

Contents

5 Turing Machine and Other Phases of Compiler	3
5.1 Introduction	3
5.2 Design Turing Machine	7
5.3 Different Types or Variants of Turing Machine.....	9
5.4 Introduction to Undecidability.....	13
5.4.1 Halting problem in Turing machine	14
5.4.2 Inverted Halting Problem.....	15
5.4.3 Recursive and Recursively Enumerable Languages	16
6 Other Phases of Compiler	19
6.1 Syntax Directed Translation	19
6.1.2 Syntax Directed Definition	21
6.1.3 Evaluation Order Of Syntax Direct Definition	23
6.2 Intermediate Code generation	24
6.2.1 Variants of Syntax Trees.....	25
6.2.2 Three Address Code	30
6.3 Code Generation	31
6.3.1 Issues in the Design of a Code Generator	31

Module 5

Introduction to Turing Machine: problems that computers cannot solve, The Turing machine problems, Programming Techniques for Turing machine, Extensions to the Basic Turing machine

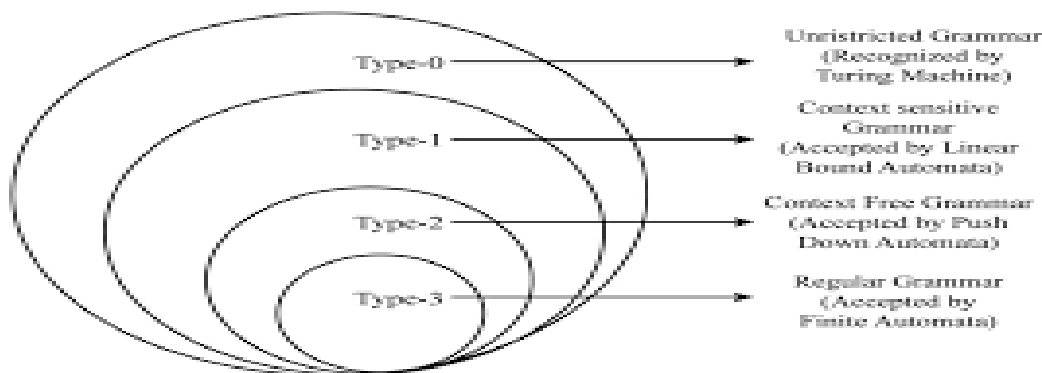
Undecidability: A language That is not Recursively Enumerable, An undecidable Problem That is RE.

Other phases of Compilers: Syntax Directed Translation- Syntax Directed Definitions, Evaluation of orders of SDD's,

Intermediate Code Generation-Variants of Syntax Trees, Three Address Code

Code Generation -Issues in the Design of a Code Generator

5 Turing Machine and Other Phases of Compiler



5.1 Introduction

Define Turing machine and Explain working of Turing machine

A Turing machine is a theoretical model of computation introduced by Alan Turing in 1936. It serves as a fundamental concept in computer science and mathematics for understanding the limits and capabilities of computation. **A Turing Machine is an accepting device which accepts the languages (recursively enumerable set) generated by type 0 grammars**

Definition

A Turing Machine (TM) is a mathematical model which consists of an infinite length tape divided into cells on which input is given. It consists of a head which reads the input tape. A state register stores the state of the Turing machine. After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left. If the TM reaches the final state, the input string is accepted, otherwise rejected.

A Turing Machine can be formally described as a 7-tuple $(Q, X, \Sigma, \delta, q_0, B, F)$ where –

- Q is a finite set of states
- X is the tape alphabet
- Σ is the input alphabet
- δ is a transition function; $\delta : Q \times X \rightarrow Q \times X \times \{\text{Left_shift}, \text{Right_shift}\}$.
- q_0 is the initial state
- B is the blank symbol
- F is the set of final states

At its core, a Turing machine consists of the following components:

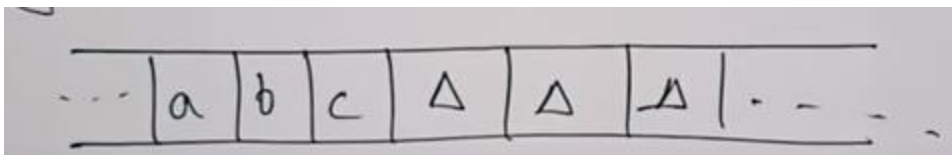
- **Tape:** An infinite tape divided into cells, where each cell can contain a symbol from a finite alphabet. The tape extends infinitely in both directions.
- **Head:** A read/write head that can move left or right along the tape, reading the symbol at the current cell and writing a new symbol if necessary.

- **State Register:** The Turing machine has a finite set of internal states, only one of which is active at any given time.
- **Transition Function:** A set of rules that determine the machine's behavior based on its current state and the symbol it reads from the tape. These rules dictate what symbol to write, which direction to move the head, and which state to transition to next.

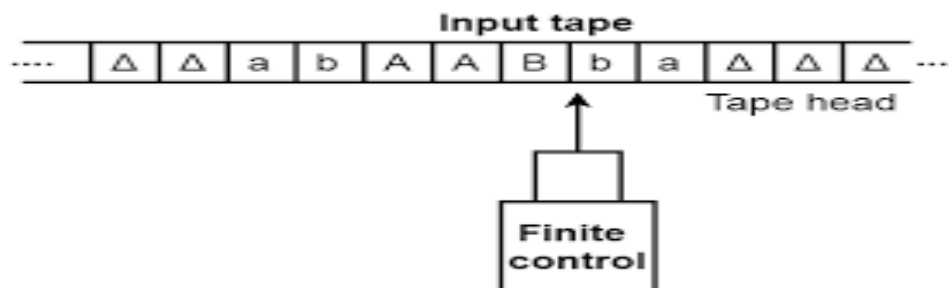
Basic Model or working of Turing Machine:

- The input tape is having infinite no. of cells, each cell containing one input symbol. The empty tape is filled by blank characters

Input Tape



- The Finite control and the tape head which is responsible for reading the current input symbols. The tape head can move from left to right or right to left
- The finite set of states through which machine has to undergo
- The finite set of symbols called external symbols which are used in building the logic of Turing machine



Initialization: The Turing machine begins in a predefined initial state with its head positioned on the initial cell of the tape, which contains the input string.

Execution: At each step, the machine consults the transition function to determine the action to take based on its current state and the symbol it reads from the tape. It updates the tape, moves the head, and transitions to a new state according to the transition rules.

Halt: The machine halts when it reaches a special halting state, indicating that the computation is complete. At this point, the output of the computation is whatever is written on the tape.

The Turing machine is capable of simulating any algorithmic process, making it a foundational concept in theoretical computer science. Its simplicity allows researchers to reason about the properties of algorithms and computability.

Operation on the Tape :

- Read or scan the symbol below the tape head
- Update/write a symbol below the tape head
- Move the tape head one step left
- Move the tape one step right

The language accepted by Turing Machine

- The TM accepts all the all the language even-though they are recursively enumerable
- Recursive means repeating the same set of rules any no. of times
- Enumerable means list of elements
- TM Also acceptable the computable functions, such as addition, subtractions, multiplications and divisions so on

Why Turing Machine is important:

- **Theoretical Foundation:** It provides a theoretical foundation for understanding the concept of computation. By defining a simple yet powerful model of computation, Turing Machines allow researchers to explore the fundamental properties of algorithms and computability.
- **Universal Computability:** The Turing Machine concept is central to the Church-Turing thesis, which suggests that any algorithmic process that can be computed by any mechanical means can be computed by a Turing Machine. This thesis underpins our understanding of what is computable and what is not.
- **Limits of Computation:** Turing Machines help delineate the boundaries of what can and cannot be computed. By studying the capabilities and limitations of Turing Machines, researchers can identify problems that are solvable and those that are not, leading to insights into complexity theory and the classification of computational problems.
- **Basis for Computer Science:** Turing Machines serve as a foundational concept in computer science education. They provide a theoretical framework for understanding algorithms, data structures, and computational complexity, which are essential components of modern computer science curricula.
- **Tool for Analysis:** Turing Machines are used as a tool for analyzing the complexity of algorithms and problems. By modeling algorithms as Turing Machines, researchers can study their time and space complexity, helping to optimize algorithms and improve their efficiency.
- **Philosophical Implications:** The concept of the Turing Machine has profound philosophical implications regarding the nature of mind, intelligence, and the limits of human understanding. It has sparked debates and discussions about artificial intelligence, consciousness, and the nature of computation.

