Course Outcomes

CO 1: Acquire fundamental understanding of the core concepts in automata theory and Theory of Computation

CO 2: Design and develop lexical analyzers, parsers and code generators

CO 3: Design Grammars and Automata (recognizers) for different language classes and become knowledgeable about restricted models of Computation (Regular, Context Free) and their relative powers.

CO 4: Acquire fundamental understanding of the structure of a Compiler and Apply concepts automata theory and Theory of Computation to design Compilers

CO 5: Design computations models for problems in Automata theory and adaptation of such model in the field of compilers

Institution Vision

To produce top-quality engineers who are groomed for attaining excellence in their profession and competitive enough to help in the growth of nation and global society.

Institution Mission

M1: To offer affordable high-quality graduate program in engineering with value education and make the students socially responsible.

M2: To support and enhance the institutional environment to attain research excellence in both faculty and students and to inspire them to push the boundaries of knowledge base.

M3: To identify the common areas of interest amongst the individuals for the effective industry-institute partnership in a sustainable way by systematically working together.

M4: To promote the entrepreneurial attitude and inculcate innovative ideas among the engineering professionals.

Department Vision

To be a center of excellence in Information Science & Engineering education, research and training to meet the growing needs of the industry and society.

Department Mission

M1: To impart theoretical and practical knowledge through the concepts and technologies in Information Science and Engineering

M2: To foster research, collaboration and higher education with premier institutions and industries.

M3: Promote innovation and entrepreneurship to fulfill the needs of the society and industry

Program Educational Objectives

PEO1: Analyse, design and implement solutions to the real-world problems in the field of Information Science and Engineering with multidisciplinary setup

PEO2: Keep abreast with the technology, innovation and pursue higher education with high standards of social and professional ethics

PEO3: Develop professional and entrepreneurship skills to work effectively as an individual and in a team to meet the ever-changing goals of the organization

Program Outcomes

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.

PO2: Problem Analysis: Identify, formulate, review research literature, and analyse complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural science and engineering sciences.

PO3: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal and environmental considerations.

PO4: Conduct investigations of complex problems: Use research based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: Modern tool usage: Create, select and apply appropriate techniques, resources and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations

PO6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice

PO7: Environment sustainability: Understand the impact of the professional engineering solutions in the societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to ones own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Lifelong learning: Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broader context of technological change.

Program Specific Outcomes

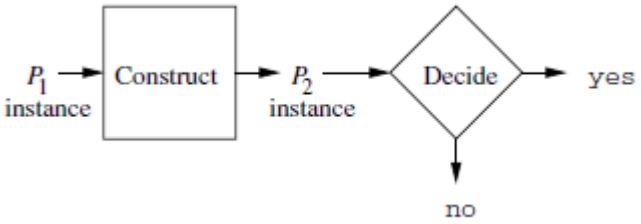
PSO1: Design, implement and maintain the information systems that fulfill the current needs of the industry and society

PSO2: Apply computational theory, storage and networking concepts to solve the day to day problems of the world

INTRODUCTION TO TURING MACHINES

Problems That Computers Cannot Solve

A problem that cannot be solved by computer is called undecidable. We could prove this new problem undecidable by a technique, assume there is a program to solve a problem and develop a paradoxical program that must do two contradictory things, once we have one problem that we know is undecidable, we no longer have to prove the existence of a paradoxical situation. It is sufficient to show that if we could solve the new problem, then we could use that solution to solve a problem we already know is undecidable. The strategy is suggested in Fig. the technique is called the reduction of P_1 to P_2.



Suppose that we know problem P_1 is undecidable, and P_2 is a new problem that we would like to prove is undecidable as well. We suppose that there is a program represented in Fig. by the diamond labeled “decide” this program prints yes or no, depending on whether its input instance of problem P_2 is or is not in the language of that problem.

In order to make a proof that problem P1 is undecidable, we have to invent a construction, represented by the square box in Fig. that converts instances of P_1 to instances of P_2 that have the same answer. That is, any string in the language P_1 is converted to some string in the language P_2, and any string over the alphabet of P_1 that is not in the language P_1 is converted to a string that is not in the language P_2. Once we have this construction, we can solve P_1 as follows:

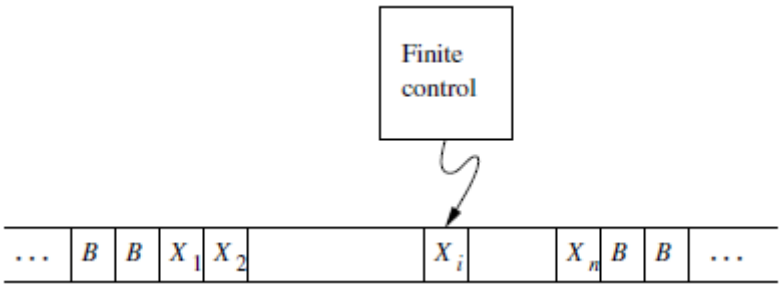
1. Given an instance of P_1, that is, given a string w that may or may not be in the language P_1, apply the construction algorithm to produce a string x.
2. Test whether x is in P_2, and give the same answer about w and P_1.

   If w is in P_1, then x is in P_2, so this algorithm says yes. If w is not in P_1, then x is not in P_2, and the algorithm says no. Either way, it says the truth about w. Since we assumed that no algorithm to decide membership of a string in P_1 exists, we have a proof by contradiction that the hypothesized decision algorithm for P_2 does not exist i.e. P_2 is undecidable.

