type of an expression is determined by the types of its operands and the operator being applied. The type attribute is inherited from the parent node to the child nodes.

An example of a synthesized translation is code generation for expressions in a programming language. The code for an expression is generated by combining the code for its operands and applying the operator. The code attribute is synthesized from the child nodes to the parent node.

**8. Syntax tree (allocated tree)**

A syntax tree, also known as an allocated tree, is a graphical representation of the structure of a sentence in a language. It is used in linguistics to analyze and understand the grammatical structure of sentences. The tree is composed of nodes, which represent words or groups of words, and edges, which represent the relationships between them.

The type of syntax tree depends on the type of grammar used to generate it. The two main types of grammars are constituency grammars and dependency grammars. Constituency grammars generate trees based on the hierarchical structure of phrases in a sentence, while dependency grammars generate trees based on the relationships between individual words in a sentence.

Here is an example of a constituency syntax tree for the sentence "The cat sat on the mat":

```
S
/ \
NP VP
/ / \
Det V PP
| | |
The cat on
| |
sat the mat
```

In this tree, S represents the sentence as a whole, NP represents the noun phrase "the cat", VP represents the verb phrase "sat on the mat", Det represents the determiner "the", V represents the verb "sat", and PP represents the prepositional phrase "on the mat".

Some examples of authoritative reference publications or domain names for syntax trees include:

1. "Syntactic Structures" by Noam Chomsky - This book is considered a foundational work in generative grammar and introduced many concepts that are still used today, including phrase structure rules and transformational rules.

2. Stanford Parser - This is a popular natural language processing tool that uses probabilistic context-free grammars to generate syntax trees for sentences.

3. Linguistic Data Consortium - This organization collects and distributes linguistic data for research purposes, including annotated corpora with syntax trees for various languages.

## 9. Three address code

Three-address code is a type of intermediate code used in compilers to represent the code in a form that is easier to analyze and optimize. It is called "three-address" because each instruction in the code contains up to three memory addresses or operands.

The operands are typically either variables or constants, and the instructions manipulate the values stored in these operands. The three-address code is designed to be simple and easy to understand, which makes it easier for the compiler to generate efficient machine code.

An example of three-address code might look like this:

```
t1 = a + b
t2 = c * d
t3 = t1 - t2
e = t3
```

In this example, `a`, `b`, `c`, and `d` are variables, and `e` is the result of the computation. The first instruction adds `a` and `b` together and stores the result in a temporary variable `t1`. The second instruction multiplies `c` and `d` together and stores the result in another temporary variable `t2`. The third instruction subtracts `t2` from `t1` and stores the result in yet another temporary variable `t3`. Finally, the fourth instruction assigns the value of `t3` to the variable `e`.