In summary,

the Turing Machine is important because it forms the basis of our understanding of computation, provides a framework for analyzing algorithms and complexity, and has far-reaching implications for computer science, philosophy, and our understanding of the universe.

Applications of Turing Machine:

- **Algorithm Design and Analysis:** Turing machines provide a theoretical framework for designing and analyzing algorithms. Many algorithms and data structures are developed and evaluated based on their behavior when simulated on a Turing machine model. This helps computer scientists understand the efficiency and correctness of algorithms.

Example: Consider the development of sorting algorithms like Quicksort or Merge Sort. While these algorithms may not be directly implemented as Turing machines, their efficiency and correctness can be analyzed using Turing machine concepts. Understanding how these algorithms behave when processing input data and how their performance scales with the size of the input is essential for algorithm design and analysis.

- **Programming Language Theory:** Turing machines are used in the study of programming languages and compilers. Concepts such as computability and decidability are essential for understanding the capabilities and limitations of programming languages. Turing machines help formalize these concepts and provide a basis for language design.

Example: Compilers translate high-level code into machine-readable instructions. While compilers themselves may not be Turing machines, the concepts of computability and language expressiveness, which are fundamental to programming language theory, are grounded in Turing machine concepts. Understanding the capabilities and limitations of programming languages often involves reasoning about their theoretical foundations, including Turing machine theory.

- **Automata Theory:** Turing machines are a central concept in automata theory, which studies abstract machines and their computational capabilities. Automata theory is used in various fields, including compiler design, natural language processing, and artificial intelligence.

Example: Finite state machines (FSMs) are used in various applications, such as traffic light controllers or vending machines. While FSMs are a simplified form of Turing machines, they are analyzed using similar principles. For instance, a vending machine may transition between states based on user inputs, similar to how a Turing machine transitions between states based on input symbols.

- **Formal Verification:** Turing machines are employed in formal methods for verifying the correctness of software and hardware systems. Formal verification techniques use mathematical models, such as Turing machines, to rigorously analyze system behavior and ensure that systems meet their specifications.

Example: In the development of safety-critical software, formal verification techniques ensure that the software behaves correctly under all conditions. Model checking is a formal verification method where a system model (which can be represented using Turing machine concepts) is systematically analyzed to check if certain properties hold. For example, a model of an aircraft control system can be verified to ensure that it always maintains safe flight conditions.

- **Cryptography:** Turing machines play a role in theoretical cryptography, particularly in the analysis of cryptographic protocols and algorithms. Cryptographers use computational models like Turing machines to reason about the security properties of cryptographic systems and to prove theorems about their behavior.

Example: Cryptographic protocols and algorithms rely on mathematical principles that are grounded in computational complexity theory, which is based on Turing machine concepts. For instance, the security of the RSA encryption algorithm relies on the difficulty of factoring large prime numbers, which is believed to be a computationally hard problem according to Turing machine-based complexity theory.

- **Complexity Theory:** Turing machines are fundamental to complexity theory, which classifies computational problems based on their inherent difficulty. The concept of Turing machines helps classify problems into complexity classes, such as P, NP, and NP-complete, providing insights into the solvability and complexity of computational tasks.

Example: The traveling salesman problem (TSP) is a classic computational problem that involves finding the shortest possible route that visits a set of cities exactly once and returns to the starting city. Complexity theory, based on Turing machine concepts, helps classify problems like TSP into complexity classes (e.g., NP-complete) based on their computational difficulty and scalability.

- **Artificial Intelligence:** While not directly applied in AI systems, Turing machines have influenced the theoretical foundations of artificial intelligence. The Turing test, proposed by Alan Turing, is a thought experiment to evaluate a machine's ability to exhibit intelligent behavior indistinguishable from that of a human.

Example: While not directly implementing Turing machine concepts, artificial intelligence (AI) algorithms are developed and evaluated based on theoretical models of computation, including Turing machines. Natural language processing (NLP) systems, for example, use algorithms to analyze and understand human language, and their performance can be analyzed in terms of computational complexity and efficiency, which are grounded in Turing machine theory.

5.2 Design Turing Machine

1. Construct a Turing Machine which accepts the language of “aba” over

<https://www.youtube.com/watch?v=V36tLHdAs20>

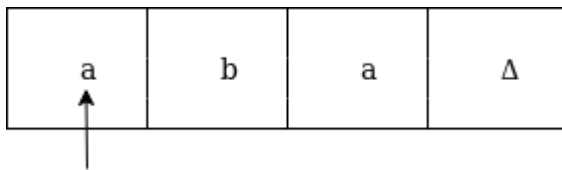
We will assume that on input tape the string 'aba' is placed like this:

We will assume that on input tape the string 'aba' is placed like this:

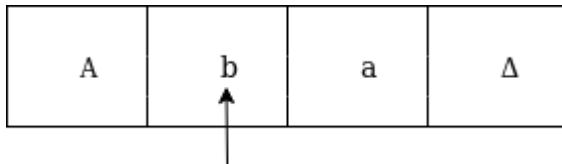
a	b	a	Δ	-----
---	---	---	---	-------

The tape head will read out the sequence up to the Δ characters. If the tape head is readout 'aba' string then TM will halt after reading Δ.

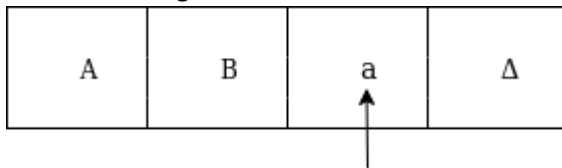
Now, we will see how this turing machine will work for aba. Initially, state is q₀ and head points to a as:



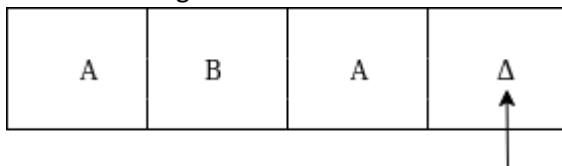
The move will be $\delta(q_0, a) = \delta(q_1, A, R)$ which means it will go to state q_1 , replaced a by A and head will move to right as:



The move will be $\delta(q_1, b) = \delta(q_2, B, R)$ which means it will go to state q_2 , replaced b by B and head will move to right as:



The move will be $\delta(q_2, a) = \delta(q_3, A, R)$ which means it will go to state q_3 , replaced a by A and head will move to right as:



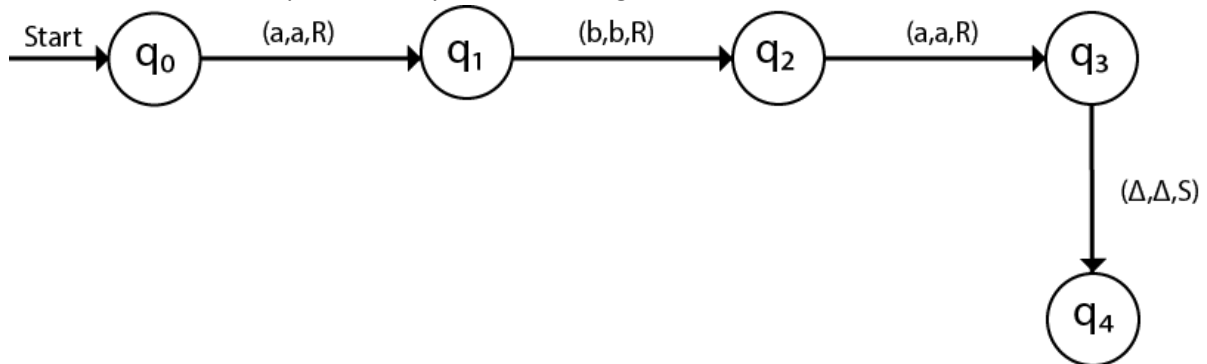
The move $\delta(q_3, \Delta) = (q_4, \Delta, S)$ which means it will go to state q_4 which is the HALT state and HALT state is always an accept state for any TM.