The Turing Machine

Notation for the Turing Machine

The machine consists of a finite control, which can be in any of a finite set of states. There is a tape divided into squares or cells each cell can hold any one of a finite number of symbols.



Initially, the input, which is a finite-length string of symbols chosen from the input alphabet, is placed on the tape. All other tape cells, extending in infinitely to the left and right, initially hold a special symbol called the blank. The blank is a tape symbol, but not an input symbol, and there may be other tape symbols besides the input symbols and the blank, as well.

There is a tape head that is always positioned at one of the tape cells. The Turing machine is said to be scanning that cell. Initially, the tape head is at the leftmost cell that holds the input.

A move of the Turing machine is a function of the state of the finite control and the tape symbol scanned. In one move, the Turing machine will:

1. **Change state**. The next state optionally may be the same as the current state.
2. **Write a tape symbol in the cell scanned**. This tape symbol replaces whatever symbol was in that cell. Optionally, the symbol written may be the same as the symbol currently there.
3. **Move the tape head left or right**. In our formalism we require a move, and do not allow the head to remain stationary. This restriction does not constrain what a Turing machine can compute, since any sequence of moves with a stationary head could be condensed, along with the next tape-head move, into a single state change, a new tape symbol, and a move left or right.

The formal notation we shall use for a Turing machine (TM) is similar to that used for finite automata or PDA's. We describe a TM by the 7-tuple:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

whose components have the following meanings:

Q: The finite set of *states* of the finite control.

Σ: The finite set of *input symbols*.

Γ: The complete set of *tape symbols*; Σ is always a subset of Γ.

δ: The *transition function*. The arguments of $\delta(q, X)$ are a state q and a tape symbol X. The value of $\delta(q, X)$, if it is defined, is a triple (p, Y, D), where:

1. p is the next state, in Q.
2. Y is the symbol, in Γ, written in the cell being scanned, replacing whatever symbol was there.
3. D is a *direction*, either L or R, standing for "left" or "right," respectively, and telling us the direction in which the head moves.

q_0: The *start state*, a member of Q, in which the finite control is found initially.

B: The *blank* symbol. This symbol is in Γ but not in Σ; i.e., it is not an input symbol. The blank appears initially in all but the finite number of initial cells that hold input symbols.

F: The set of *final* or *accepting* states, a subset of Q.

Instantaneous Descriptions for Turing Machines

A notation for configurations or instantaneous descriptions (ID's), like the notation developed for PDA's. Since a TM, in principle, has an in infinitely long tape. we might imagine that it is impossible to describe the configurations of a TM succinctly. However, after any finite number of moves, the TM can have visited only a finite number of cells□ even though the number of cells visited can eventually grow beyond any finite limit. Thus, in every ID, there is an infinite prefix and an infinite suffix of cells that have never been visited. These cells must all hold either blanks or one of the finite number of input symbols. We thus show in an ID only the cells between the leftmost and the rightmost non blanks. Under special conditions, when the head is scanning one of the leading or trailing blanks, a finite number of blanks to the left or right of the nonblank portion of the tape must also be included in the ID.

In addition to representing the tape, we must represent the finite control and the tape-head position. To do so, we embed the state in the tape, and place it immediately to the left of the cell scanned. To disambiguate the tape plus state string, we have to make sure that we do not use as a state any symbol that is also a tape symbol. However, it is easy to change the names of the states so they have nothing in common with the tape symbols, since the operation of the TM does not depend on what the states are called. Thus, we shall use the string $X_1X_2\ldots X_{i-1}qX_iX_{i+1}\ldots X_n$ to represent an ID in which

1. q is the state of the Turing machine.
2. The tape head is scanning the ith symbol from the left.
3. $X_1X_2\cdots X_n$ is the portion of the tape between the leftmost and the rightmost nonblank. As an exception, if the head is to the left of the leftmost nonblank or to the right of the rightmost nonblank, then some prefix or suffix of $X_1X_2\cdots X_n$ will be blank, and i will be 1 or n, respectively.

We describe moves of a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ by the $\underset{M}{\vdash}$ notation that was used for PDA's. When the TM M is understood, we shall use just $\vdash$ to reflect moves. As usual, $\overset{*}{\underset{M}{\vdash}}$, or just $\overset{*}{\vdash}$, will be used to indicate zero, one, or more moves of the TM M.

Suppose $\delta(q, X_i) = (p, Y, L)$; i.e., the next move is leftward. Then

$$X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n \underset{M}{\vdash} X_1X_2\cdots X_{i-2}pX_{i-1}YX_{i+1}\cdots X_n$$

Notice how this move reflects the change to state p and the fact that the tape head is now positioned at cell $i-1$. There are two important exceptions:

1. If $i = 1$, then M moves to the blank to the left of X_1. In that case,
$$qX_1X_2\cdots X_n \underset{M}{\vdash} pBYX_2\cdots X_n$$
2. If $i = n$ and $Y = B$, then the symbol B written over X_n joins the infinite sequence of trailing blanks and does not appear in the next ID. Thus,
$$X_1X_2\cdots X_{n-1}qX_n \underset{M}{\vdash} X_1X_2\cdots X_{n-2}pX_n$$

Now, suppose $\delta(q, X_i) = (p, Y, R)$; i.e., the next move is rightward. Then

$$X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n \underset{M}{\vdash} X_1X_2\cdots X_{i-1}YpX_{i+1}\cdots X_n$$

Here, the move reflects the fact that the head has moved to cell $i+1$. Again there are two important exceptions:

