**Applications of TOC principles in compiler designing**

**Introduction**

**THE MANASI**

Theory of Computation is also known as Automata theory, it is a theoretical field of Computer Science and Mathematics which focuses on the logic of computation with respect to basic machines known as automata.

Chaitanya

Automata enables scientists to understand how machines compute the functions and solve problems. The main motivation behind developing Automata Theory was to develop methods to describe and analyse the dynamic behaviour of discrete systems.

Uma

Automata is originated from the word “Automaton” which is closely related to “Automation”.

In theoretical computer science, the theory of computation is the branch that deals with whether and how efficiently problems can be solved on a model of computation, using an algorithm.

Bhavin

The Automatic consists of states and transitions the state is represented by circles and the transition is represented by arrows. Automata is a kind of machine which takes some string as input and this input goes through a finite number of states and may enter to the final state.

**THE MANASI**

Yes, absolutely right. Up till now we have discussed how automata works and how it contributes in the field of computation or we can say we have discussed some of the important principles of the TOC Right!

Now question arises how this theory of computation is applicable in compiler construction

Chaitanya

Yeah, before that I would like to a point here that, Compilers are one of the richest destinations of theory of computation concepts. Maybe it would not be wrong to say that a lot of developments in theory of computation have happened from the need and wish to have good programming languages and systems to process them (e.g., compilers, interpreters and runtime systems).

Uma

**Yes right, now we let us discuss the phases of Compiler Design so that it will be easy to understand how automata theory applicable in the compiler phases.**

**Compiler** operates in various phases each phase transforms the source program from one representation to another. Every phase takes inputs from its previous stage and feeds its output to the next phase of the compiler. There are 6 phases in a compiler. Each of this phase help in converting the high-level langue the machine code.

Bhavin

These phases are-

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate code generator
5. Code optimizer
6. Code generator

**THE MANASI**

Let us start from lexical analysis

Lexical Analysis is the first phase when compiler scans the source code. This process can be left to right, character by character, and group these characters into tokens.  
Here, the character stream from the source program is grouped in meaningful sequences by identifying the tokens. It makes the entry of the corresponding tickets into the symbol table and passes that token to next phase.

Chaitanya

And this is when the Finite automata comes into the picture. The task of lexical analysis as Manasi has mentioned is performed by the Finite automata.

Finite automata is a state machine that takes a string of symbols as input and changes its state accordingly. Finite automata is a recognizer for regular expressions. When a regular expression string is fed into finite automata, it changes its state for each literal. If the input string is successfully processed and the automata reaches its final state, it is accepted, i.e., the string just fed and this can be said to be a valid token of the language in hand.

Uma

Exactly it is. Now the output of this phase is then passed to the syntax analysis of the compiler.

And Syntax analysis is all about discovering structure in code. It determines whether or not a text follows the expected format. The main aim of this phase is to make sure that the source code was written by the programmer is correct or not.

Bhavin

Yeah, just like grammar in the English language as it checks whether the particular statement is grammatically correct or not, but using some grammar rules.

**THE MANASI**

Exactly. In Syntax analysis is based on the rules based on the specific programming language by constructing the parse tree with the help of tokens generated by Finite automata in lexical analysis. It also determines the structure of source language and grammar or syntax of the language.  
For example, it performs the tasks like-

Obtain tokens from the lexical analyser

Bhavin Checks if the expression is syntactically correct or not

Chaitanya Report all syntax errors

Uma Construct a hierarchical structure which is known as a parse tree

**THE MANASI**

Precisely guys! Now do you know according to Chomsky hierarchy, grammars is divided into 4 types:

Bhavin Type 0 which is known as unrestricted grammar.

Chaitanya Type 1 which is known as context sensitive grammar.

Uma Type 2 which is known as context free grammar.

**THE MANASI** Type 3 which is Regular Grammar.

Bhavin

Now let us see more details of this

**First is Type 0 i.e., Unrestricted Grammar:**

Which is also known as formal grammar. The language generated by this grammar is called as Recursively Enumerable languages. Which is recognize by the Turing Machine

--------------------------------------------------------------------------------------------

Abstract machines: decider

In [computability theory](https://en.wikipedia.org/wiki/Computability_theory), a **machine that always halts**, also called a **decider**[[](https://en.wikipedia.org/wiki/Machine_that_always_halts#cite_note-1)

---------------------------------------------------------------------------------------------------

Chaitanya

**Second is the Type 1 grammar which is also called as Context Sensitive Grammar**

Type-1 grammars generate the context-sensitive languages. The language generated by the grammar are recognized by the Linear Bound Automata

In Type 1 of all Type 1 grammar should be Type 0.

Uma

**Third is the Type 2 grammar which is Context Free Grammar:**

Type-2 grammars generate the context-free languages. The language generated by the grammar is recognized by a Pushdown automaton.

In Type 2,

1. First of all it should be Type 1.

2. Left hand side of production can have only one variable.

**THE MANASI**

**And last but not least is the Type 3 Regular Grammar:**

Type-3 grammars generate regular languages. These languages are exactly all languages that can be accepted by a finite state automaton.

Type 3 is most restricted form of grammar.

Bhavin

Yeah so, we can say that in the syntax analysis which is the second phase in compiler design we use Turing machine, Linear Bounded Automata, Push down automata and Finite Automata respectively for each type of the Grammar.

**All say YES/ Exactly/ Yep/ RIGHT**

Uma

Further down, a variety of concepts from theory of computation are heavily used in compilers as we have discussed. Now the problems of language processing and formal verification are rife with NP-hard (e.g., register allocation, instruction scheduling etc.) and undecidable problems (e.g., property proving on higher logics and calculi, certain types of type-checking problems, dead code elimination, inter-procedural points-to analysis etc.).

Chaitanya

All these may sound a bit exotic or unknown to us. However, people who have developed the theory of compilers and programming languages have had to deal with these questions on the ground.

**THE MANASI**

Theory of computation therefore provides all the foundations to determine what and how far a compiler can aspire to do when trying to generate optimised code. We can even say TOC is a definite prerequisite for learning compilers

Chaitanya

Conclusion:

* Finite Automata are used two of the three front-end phases of the compiler.
* Context Free Grammar is used in second phase (PDA)
* Context Sensitive Grammar in second phase of compiler (LBA)
* Unrestricted Grammar in second phase of compiler (TM)

## NOTE for next phase of

* Compiler operates in various phases each phase transforms the source program from one representation to another
* Six phases of [compiler design](https://www.guru99.com/compiler-tutorial.html) are 1) Lexical analysis 2) Syntax analysis 3) Semantic analysis 4) Intermediate code generator 5) Code optimizer 6) Code Generator
* Lexical Analysis is the first phase when compiler scans the source code
* Syntax analysis is all about discovering structure in text
* Semantic analysis checks the semantic consistency of the code
* Once the semantic analysis phase is over the compiler, generate intermediate code for the target machine
* Code optimization phase removes unnecessary code line and arranges the sequence of statements
* Code generation phase gets inputs from code optimization phase and produces the page code or object code as a result
* A symbol table contains a record for each identifier with fields for the attributes of the identifier
* Error handling routine handles error and reports during many phases