The same TM can be represented by Transition Table:

States	a	b	Δ
q_0	(q_1, A, R)	—	—
q_1	—	(q_2, B, R)	—
q_2	(q_3, A, R)	—	—

q3	–	–	(q4, Δ , S)
q4	–	–	–

The same TM can be represented by Transition Diagram:



2. Design Turing machine which recognize the language $L=0^*1^*0$

<https://www.youtube.com/watch?v=uSpeMO52Wxo>

Q: Design Turing machine to accept the language $L=\{0^n1^n2^n/n \geq 0\}$.

https://www.youtube.com/watch?v=ast_i508wmk

5.3 Different Types or Variants of Turing Machine

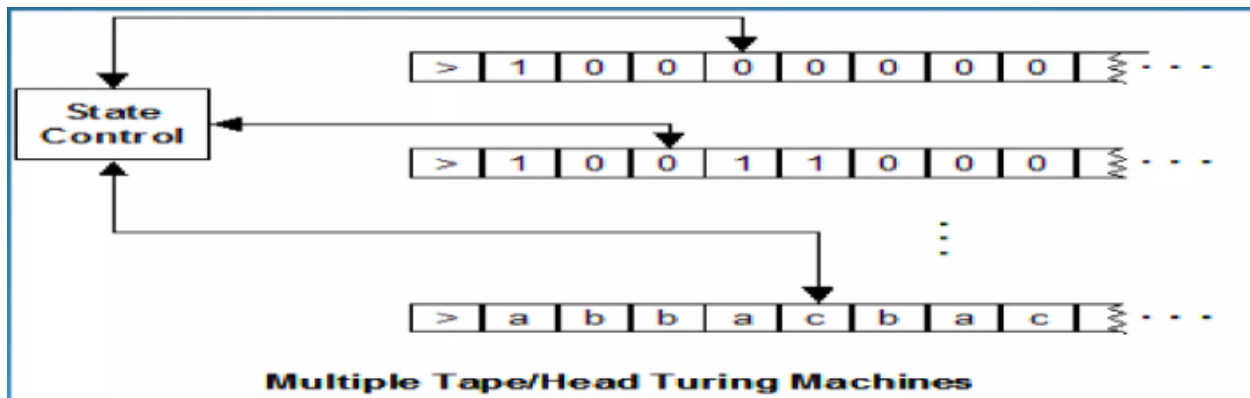
Turing machines are theoretical models of computation that consist of a tape, a read/write head, and a finite set of states. There are several variants and extensions of Turing machines, each with its own set of characteristics and capabilities. Some common variants of Turing machines include:

- Multi-tape Turing machine
- Multi-head Turing machine
- Non-deterministic Turing machine

Multi-tape Turing Machine:

Characteristics:

- Multi-tape Turing machines have multiple tapes, each with its own read/write head.
- Each tape operates independently but can communicate with other tapes during computation.
- The transition function of a multi-tape Turing machine is extended to accommodate multiple tapes.



Steps:

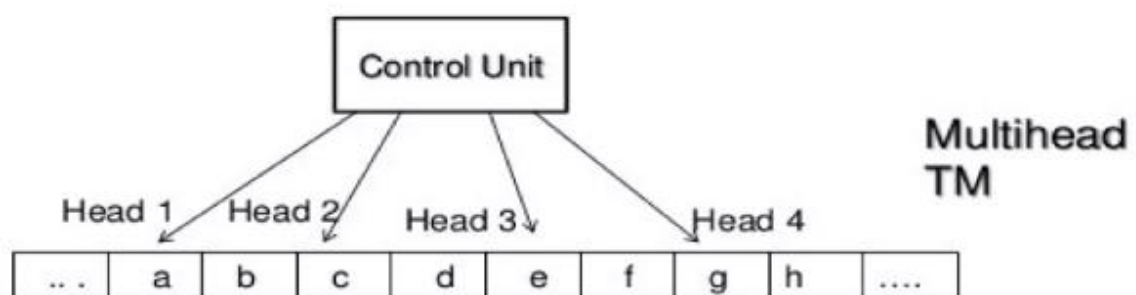
- **Initialization:** Initialize the multi-tape Turing machine by specifying the initial state, tape contents, and initial positions of each tape head.
- **Transition Function:** Define the transition function, which specifies how the machine transitions between states based on the symbols read from each tape.
- **Computation:** Execute the transition function iteratively, moving the tape heads and updating the tape contents according to the specified transitions.
- **Halting Condition:** Determine if the machine halts by reaching a halting state or if no valid transition is available.
- **Output:** Output the final tape contents or any other relevant information.

Example: Multiplying two binary numbers using a multi-tape Turing machine. Each tape represents one binary number, and the machine performs binary multiplication by repeatedly adding and shifting bits.

Multi-head Turing machine:


Characteristics:

- Multi-head Turing machines have multiple read/write heads operating on a single tape.
- Each head can independently read, write, or move left or right on the tape.
- The transition function of a multi-head Turing machine accounts for the actions of multiple heads.



1 st track	1	B	B	B	B	B	B	B	..
2 nd track	B	B	1	B	B	B	B	B	..
3 rd track	..	B	B	B	B	1	B	B	B	..
4 th track	..	B	B	B	B	B	B	1	B	.
5 th track	..	a	b	c	d	e	f	g	h	.

**Multi track
TM**



Steps:

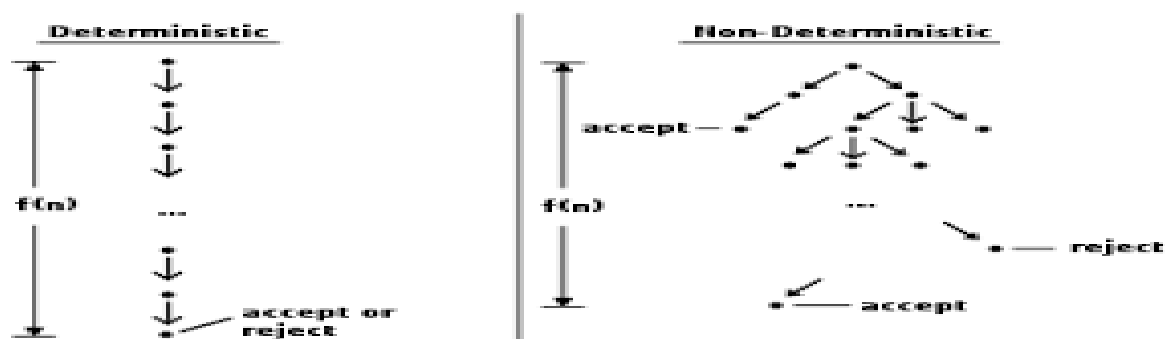
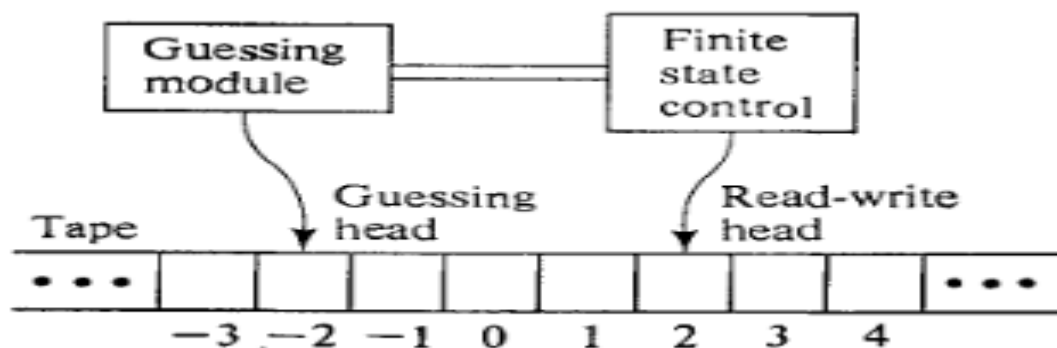
- Initialization: Initialize the multi-head Turing machine by specifying the initial state, tape contents, and initial positions of each head.
- Transition Function: Define the transition function, which specifies how the machine transitions between states based on the symbols read by each head.
- Computation: Execute the transition function iteratively, moving each head and updating the tape contents according to the specified transitions.
- Halting Condition: Determine if the machine halts by reaching a halting state or if no valid transition is available.
- Output: Output the final tape contents or any other relevant information.