1. If $i = n$, then the $i+1$st cell holds a blank, and that cell was not part of the previous ID. Thus, we instead have
$$X_1X_2\cdots X_{n-1}qX_n \underset{M}{\vdash} X_1X_2\cdots X_{n-1}YpB$$
2. If $i = 1$ and $Y = B$, then the symbol B written over X_1 joins the infinite sequence of leading blanks and does not appear in the next ID. Thus,
$$qX_1X_2\cdots X_n \underset{M}{\vdash} pX_2\cdots X_n$$

Example Let us design a Turing machine and see how it behaves on a typical input. The TM we construct will accept the language $\{0^n1^n \mid n \geq 1\}$. Initially, it is given a finite sequence of 0's and 1's on its tape, preceded and followed by an infinity of blanks. Alternately, the TM will change a 0 to an X and then a 1 to a Y, until all 0's and 1's have been matched.

In more detail, starting at the left end of the input, it enters a loop in which it changes a 0 to an X and moves to the right over whatever 0's and Y's it sees, until it comes to a 1. It changes the 1 to a Y, and moves left, over Y's and 0's, until it finds an X. At that point, it looks for a 0 immediately to the right, and if it finds one, changes it to X and repeats the process, changing a matching 1 to a Y.

If the nonblank input is not in $\mathbf{0^*1^*}$, then the TM will eventually fail to have a next move and will die without accepting. However, if it finishes changing all the 0's to X's on the same round it changes the last 1 to a Y, then it has found its input to be of the form 0^n1^n and accepts. The formal specification of the TM M is

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\})$$

where δ is given by the table in Fig.

	Symbol				
State	0	1	X	Y	B
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	$(q_2, 0, L)$	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	(q_4, B, R)
q_4	–	–	–	–	–

Figure A Turing machine to accept $\{0^n1^n \mid n \geq 1\}$

As M performs its computation, the portion of the tape, where M's tape head has visited, will always be a sequence of symbols described by the regular expression $\mathbf{X^*0^*Y^*1^*}$. That is, there will be some 0's that have been changed to X's, followed by some 0's that have not yet been changed to X's. Then there are some 1's that were changed to Y's, and 1's that have not yet been changed to Y's. There may or may not be some 0's and 1's following.

State q_0 is the initial state, and M also enters state q_0 every time it returns to the leftmost remaining 0. If M is in state q_0 and scanning a 0, the rule in the upper-left corner of Fig. 8.9 tells it to go to state q_1, change the 0 to an X, and move right. Once in state q_1, M keeps moving right over all 0's and Y's that it finds on the tape, remaining in state q_1. If M sees an X or a B, it dies. However, if M sees a 1 when in state q_1, it changes that 1 to a Y, enters state q_2, and starts moving left.

In state q_2, M moves left over 0's and Y's, remaining in state q_2. When M reaches the rightmost X, which marks the right end of the block of 0's that have already been changed to X, M returns to state q_0 and moves right. There are two cases:

1. If M now sees a 0, then it repeats the matching cycle we have just described.

2. If M sees a Y, then it has changed all the 0's to X's. If all the 1's have been changed to Y's, then the input was of the form 0^n1^n, and M should accept. Thus, M enters state q_3, and starts moving right, over Y's. If the first symbol other than a Y that M sees is a blank, then indeed there were an equal number of 0's and 1's, so M enters state q_4 and accepts. On the other hand, if M encounters another 1, then there are too many 1's, so M dies without accepting. If it encounters a 0, then the input was of the wrong form, and M also dies.

Here is an example of an accepting computation by M. Its input is 0011. Initially, M is in state q_0, scanning the first 0, i.e., M's initial ID is $q_0 0011$. The entire sequence of moves of M is:

$$q_0 0011 \vdash Xq_1 011 \vdash X0q_1 11 \vdash Xq_2 0Y1 \vdash q_2 X0Y1 \vdash$$
$$Xq_0 0Y1 \vdash XXq_1 Y1 \vdash XXYq_1 1 \vdash XXq_2 YY \vdash Xq_2 XYY \vdash$$
$$XXq_0 YY \vdash XXYq_3 Y \vdash XXYYq_3 B \vdash XXYYBq_4 B$$

For another example, consider what M does on the input 0010, which is not in the language accepted.

$$q_0 0010 \vdash Xq_1 010 \vdash X0q_1 10 \vdash Xq_2 0Y0 \vdash q_2 X0Y0 \vdash$$
$$Xq_0 0Y0 \vdash XXq_1 Y0 \vdash XXYq_1 0 \vdash XXY0q_1 B$$

The behavior of M on 0010 resembles the behavior on 0011, until in ID $XXYq_1 0$ M scans the final 0 for the first time. M must move right, staying in state q_1, which takes it to the ID $XXY0q_1 B$. However, in state q_1 M has no move on tape symbol B; thus M dies and does not accept its . □

Transition Diagrams for Turing Machines

We can represent the transitions of a Turing machine pictorially, much as we did for the PDA. A *transition diagram* consists of a set of nodes corresponding to the states of the TM. An arc from state q to state p is labeled by one or more items of the form X/YD, where X and Y are tape symbols, and D is a direction, either L or R. That is, whenever $\delta(q, X) = (p, Y, D)$, we find the label X/YD on the arc from q to p. However, in our diagrams, the direction D is represented pictorially by $\leftarrow$ for "left" and $\rightarrow$ for "right."

As for other kinds of transition diagrams, we represent the start state by the word "Start" and an arrow entering that state. Accepting states are indicated by double circles. Thus, the only information about the TM one cannot read directly from the diagram is the symbol used for the blank. We shall assume that symbol is B unless we state otherwise.

Example shows the transition diagram for the Turing machine of Example whose transition function was given in Fig

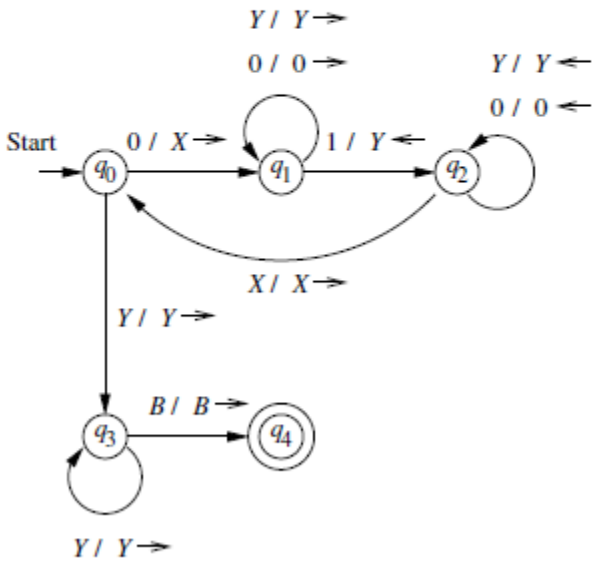


Figure Transition diagram for a TM that accepts strings of the form 0^n1^n

Example: While today we find it most convenient to think of Turing machines as recognizers of languages, or equivalently, solvers of problems. Turing's original view of his machine was as a computer of integer valued functions. In his scheme, integers were represented in unary, as blocks of a single character and the machine computed by changing the lengths of the blocks or by constructing new blocks elsewhere on the tape. In this simple example, we shall show how a Turing machine might compute the function - which is called monus or proper subtraction and is defined by m-n = max (m – n, 0). That is m-n is m - n if m > n and 0 if m < n.

A TM that performs this operation is specified by

$$M = (\{q_0, q_1, \ldots, q_6\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B)$$

Note that, since this TM is not used to accept inputs, we have omitted the seventh component, which is the set of accepting states. M will start with a tape consisting of 0^m10^n surrounded by blanks. M halts with $0^{m \dot{-} n}$ on its tape, surrounded by blanks.

M repeatedly finds its leftmost remaining 0 and replaces it by a blank. It then searches right, looking for a 1. After finding a 1, it continues right, until it comes to a 0, which it replaces by a 1. M then returns left, seeking the leftmost 0, which it identifies when it first meets a blank and then moves one cell to the right. The repetition ends if either:

1. Searching right for a 0, M encounters a blank. Then the n 0's in 0^m10^n have all been changed to 1's, and $n+1$ of the m 0's have been changed to B. M replaces the $n+1$ 1's by one 0 and n B's, leaving $m-n$ 0's on the tape. Since $m \geq n$ in this case, $m - n = m \dot{-} n$.

2. Beginning the cycle, M cannot find a 0 to change to a blank, because the first m 0's already have been changed to B. Then $n \geq m$, so $m \dot{-} n = 0$. M replaces all remaining 1's and 0's by B and ends with a completely blank tape.

Figure gives the rules of the transition function δ, and we have also represented δ as a transition diagram in Fig. The following is a summary of the role played by each of the seven states:

q_0: This state begins the cycle, and also breaks the cycle when appropriate. If M is scanning a 0, the cycle must repeat. The 0 is replaced by B, the head moves right, and state q_1 is entered. On the other hand, if M is scanning 1, then all possible matches between the two groups of 0's on the tape have been made, and M goes to state q_5 to make the tape blank.

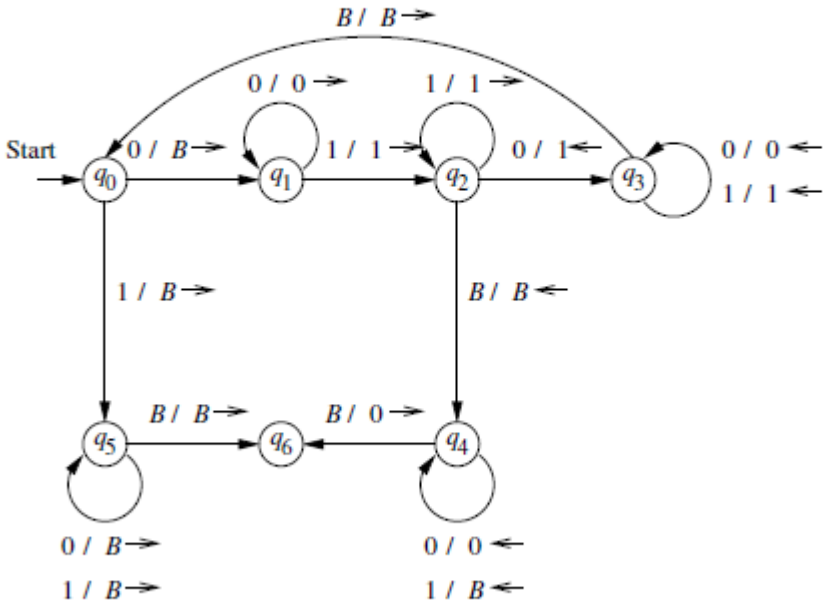
q_1: In this state, M searches right, through the initial block of 0's, looking for the leftmost 1. When found, M goes to state q_2.

q_2: M moves right, skipping over 1's, until it finds a 0. It changes that 0 to a 1, turns leftward, and enters state q_3. However, it is also possible that there are no more 0's left after the block of 1's. In that case, M in state q_2 encounters a blank. We have case (1) described above, where n 0's in the second block of 0's have been used to cancel n of the m 0's in the first block, and the subtraction is complete. M enters state q_4, whose purpose is to convert the 1's on the tape to blanks.

q_3: M moves left, skipping over 0's and 1's, until it finds a blank. When it finds B, it moves right and returns to state q_0, beginning the cycle again.