Example: Recognizing a palindrome using a multi-head Turing machine. Multiple heads scan the input string simultaneously from both ends, and the machine halts if the string is a palindrome

Non-Deterministic Turing machine:

Characteristics:

- Non-deterministic Turing machines can have multiple possible transitions from a given state on a given input symbol.
- They explore all possible computation paths simultaneously rather than following a single deterministic path.
- Non-deterministic Turing machines accept if any of the possible computation paths lead to an accepting state.



Steps:

- Initialization: Initialize the non-deterministic Turing machine by specifying the initial state and tape contents.
- Transition Function: Define the transition function, allowing multiple possible transitions from each state on each input symbol.
- Computation: Explore all possible computation paths simultaneously, branching out to follow each possible transition.
- Halting Condition: Determine if the machine halts by reaching an accepting state on any computation path.
- Output: Accept or reject the input based on whether any computation path leads to an accepting state.

Example: Recognizing the language $\{0^n 1^n \mid n \geq 0\}$ using a non-deterministic Turing machine. The machine guesses the midpoint of the input string and verifies that the number of 0s matches the number of 1s on both sides.

Problems that computers cannot solve:

The Turing machine is a theoretical model of computation introduced by Alan Turing in 1936. It consists of a tape, a read/write head, a finite set of states, and a transition function. While Turing machines serve as a foundation for understanding the capabilities and limitations of computation, they also highlight certain problems that computers cannot solve.

Example:

Halting Problem: Given a description of a Turing machine and its input, determine whether the machine will eventually halt (stop) or continue running indefinitely. Alan Turing proved that no algorithm can solve the halting problem for all possible inputs and Turing machines. This means that there are programs for which it is impossible to determine whether they will terminate or run forever.

Consider a Turing machine that simulates a program that searches for solutions to the famous mathematical problem known as the "Goldbach conjecture." The Goldbach conjecture states that every even integer greater than 2 can be expressed as the sum of two prime numbers. The program takes an even integer

n as input and attempts to find two prime numbers that sum up to n . It starts by searching for prime numbers, then exhaustively checks all possible pairs to see if their sum equals n . If a solution is found, the machine halts; otherwise, it continues searching indefinitely.

Scenario: Now, let's suppose we have a Turing machine that analyzes other Turing machines to determine whether they halt on a given input. We feed this analyzing machine the description of the Goldbach conjecture-solving program and an even integer n as input. The analyzing machine attempts to determine whether the Goldbach conjecture-solving program halts on the input n . If it does, the analyzing machine halts and outputs "halts"; otherwise, it continues analyzing indefinitely.

Dilemma: Here's the dilemma: If the analyzing machine outputs "halts," it implies that the Goldbach conjecture-solving program halts on input n . However, if the analyzing machine does not halt, it could be because the Goldbach conjecture-solving program either runs indefinitely (if n violates the conjecture) or halts but takes an extremely long time to do so. Therefore, we cannot rely on the output of the analyzing machine to accurately determine whether the Goldbach conjecture-solving program halts on input n .

Conclusion: This example demonstrates the fundamental challenge posed by the halting problem: even with sophisticated analysis tools, we cannot reliably determine whether a given program halts or runs indefinitely on all possible inputs.

5.4 Introduction to Undecidability

Undecidability is a concept within the realm of computer science and mathematical logic that deals with the limitations of computation and problem-solving. At its core, undecidability asserts that there are certain problems for which it is impossible to construct an algorithm or a computational procedure that will always correctly determine whether a given input meets a specified criterion or not.

The notion of undecidability was famously introduced by the **mathematician and logician Kurt Gödel in the early 20th century through his incompleteness theorems**. These theorems showed that within any formal system expressive enough to encompass arithmetic, there exist statements that cannot be proven

true or false within that system. This fundamental limitation applies not only to arithmetic but also to many other formal systems, including those used in computer science.

In computer science, undecidability manifests itself through problems such as the Halting Problem, first formulated by Alan Turing in the 1930s. The Halting Problem asks whether it is possible to write a program that, given any other program and its input, can determine whether the program will eventually halt (terminate) or continue running indefinitely. Turing's proof demonstrated that no such algorithm exists.

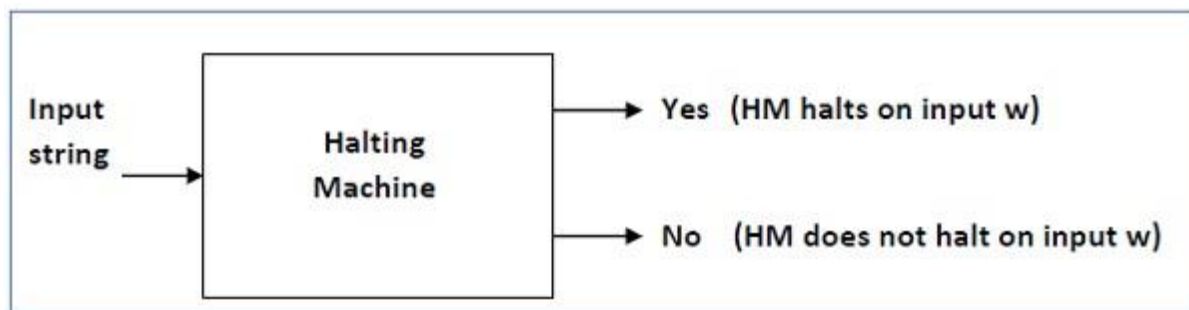
Undecidability has profound implications for theoretical computer science and the study of computation. It sets fundamental limits on what computers can and cannot do, highlighting the existence of problems that are inherently beyond the reach of algorithmic solutions. This concept is foundational to fields such as computational complexity theory, formal languages, and automata theory, shaping our understanding of the theoretical boundaries of computation.

5.4.1 Halting problem in Turing machine

The Halting Problem is a fundamental concept in computer science, particularly in the realm of computability theory, introduced by Alan Turing in 1936. It deals with the question of whether a given algorithm (or program) can determine whether another algorithm, when given a specific input, will eventually halt (terminate) or run indefinitely (loop).

Formally, the Halting Problem can be stated as follows: Given a description of a Turing machine M and an input w , can we determine whether M , when given input w , will eventually halt or continue running indefinitely?

Turing demonstrated that it's impossible to create a general algorithm that can solve the Halting Problem for all possible inputs and Turing machines. He proved this using a technique called diagonalization, which involves constructing a Turing machine that, when given its own description as input, behaves in a way that contradicts the assumption that such a solving algorithm exists.



Here's a high-level explanation of Turing's proof:

Suppose there exists an algorithm H that can solve the Halting Problem.

We can then construct a new Turing machine, let's call it D , that, given the description of any Turing machine M , simulates M on its own description.

If M halts on its own description, D enters an infinite loop; otherwise, D halts.

Now, consider what happens when we feed D its own description as input:

If D halts, then it must be because M (D itself) loops on its own description, contradicting the assumption that D halts.

If D loops, then it must be because M (D itself) halts on its own description, contradicting the assumption that D loops.

Thus, we've reached a contradiction, proving that no algorithm H can exist to solve the Halting Problem for all possible inputs and Turing machines.