State	Symbol 0	1	B
q_0	(q_1, B, R)	(q_5, B, R)	–
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	–
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
q_6	–	–	–

A Turing machine that computes the proper-subtraction function



Transition diagram for the TM

q_4: Here, the subtraction is complete, but one unmatched 0 in the first block was incorrectly changed to a B. M therefore moves left, changing 1's to B's, until it encounters a B on the tape. It changes that B back to 0, and enters state q_6, wherein M halts.

q_5: State q_5 is entered from q_0 when it is found that all 0's in the first block have been changed to B. In this case, described in (2) above, the result of the proper subtraction is 0. M changes all remaining 0's and 1's to B and enters state q_6.

q_6: The sole purpose of this state is to allow M to halt when it has finished its task. If the subtraction had been a subroutine of some more complex function, then q_6 would initiate the next step of that larger computation.

The Language of a Turing Machine

A Turing machine accepts a language, the input string is placed on the tape, and the tape head begins at the leftmost input symbol. If the TM eventually enters an accepting state then the input is accepted, and otherwise not.

The set of languages we can accept using a Turing machine is often called the **recursively enumerable languages** or RE languages. The term "recursively enumerable" comes from computational formalisms that predate the Turing machine but that defi ne the same class of languages or arithmetic functions.

> More formally, let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a Turing machine. Then $L(M)$ is the set of strings w in Σ^* such that $q_0 w \vdash^* \alpha p \beta$ for some state p in F and any tape strings α and β.

Turing Machines and Halting

There is another notion of "acceptance" that is commonly used for Turing machines: acceptance by halting. We say a TM halts if it enters a state q, scanning a tape symbol X, and there is no move in this situation i.e. δ(q,X) is undefined.

Example: The Turing machine M of previous Example was not designed to accept a language, rather we viewed it as computing an arithmetic function. Note, however, that M halts on all strings of 0's and 1's since no matter what string M finds on its tape, it will eventually cancel its second group of 0's, if it can find such a group, against its first group of 0's and thus must reach state q_6 and halt.

We can always assume that a TM halts if it accepts. That is, without changing the language accepted, we can make δ(q,X) undefined whenever q is an accepting state. In general, without otherwise stating so:

1. We assume that a TM always halts when it is in an accepting state.

Unfortunately, it is not always possible to require that a TM halts even if it does not accept. Those languages with Turing machines that do halt eventually, regardless of whether or not they accept, are called recursive.

Turing machines that always halt, regardless of whether or not they accept, are a good model of an "algorithm". If an algorithm to solve a given problem exists, then we say the problem is decidable, so TM's that always halt figure importantly into decidability theory.

Programming Techniques for Turing Machines:

Storage in the State

We can use the finite control not only to represent a position in the "program" of the Turing machine, but to hold a finite amount of data. Figure below suggests this technique, There, we see the finite control consisting of not only a "control" state q, but three data elements A, B, and C. The technique requires no extension to the TM model we merely think of the state as a tuple. In the case of Fig we should think of the state as [q, A,B,C]. Regarding states this way allows us to describe transitions in a more systematic way, often making the strategy behind the TM program more transparent.

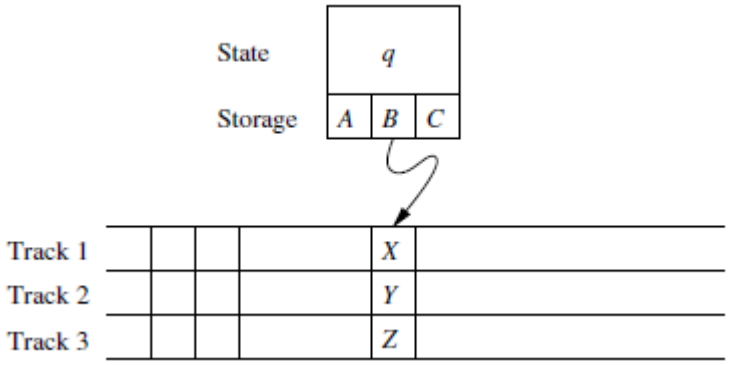


Figure : A Turing machine viewed as having finite control storage and multiple tracks

Example : We shall design a TM

$$M = (Q, \{0,1\}, \{0,1,B\}, \delta, [q_0, B], B, \{[q_1, B]\})$$

that remembers in its finite control the first symbol (0 or 1) that it sees, and checks that it does not appear elsewhere on its input. Thus, M accepts the language $\mathbf{01}^* + \mathbf{10}^*$. Accepting regular languages such as this one does not stress the ability of Turing machines, but it will serve as a simple demonstration.

The set of states Q is $\{q_0, q_1\} \times \{0,1,B\}$. That is, the states may be thought of as pairs with two components:

a) A control portion, q_0 or q_1, that remembers what the TM is doing. Control state q_0 indicates that M has not yet read its first symbol, while q_1 indicates that it *has* read the symbol, and is checking that it does not appear elsewhere, by moving right and hoping to reach a blank cell.

b) A data portion, which remembers the first symbol seen, which must be 0 or 1. The symbol B in this component means that no symbol has been read.