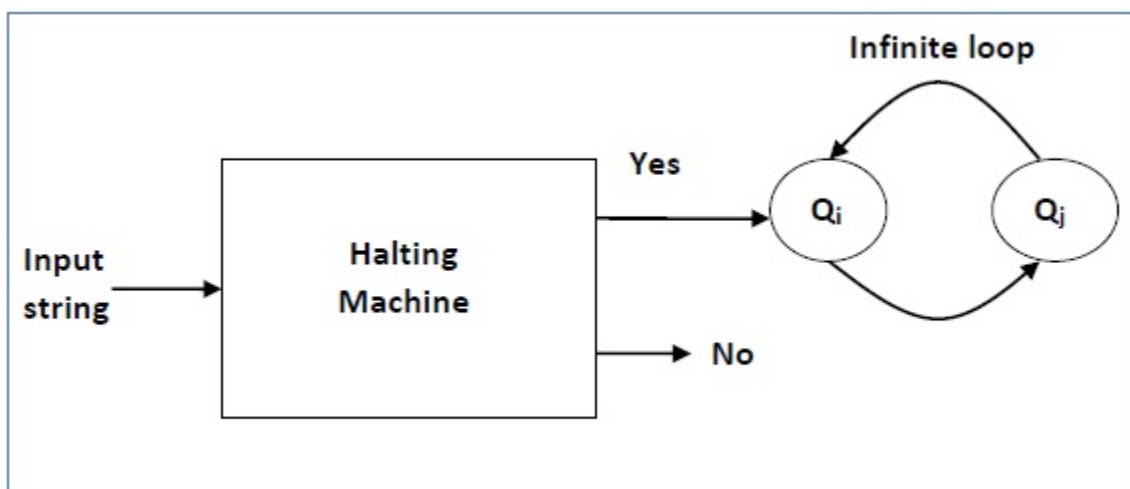
In essence, Turing's proof demonstrates the inherent limitation of algorithmic computation. While specific instances of the Halting Problem may be solvable for certain inputs and algorithms, there's no general algorithm that can solve it for all possible cases. This has significant implications for the theory of computation and the development of computer science.

5.4.2 Inverted Halting Problem

The concept of an "inverted halting machine" isn't a standard term in computer science or theoretical computation. However, if we interpret it as a hypothetical variation of the Halting Problem or a machine designed to do the opposite of what a typical halting machine does, we can explore it conceptually.

In the context of the Halting Problem, a halting machine (HM) is a theoretical construct or algorithm that attempts to determine whether a given Turing machine will halt (terminate) or run indefinitely (loop) when provided with a specific input.

Now, if we consider an "inverted halting machine," we might imagine it as a theoretical construct or algorithm that tries to determine whether a given Turing machine will loop indefinitely on a particular input, as opposed to determining whether it will halt. In other words, instead of asking "Will this Turing machine halt?" the inverted halting machine would ask, "Will this Turing machine loop forever?"



Conceptually, creating such an inverted halting machine faces the same fundamental challenge as the original Halting Problem: it's impossible to construct a general algorithm that can correctly determine whether any given Turing machine will loop indefinitely for all possible inputs. This impossibility was proven by Alan Turing in his original work on the Halting Problem.

The proof of the Halting Problem's undecidability applies equally to the inverted version. Just as no algorithm can solve the Halting Problem for all Turing machines and inputs, no algorithm can accurately determine whether a Turing machine will loop indefinitely for all cases.

In summary, while the term "inverted halting machine" isn't standard, if we interpret it as a hypothetical construct or algorithm attempting to solve the opposite of the Halting Problem, it faces the same inherent limitations and undecidability as the original Halting Problem.

5.4.3 Recursive and Recursively Enumerable Languages

When a Turing machine T operates on an input string S , there are three outcomes, these are;

- It halts and accepts the string.
- It halts and rejects the string.
- Never halts, proceeds infinitely.

In theoretical computer science, recursive languages and recursively enumerable languages are two classes of languages defined in the context of formal language theory and computability theory.

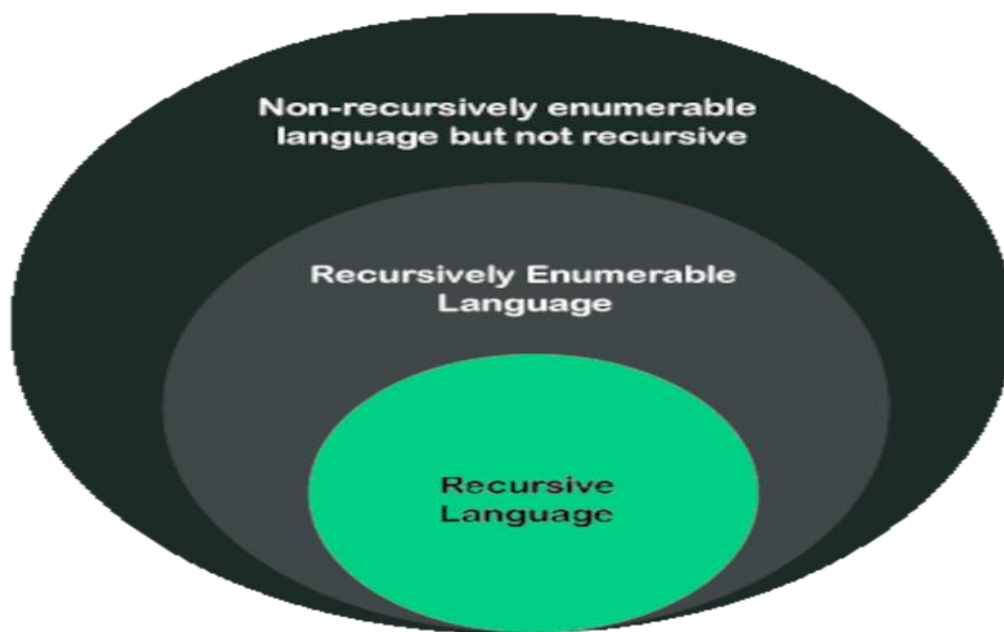


Figure: Relationship between Recursive and Recursively Enumerable Language

1. Recursive Languages (Decidable Languages):

A language L is said to be recursive or decidable if there exists a Turing machine that, when given any input string, halts and correctly decides whether the input string belongs to L or not.

Formally, a language L is recursive if there exists a Turing machine M such that for any input string w :

If w is in L , then M halts and accepts w .

If w is not in L , then M halts and rejects w .

In simpler terms, a recursive language is one for which we can write a computer program (a Turing machine, in this case) that always terminates and correctly determines whether any given string belongs to the language or not.

an example of a recursive language:

Consider the language $L = \{0^n1^n \mid n \geq 0\}$, which consists of all strings of the form "0" repeated n times followed by "1" repeated n times, where n is a non-negative integer.

To show that L is recursive, we can describe an algorithm (or a Turing machine) that decides whether any given string belongs to L or not.

Algorithm to decide membership in L :

Start scanning the input string from left to right.

If the current symbol is "0", continue scanning.

If the current symbol is "1", start scanning from the beginning of the string and ensure that there are the same number of "0"s as there are "1"s.

If the number of "0"s matches the number of "1"s, accept the string; otherwise, reject it.

This algorithm will always halt and correctly decide whether any given string belongs to L or not. If the input string is of the form 0^n1^n , the algorithm will accept it; otherwise, it will reject it.

Therefore, the language $L = \{0^n1^n \mid n \geq 0\}$ is recursive, as there exists an algorithm (or Turing machine) that decides membership for any given string.

2. Recursively Enumerable Languages (Semi-decidable Languages):

A language L is said to be recursively enumerable or semi-decidable if there exists a Turing machine that, when given any input string that belongs to L , halts and accepts it, but may either halt and reject or loop indefinitely when given an input string that does not belong to L .

Formally, a language L is recursively enumerable if there exists a Turing machine M such that for any input string w :

If w is in L , then M halts and accepts w .

If w is not in L , then M either halts and rejects w or loops indefinitely.

In simpler terms, a recursively enumerable language is one for which we can write a computer program (a Turing machine) that always terminates and correctly accepts any string in the language, but may not necessarily halt for strings not in the language.

In summary, recursive languages are those for which there exists an algorithm that always halts and correctly decides membership, while recursively enumerable languages are those for which there exists an algorithm that always halts and correctly accepts strings in the language, but may not halt for strings not in the language. Both classes are important in the study of computability and formal language theory, providing insights into the limits of computation and the nature of different types of languages.

For example, let's illustrate recursively enumerable languages (also known as semi-decidable languages) with an example:

Consider the language $L = \{w\#w \mid w \text{ is any string}\}$.

Here, "#" is a special symbol that separates two identical strings. So, any string followed by "#" and then the same string again belongs to the language L.

For example:

"abc#abc" is in L

"xyz#xyz" is in L

"123#123" is in L

But, "abc#xyz" is not in L because the two sides of "#" are different.

To show that L is recursively enumerable, we can describe a Turing machine that accepts strings in L but may not halt for strings not in L.