The transition function δ of M is as follows:

1. $\delta([q_0, B], a) = ([q_1, a], a, R)$ for $a = 0$ or $a = 1$. Initially, q_0 is the control state, and the data portion of the state is B. The symbol scanned is copied into the second component of the state, and M moves right, entering control state q_1 as it does so.

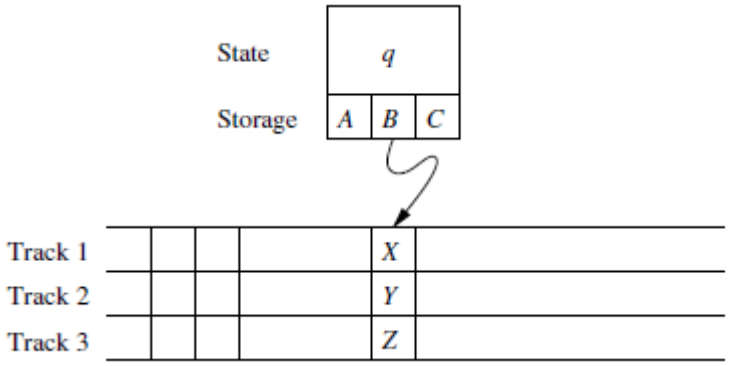
2. $\delta([q_1, a], \overline{a}) = ([q_1, a], \overline{a}, R)$ where $\overline{a}$ is the "complement" of a, that is, 0 if $a = 1$ and 1 if $a = 0$. In state q_1, M skips over each symbol 0 or 1 that is different from the one it has stored in its state, and continues moving right.

3. $\delta([q_1, a], B) = ([q_1, B], B, R)$ for $a = 0$ or $a = 1$. If M reaches the first blank, it enters the accepting state $[q_1, B]$.

Notice that M has no definition for $\delta([q_1, a], a)$ for $a = 0$ or $a = 1$. Thus, if M encounters a second occurrence of the symbol it stored initially in its finite control, it halts without having entered the accepting state.

Multiple Tracks

The tape of a Turing machine as composed of several tracks. Each track can hold one symbol and the tape alphabet of the TM consists of tuples, with one component for each "track".



Thus, for instance, the cell scanned by the tape head in Fig contains the symbol [X, Y, Z]. Like the technique of storage in the finite control, using multiple tracks does not extend what the Turing machine can do. It is simply a way to view tape symbols and to imagine that they have a useful structure.

Example : A common use of multiple tracks is to treat one track as holding the data and a second track as holding a mark. We can check off each symbol as we "use" it, or we can keep track of a small number of positions within the data by marking only those positions. Examples were two instances of this technique, but in neither example did we think explicitly of the tape as if it were composed of tracks. In the present example, we shall use a second track explicitly to recognize the non context free language.

$$L_{wcw} = \{wcw \mid w \text{ is in } (\mathbf{0}+\mathbf{1})^+\}$$

The Turing machine we shall design is:

$$M = (Q, \Sigma, \Gamma, \delta, [q_1, B], [B, B], \{[q_9, B]\})$$

where:

Q: The set of states is $\{q_1, q_2, \ldots, q_9\} \times \{0, 1, B\}$, that is, pairs consisting of a control state q_i and a data component: 0, 1, or blank. We again use the technique of storage in the finite control, as we allow the state to remember an input symbol 0 or 1.

Γ: The set of tape symbols is $\{B, *\} \times \{0, 1, c, B\}$. The first component, or track, can be either blank or "checked," represented by the symbols B and $*$, respectively. We use the $*$ to check off symbols of the first and second groups of 0's and 1's, eventually confirming that the string to the left of the center marker c is the same as the string to its right. The second component of the tape symbol is what we think of as the tape symbol itself. That is, we may think of the symbol $[B, X]$ as if it were the tape symbol X, for $X = 0, 1, c, B$.

Σ: The input symbols are $[B, 0]$, $[B, 1]$, and $[B, c]$, which, as just mentioned, we identify with 0, 1, and c, respectively.

δ: The transition function δ is defined by the following rules, in which a and b each may stand for either 0 or 1.

1. $\delta([q_1, B], [B, a]) = ([q_2, a], [*, a], R)$. In the initial state, M picks up the symbol a (which can be either 0 or 1), stores it in its finite control, goes to control state q_2, "checks off" the symbol it just scanned, and moves right. Notice that by changing the first component of the tape symbol from B to $*$, it performs the check-off.
2. $\delta([q_2, a], [B, b]) = ([q_2, a], [B, b], R)$. M moves right, looking for the symbol c. Remember that a and b can each be either 0 or 1, independently, but cannot be c.
3. $\delta([q_2, a], [B, c]) = ([q_3, a], [B, c], R)$. When M finds the c, it continues to move right, but changes to control state q_3.
4. $\delta([q_3, a], [*, b]) = ([q_3, a], [*, b], R)$. In state q_3, M continues past all checked symbols.
5. $\delta([q_3, a], [B, a]) = ([q_4, B], [*, a], L)$. If the first unchecked symbol that M finds is the same as the symbol in its finite control, it checks this symbol, because it has matched the corresponding symbol from the first block of 0's and 1's. M goes to control state q_4, dropping the symbol from its finite control, and starts moving left.
6. $\delta([q_4, B], [*, a]) = ([q_4, B], [*, a], L)$. M moves left over checked symbols.

6. $\delta([q_4, B], [*, a]) = ([q_4, B], [*, a], L)$. M moves left over checked symbols.
7. $\delta([q_4, B], [B, c]) = ([q_5, B], [B, c], L)$. When M encounters the symbol c, it switches to state q_5 and continues left. In state q_5, M must make a decision, depending on whether or not the symbol immediately to the left of the c is checked or unchecked. If checked, then we have already considered the entire first block of 0's and 1's — those to the left of the c. We must make sure that all the 0's and 1's to the right of the c are also checked, and accept if no unchecked symbols remain to the right of the c. If the symbol immediately to the left of the c is unchecked, we find the leftmost unchecked symbol, pick it up, and start the cycle that began in state q_1.
8. $\delta([q_5, B], [B, a]) = ([q_6, B], [B, a], L)$. This branch covers the case where the symbol to the left of c is unchecked. M goes to state q_6 and continues left, looking for a checked symbol.
9. $\delta([q_6, B], [B, a]) = ([q_6, B], [B, a], L)$. As long as symbols are unchecked, M remains in state q_6 and proceeds left.
10. $\delta([q_6, B], [*, a]) = ([q_1, B], [*, a], R)$. When the checked symbol is found, M enters state q_1 and moves right to pick up the first unchecked symbol.
11. $\delta([q_5, B], [*, a]) = ([q_7, B], [*, a], R)$. Now, let us pick up the branch from state q_5 where we have just moved left from the c and find a checked symbol. We start moving right again, entering state q_7.
12. $\delta([q_7, B], [B, c]) = ([q_8, B], [B, c], R)$. In state q_7 we shall surely see the c. We enter state q_8 as we do so, and proceed right.
13. $\delta([q_8, B], [*, a]) = ([q_8, B], [*, a], R)$. M moves right in state q_8, skipping over any checked 0's or 1's that it finds.
14. $\delta([q_8, B], [B, B]) = ([q_9, B], [B, B], R)$. If M reaches a blank cell in state q_8 without encountering any unchecked 0 or 1, then M accepts. If M first finds an unchecked 0 or 1, then the blocks before and after the c do not match, and M halts without accepting.

Subroutines

As with programs in general, it helps to think of Turing machines as built from a collection of interacting components, or "subroutines". A Turing machine subroutine is a set of states that perform some useful process. This set of states includes a start state and another state that temporarily has no moves, and that serves as the "return" state to pass control to whatever other set of states called the subroutine. The "call" of a subroutine occurs whenever there is a transition to its initial state. Since the TM has no mechanism for remembering a "return address", that is, a state to go to after it finishes, should our design of a TM call for one subroutine to be called from several states, we can

make copies of the subroutine, using a new set of states for each copy. The "calls" are made to the start states of different copies of the subroutine, and each copy "returns" to a different state.

Example :We shall design a TM to implement the function "multiplication", That is, our TM will start with 0^m10^n1 on its tape, and will end with 0^{mn} on the tape. An outline of the strategy is:

1. The tape will, in general, have one nonblank string of the form $0^i10^n10^{kn}$ for some k.
2. In one basic step, we change a 0 in the first group to B and add n 0's to the last group, giving us a string of the form $0^{i-1}10^n10^{(k+1)n}$.
3. As a result, we copy the group of n 0's to the end m times, once each time we change a 0 in the first group to B. When the first group of 0's is completely changed to blanks, there will be mn 0's in the last group.
4. The final step is to change the leading 10^n1 to blanks, and we are done.

The heart of this algorithm is a subroutine, which we call `Copy`. This subroutine helps implement step (2) above, copying the block of n 0's to the end. More precisely, `Copy` converts an ID of the form $0^{m-k}1q_10^n10^{(k-1)n}$ to ID $0^{m-k}1q_50^n10^{kn}$. Figure shows the transitions of subroutine `Copy`. This

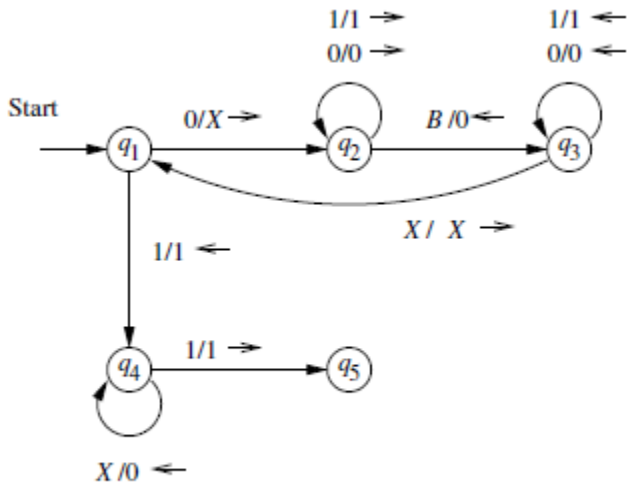


Figure The subroutine `Copy`

subroutine marks the first 0 with an X, moves right in state q_2 until it finds a blank, copies the 0 there, and moves left in state q_3 to find the marker X. It repeats this cycle until in state q_1 it finds a 1 instead of a 0. At that point, it uses state q_4 to change the X's back to 0's, and ends in state q_5.

The complete multiplication Turing machine starts in state q_0. The first thing it does is go, in several steps, from ID $q_00^m10^n$ to ID $0^{m-1}1q_10^n$. The transitions needed are shown in the portion of Fig. 8.15 to the left of the subroutine call; these transitions involve states q_0 and q_6 only.