Turing machine algorithm for L:

Start scanning the input string.

- If the left part of "#" matches the right part, accept the string.
- If the left part of "#" doesn't match the right part, continue scanning.
- If the input string doesn't contain "#", reject the string.

This Turing machine will halt and accept any string in L, but it may not halt for strings not in L. For example, if the input string is "abc#xyz", the machine will keep scanning indefinitely because the left and right parts don't match.

Therefore, the language $L = \{w\#w \mid w \text{ is any string}\}$ is recursively enumerable because there exists a Turing machine that accepts strings in L but may not halt for strings not in L.

Properties of both recursive and recursively enumerable languages.

Recursive Languages (Decidable Languages):

- **Decision Procedure:** Recursive languages have a decision procedure, meaning there exists an algorithm (or Turing machine) that can always determine whether a given string belongs to the language or not. This algorithm always halts and gives a definite answer.
- **Halts on All Inputs:** The decision procedure for recursive languages always halts for any input string, regardless of whether the string is in the language or not.
- **Closed under Complement:** Recursive languages are closed under complementation. This means that if a language L is recursive, then its complement (the set of all strings not in L) is also recursive.
- **Effective Enumeration:** Recursive languages can be effectively enumerated, meaning there exists an algorithm that can list all strings in the language in some order.

Recursively Enumerable Languages (Semi-decidable Languages):

- **Acceptance Procedure:** Recursively enumerable languages have an acceptance procedure, meaning there exists a Turing machine that can accept any string in the language, but it may not halt for strings not in the language.
- **Halts on Acceptance:** The acceptance procedure for recursively enumerable languages always halts and accepts any string that belongs to the language. However, it may not halt for strings not in the language.
- **Not Closed under Complement:** Recursively enumerable languages are not closed under complementation. This means that if a language L is recursively enumerable, its complement may or may not be recursively enumerable.
- **Semi-effective Enumeration:** Recursively enumerable languages can be semi-effectively enumerated, meaning there exists an algorithm that can list all strings in the language, but it may not halt if the language is infinite.

In summary, recursive languages have a clear decision procedure and always halt on all inputs, while recursively enumerable languages have an acceptance procedure that may not halt for strings not in the language. Recursive languages are closed under complementation and have effective enumeration, whereas recursively enumerable languages are not closed under complementation and have semi-effective enumeration.

6 Other Phases of Compiler

6.1 Syntax Directed Translation

<https://www.youtube.com/watch?v=a9pzpMH4C50>

Syntax-directed translation is a method used in compiler design where translation rules are embedded within the production rules of the grammar. These rules associate semantic actions with the syntactic structure of the input, allowing for the translation of source code into target code while parsing it.

In this approach, the translation process is closely tied to the parsing process. As the parser recognizes the syntactic structure of the input, it triggers semantic actions associated with the grammar rules. These actions can include generating intermediate representations, constructing symbol tables, and emitting target code.

Grammar + semantic rule = SDT (syntax-directed translation)

The key components of syntax-directed translation include:

- **Syntax Rules:** These rules define the syntactic structure of the input language using context-free grammar.
- **Attributes:** Attributes are used to attach additional information to grammar symbols. They can represent data types, memory locations, or any other properties relevant to translation.
- **Semantic Actions:** Semantic actions are associated with grammar rules and are triggered during parsing. These actions perform computations, generate code, or manipulate attributes.
- **Attribute Evaluation:** During parsing, attributes are evaluated based on the production rules and semantic actions, propagating information through the parse tree.
- **Intermediate Representations:** Syntax-directed translation often involves generating intermediate representations (IR) of the source code. These representations are used as an intermediate step before generating the final target code.

By integrating translation rules directly into the grammar, syntax-directed translation simplifies the design of compilers and facilitates the generation of efficient and correct code. However, complex translation tasks may require additional techniques such as semantic analysis and optimization.

Example:

Production	Semantic Rules
$E \rightarrow E + T$	$E.val := E.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T * F$	$T.val := T.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (F)$	$F.val := F.val$
$F \rightarrow \text{num}$	$F.val := \text{num.lexval}$

$E.val$ is one of the attributes of E .

num.lexval is the attribute returned by the lexical analyzer.

6.1.2 Syntax Directed Definition

<https://www.youtube.com/watch?v=nIMBA0qO684>

A Syntax Directed Definition (SDD) is a formal method used in compiler construction and programming language design to specify the behavior of language constructs. It associates semantic rules with the production rules of a grammar.

In simpler terms, an SDD defines how to process or translate the various syntactic constructs of a programming language while considering their semantics. Each production rule in the grammar is augmented with semantic actions that define what needs to be done when that rule is applied during parsing.

Syntax Directed Definition (S.D.D) = Context Free Grammar + Semantic Rules

Attributes are associated with grammar symbols and sequence rules are associated with productions. If 'x' is a symbol and 'a' is one of the attribute then x.a denotes the value of node x. These attributes may be numbers, strings, references, data types, etc

Production	Semantic Rules
$E \rightarrow E + T$	$E.val := E.val + T.val$
$E \rightarrow T$	$E.val := T.val$

Types of Attributes:

Attributes can be classified into several types based on their characteristics and how they are used in the definition. Here are some common types of attributes:

- **Synthesized attributes:** Synthesized attributes are **computed or derived from attributes of a node's children in a parse tree**. They are typically used to pass information up the parse tree during bottom-up parsing. Synthesized attributes are calculated during tree traversal and are associated with non-terminal symbols in the grammar. They are often used for code generation or to compute values for expressions.

Example:

$A \rightarrow BCD$ $A \rightarrow$ PARENT NODE B,C,D are children nodes
 $A.S \rightarrow B.S$
 $A.S \rightarrow C.S$
 $A.S \rightarrow D.S$

In this A is TAKING FROM all children nodes.

- **Inherited attributes:** Inherited attributes are passed down the parse tree from a node's parent to its children. They are used to provide context or additional information to child nodes during parsing. Inherited attributes are associated with non-terminal symbols in the grammar and are often used for type checking or scope resolution.

Example:

A → BCD A → PARENT NODE B, C, D are children nodes

C.i → A.i – parent node

C.i → B.S – sibling node

C.i → D.S – sibling node

In this A is TAKING FROM all children nodes.

Aspect	Syntax-Directed Translation (SDT)	Syntax-Directed Definition (SDD)
Purpose	Methodology for specifying translations	Formalism for defining translation schemes
Associated Actions	Translation rules embedded in the grammar	Rules specifying translation for each production
Attribute Types	Attributes can be synthesized or inherited	Attributes can be synthesized or inherited
Computation Direction	Attributes can be computed bottom-up or top-down	Specifies how attributes are computed
Example	SDT associates translation actions with grammar productions, e.g., generating intermediate code during parsing	SDD defines translation rules for each production in a context-free grammar, e.g., specifying how expressions are evaluated

Example: Consider a simple grammar for arithmetic expressions:

Expr → Expr '+' Term | Term

Term → Term '*' Factor | Factor

Factor → '(' Expr ')' | Number

Number → [0-9]

Syntax-Directed Translation (SDT) Example:

Suppose we want to generate intermediate code for the above grammar:

Production	Semantic Rules
Expr -> Expr '+' Term	Expr.code = Expr1.code + Term.code, Expr.value = Expr1.value + Term.value
Expr -> Term	Expr.code = Term.code, Expr.value = Term.value
Term -> Term '*' Factor	Term.code = Term1.code + Factor.code, Term.value = Term1.value * Factor.value
Term -> Factor	Term.code = Factor.code, Term.value = Factor.value
Factor -> '(' Expr ')'	Factor.code = Expr.code, Factor.value = Expr.value
Factor -> Number	Factor.code = "", Factor.value = Number.value

Syntax-Directed Definition (SDD) Example:

Using SDD, we define translation rules for each production:

Production	Semantic Rules
Expr -> Expr '+' Term	Expr.code = Expr1.code + '+' + Term.code, Expr.value = Expr1.value + Term.value
Expr -> Term	Expr.code = Term.code, Expr.value = Term.value
Term -> Term '*' Factor	Term.code = Term1.code + '*' + Factor.code, Term.value = Term1.value * Factor.value
Term -> Factor	Term.code = Factor.code, Term.value = Factor.value
Factor -> '(' Expr ')'	Factor.code = '(' + Expr.code + ')', Factor.value = Expr.value
Factor -> Number	Factor.code = Number.value, Factor.value = Number.value

In both cases, the translation rules specify how to generate intermediate code or perform actions during parsing. **SDT is the methodology, while SDD provides a formalism for specifying translation schemes within SDT.**

6.1.3 Evaluation Order Of Syntax Direct Definition

<https://www.youtube.com/watch?v=bqk6VCPQVP4>

- A dependency graph characterizes the possible order in which we can calculate the attributes at various nodes of the parse tree.
- If there is an edge from node M to N, then the attribute corresponding to M first be evaluated before evaluating N.
- Thus the allowable orders of evaluation are N_1, N_2, \dots, N_k if there is an edge from N_i to N_j then $i < j$
- Such an ordering embeds a directed graph into a linear order, and is called a topological sort of the graph.

- If there is any cycle in the graph ,then there is no topological sort. ie, there is no way to evaluate SDD on this parse tree

6.2 Intermediate Code generation

https://www.youtube.com/watch?v=by7yPDu_JDA

Intermediate code generation is a crucial phase in compiler design where the source code is translated into an intermediate representation that is closer to machine language but still independent of the target machine. This intermediate representation serves as a bridge between the high-level source code and the low-level machine code. Here's an overview of intermediate code generation with an example:

Purpose: Intermediate code generation simplifies the process of translating source code into machine code by breaking it down into smaller, more manageable steps. It also makes the compiler more portable as the same intermediate code can be used to generate code for different target architectures.

Types of Intermediate Code:

- **Three-address code:** Represents instructions with at most three operands.
- **Quadruples:** Consist of four fields: operator, operand1, operand2, and result.
- **Abstract Syntax Tree (AST):** Represents the hierarchical structure of the source code.
- **Postfix Notation:** Converts infix expressions to postfix for easier evaluation.

Steps:

- **Lexical Analysis:** Tokenizing the source code.
- **Syntax Analysis:** Parsing the tokens to create a parse tree or AST.
- **Semantic Analysis:** Checking the semantics and creating a symbol table.
- **Intermediate Code Generation:** Translating the parse tree/AST into intermediate code.
- **Intermediate Code Optimization:** After generating intermediate code, optimization techniques can be applied to improve efficiency.

Example:

- Let's consider a simple example of generating three-address code for a mathematical expression:
Source Code: $a = (b + c) * (d - e)$

Intermediate Code (Three-address code):

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

In this example, each line of the intermediate code represents an operation with at most three operands. t1, t2, and t3 are temporary variables used to store intermediate results.

Pseudocode Algorithm:

1. Traverse the parse tree or AST in a depth-first manner.
2. For each node:
 - If it's an arithmetic operation:
 - Generate a new temporary variable for the result.
 - Generate a three-address code instruction for the operation.
 - If it's an assignment:
 - Generate a three-address code instruction to assign the value to the target variable.
 - If it's a conditional or loop:
 - Generate code for condition evaluation and branching.
3. Continue until all nodes are visited.

Intermediate code generation simplifies the process of translating high-level source code into machine code by providing a common representation that facilitates optimization and target code generation. It plays a crucial role in the efficiency and portability of compilers.

6.2.1 Variants of Syntax Trees

A syntax tree is a tree in which each leaf node represents an operand, while each inside node represents an operator. The Parse Tree is abbreviated as the syntax tree. The syntax tree is usually used when representing a program in a tree structure

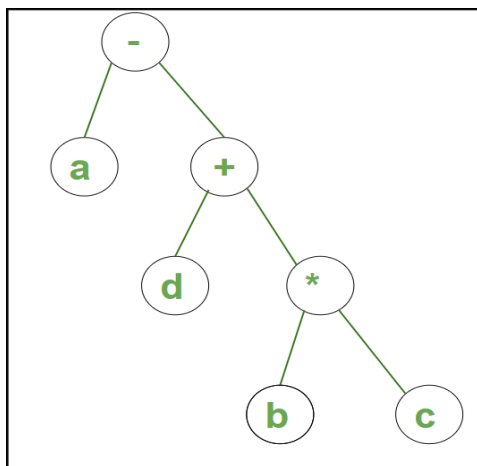


Figure: Syntax tree

A syntax tree basically has two variants which are described below:

- Directed Acyclic Graphs for Expressions (DAG)
- The Value-Number Method for Constructing DAGs

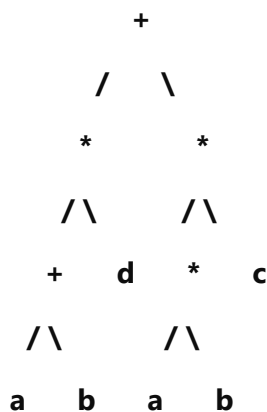
<https://www.youtube.com/watch?v=TXlZDD4DpdY>

1. Directed Acyclic Graphs for Expression (DAG)

A Directed Acyclic Graph (DAG) for expressions is another representation used by compilers to optimize code. It's constructed from the syntax tree of an expression by identifying common subexpressions and representing them only once in the graph. Here's how it works with an example:

Consider the expression: $((a + b) * c) + ((a + b) * d)$

Syntax Tree:



Algorithm for construction of DAG

For constructing a DAG, the input and output are as follows.

Input- The input will contain a basic block.

Output- The output will contain the following information-

Each node of the graph represents a label.

- If the node is a leaf node, the label represents an identifier.
- If the node is a non-leaf node, the label represents an operator.

Each node contains a list of attached identifiers to hold the computed value.

There are three possible scenarios on which we can construct a DAG.

Case 1: $x = y \text{ op } z$

where x , y , and z are operands and op is an operator.

Case 2: $x = \text{op } y$

where x and y are operands and op is an operator.

Case 3: $x = y$

where x and y are operands.

Now, we will discuss the steps to draw a DAG handling the above three cases.

Steps

To draw a DAG, follow these three steps.

Step 1:

According to step 1,

If, in any of the three cases, the y operand is not defined, then create a $\text{node}(y)$.

If, in case 1, the z operand is not defined, then create a $\text{node}(z)$.

Step 2:

According to step 2,

For case 1, create a $\text{node}(\text{op})$ with $\text{node}(y)$ as its left child and $\text{node}(z)$ as its right child. Let the name of this node be n .

For case 2, check whether there is a $\text{node}(\text{op})$ with one child node as $\text{node}(y)$. If there is no such node, then create a node.

For case 3, node n will be $\text{node}(y)$.

Step 3:

For a $\text{node}(x)$, delete x from the list of identifiers. Add x to the list of attached identifiers list found in step 2. At last, set $\text{node}(x)$ to n .

Now, we will consider an example of drawing a DAG.

Example 1

Consider the following statements with their three-address code-

$a = b * c$

$d = b$

$e = d * c$

$b = e$

$f = b + c$

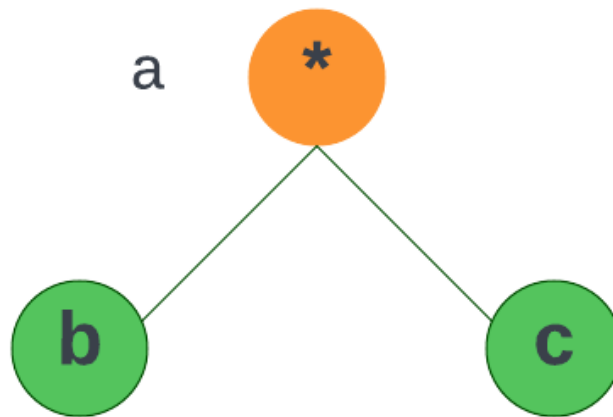
$g = d + f$

We will construct a DAG for these six statements

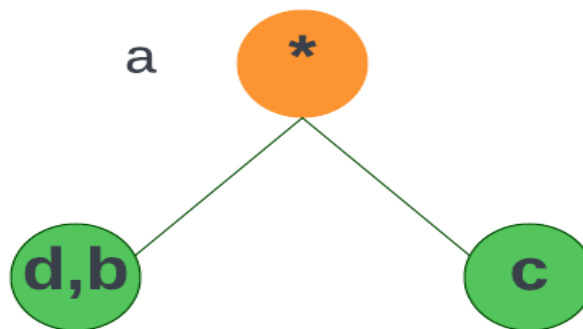
Solution:

Since we have six different statements, we will start drawing a graph from the first one.

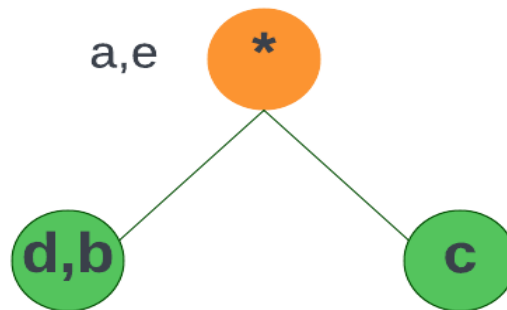
Step 1: The first statement is $a = b * c$. This statement lies in the first case from the three cases defined above. So, we will draw a node(*) with its left child node(b) and right child node(c).



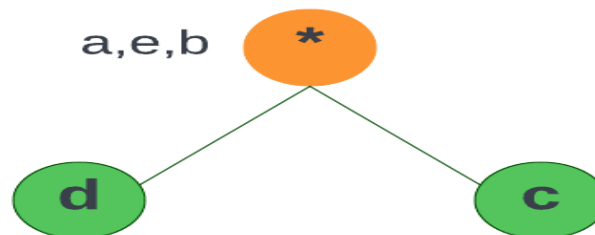
Step 2: The second statement is $d = b$. This statement lies in the third case from the three cases defined above. Since we already have a node defining operand b, i.e., node(b), we will append d to node(b).



Step 3: The third statement is $e = d * c$. This statement lies in the first case from the three cases defined above. Since we already have a node defining the $*$ (multiplication) operation, i.e., node($*$), and nodes defining operand d and c , i.e., node(d) and node(c), respectively. Thus, we will append the result of $d * c$, i.e., e to a .



Step 4: The fourth statement is $b = e$. This statement lies in the third case from the three cases defined above. Since we have both b and e defined in our graph, we will simply append b to e .



Step 5: The fifth statement is $f = b + c$. This statement lies in the first case from the three cases defined above. Since we already have a node defining operands b and c , i.e., node(b) and node(c), respectively but no node representing $+$ (addition) operation. We will draw a node($+$) with its left child node(b) and right child node(c).

Step 6: The sixth statement is $g = d + f$. This statement lies in the first case from the three cases defined above. Since we already have a node defining operands d and f . We will draw a node($+$) with its left child node(d) and right child node(f).

The above graph is the final DAG representation for the given basic block.

2. The Value-Number Method for Constructing DAGs

Nodes of a syntax tree or DAG are stored in an array of records. The integer index of the record for a node in the array is known as the value number of that node.



1	id			to entry for 1
2	num	10		
3	+	1	2	
4	=	1	3	
5	...			

(b) Array.

The signature of a node is a triple $\langle \text{op}, l, r \rangle$ where op is the label, l the value number of its left child, and r the value number of its right child. The value-number method for constructing the nodes of a DAG uses the signature of a node to check if a node with the same signature already exists in the array. If yes, returns the value number. Otherwise, creates a new node with the given signature.

Since searching an unordered array is slow, there are many better data structures to use. Hash tables are a good choice.

6.2.2 Three Address Code

<https://www.youtube.com/watch?v=0MdPdOmxMUI>

Three-address code (TAC) is a low-level intermediate representation used by compilers to translate high-level code into machine code. It typically consists of instructions that perform simple operations and have at most three operands. TAC is easier to analyze and optimize than the original high-level code. Here's an example of TAC with quadruples and triples:

Quadruples Example:

Quadruples consist of four fields: operator, operand1, operand2, and result.

Consider the following high-level code:

int a, b, c;

a = 5;

b = 10;

c = a + b;

The equivalent TAC in quadruples could be:

	operator	src1	src2	result
1.	=	5	-	a
2.	=	10	-	b

```

3.  +      a      b      t1
4.  =      t1     -      c

```

In this example:

Quadruple 1 assigns the constant value 5 to variable a.

Quadruple 2 assigns the constant value 10 to variable b.

Quadruple 3 performs addition of variables a and b, storing the result in temporary variable t1.

Quadruple 4 assigns the value of t1 to variable c.

Triples are similar to quadruples but have only three fields: operator, operand1, and operand2. The result is usually implicit.

The same code represented using triples would be:

```

1.  =   5    a
2.  =  10    b
3.  +   a    b

```

Here, there is no explicit assignment of the result of the addition. Instead, it's implied that the result of the addition operation is stored somewhere (e.g., in a temporary variable or directly in memory).

Both quadruples and triples serve as a way to represent code in a form that's easier for the compiler to manipulate, analyze, and optimize before generating the final machine code.

6.3 Code Generation

6.3.1 Issues in the Design of a Code Generator

<https://www.youtube.com/watch?v=3SjQ9WRmI4M>

Main issues in the design of a code generator are:

- Input to the code generator
- Target program
- Memory management
- Instruction selection
- Register allocation
- Evaluation order

1. Input to the code generator

In the input to the code generator, design issues in the code generator intermediate code created by the frontend and information from the symbol table that defines the run-time addresses of the data objects signified by the names in the intermediate representation are fed into the code generator. Intermediate codes may be represented mainly in quadruples, triples, indirect triples, postfix notation, syntax trees, DAGs(Directed Acyclic Graph), etc. The code generation step assumes that the input is free of all syntactic and state semantic mistakes, that all essential type checking has been performed, and that type-conversion operators have been introduced where needed.

2. Target program

The code generator's output is the target program. The result could be:

- **Assembly language:** It allows subprograms to be separately compiled.
- **Relocatable machine language:** It simplifies the code generating process.
- **Absolute machine language:** It can be stored in a specific position in memory and run immediately.

3. Memory management

In the memory management design, the source program's frontend and code generator map names address data items in run-time memory. It utilizes a symbol table. In a three-address statement, a name refers to the name's symbol-table entry. Labels in three-address statements must be transformed into instruction addresses.

For example,

j: goto i generates the following jump instruction:

if $i < j$, A backward jump instruction is generated with a target address equal to the quadruple i code location.

If $i > j$, It's a forward jump. The position of the first quadruple j machine instruction must be saved on a list for quadruple i. When i is processed, the machine locations for all instructions that forward hop to i are populated.

4. Instruction selection

In the Instruction selection, the design issues in the code generator program's efficiency will be improved by selecting the optimum instructions. It contains all of the instructions, which should be thorough and consistent. Regarding efficiency, instruction speeds, and machine idioms have a big effect. Instruction selection is simple if we don't care about the target program's efficiency.

The relevant three-address statements, for example, would be translated into the following code sequence:

P:=Q+R

S:=P+T

MOV Q, R0

ADD R, R0

MOV R0, P

MOV P, R0

ADD T, R0

MOV R0, S

The fourth sentence is unnecessary since the P value is loaded again in that statement already stored. It results in an inefficient code sequence. A given intermediate representation can be translated into several distinct code sequences, each with considerable cost differences. Previous knowledge of instruction cost is required to build good sequences, yet reliable cost information is difficult to forecast.

5. Register allocation

In the Register allocation, design issues in the code generator can be accessed faster than memory. The instructions involving operands in the register are shorter and faster than those involved in memory operands.

The following sub-problems arise when we use registers:

- Register allocation: In register allocation, we select the set of variables that will reside in the register.
- Register assignment: In the Register assignment, we pick the register that contains a variable.

Certain machines require even-odd pairs of registers for some operands and results.

Example

Consider the following division instruction of the form:

D x, y

Where,

x is the dividend even register in even/odd register pair

y is the divisor

An odd register is used to hold the quotient.

6. Evaluation order

The code generator determines the order in which the instructions are executed. The target code's efficiency is influenced by order of computations. Many computational orders will only require a few registers to store interim results. However, choosing the best order is a completely challenging task in the general case.