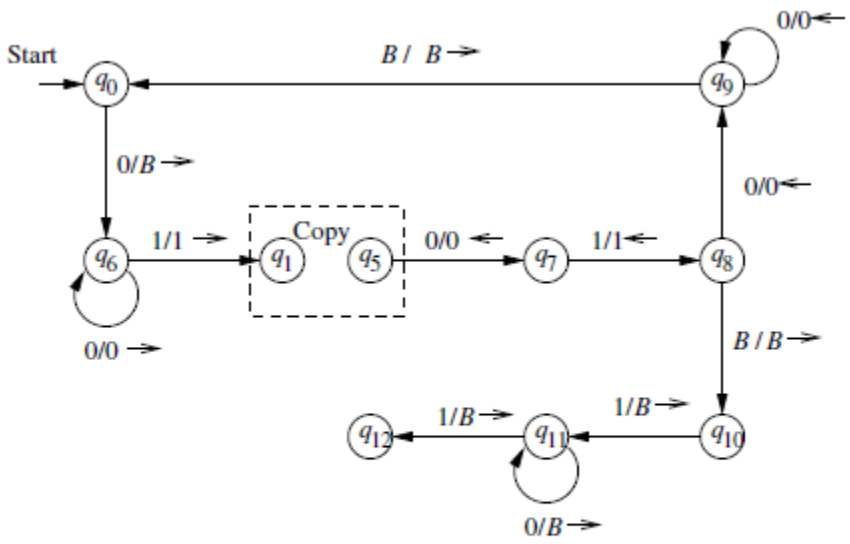


Figure : The complete multiplication program uses the subroutine Copy

Then, to the right of the subroutine call in Fig. we see states q_7 through q_{12}. The purpose of states q_7, q_8, and q_9 is to take control after Copy has just copied a block of n 0's, and is in ID $0^{m-k}1q_50^n10^{kn}$. Eventually, these states bring us to state $q_00^{m-k}10^n10^{kn}$. At that point, the cycle starts again, and Copy is called to copy the block of n 0's again.

As an exception, in state q_8 the TM may find that all m 0's have been changed to blanks (i.e., $k = m$). In that case, a transition to state q_{10} occurs. This state, with the help of state q_{11}, changes the leading 10^n1 to blanks and enters the halting state q_{12}. At this point, the TM is in ID $q_{12}0^{mn}$, and its job is done.

Extensions to the Basic Turing Machine

1. **The multitape Turing machine**
2. **The nondeterministic Turing machine:** An extension of the basic model that is allowed to make any of a finite set of choices of move in a given situation.

Multitape Turing Machines

A multitape TM is as suggested by below figure. The device has a finite control(state), and some finite number of tapes. Each tape is divided into cells, and each cell can hold any symbol of the finite tape alphabet. As in the single-tape TM, the set of tape symbols includes a blank, and has a subset called the input symbols, of which the blank is not a member. The set of states includes an initial state and some accepting states. Initially:

1. The input, a finite sequence of input symbols, is placed on the first tape.

2. All other cells of all the tapes hold the blank.
3. The finite control is in the initial state.
4. The head of the first tape is at the left end of the input.
5. All other tape heads are at some arbitrary cell. Since tapes other than the first tape are completely blank, it does not matter where the head is placed initially all cells of these tapes "look" the same.

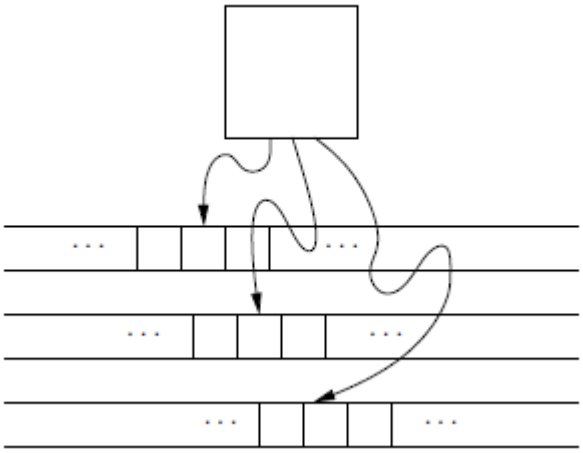


Figure: A multitape Turing machine

A move of the multitape TM depends on the state and the symbol scanned by each of the tape heads. In one move, the multitape TM does the following:

1. The control enters a new state, which could be the same as the previous state.
2. On each tape, a new tape symbol is written on the cell scanned. Any of these symbols may be the same as the symbol previously there.
3. Each of the tape heads makes a move, which can be either left, right, or stationary. The heads move independently, so different heads may move in different directions, and some may not move at all.

We shall not give the formal notation of transition rules, whose form is a straightforward generalization of the notation for the one tape TM, except that directions are now indicated by a choice of L, R, or S. For the one tape machine, we did not allow the head to remain stationary, so the S option was not present. You should be able to imagine an appropriate notation for instantaneous descriptions of the con guration of a multitape TM, we shall not give this notation formally. Multitape Turing machines, like one tape TM's accept by entering an accepting state.

Equivalence of One-Tape and Multitape TMs

The recursively enumerable languages are defined to be those accepted by a one tapeTM. Surely, multitape TMs accept all the recursively enumerable languages, since a one tape TM is a multitape TM. However, are there languages that are not recursively enumerable, yet are accepted by multitape TM's. The answer is "no", and we prove this fact by showing how to simulate a multitape TM by a one tape TM.

Theorem : Every language accepted by a multitape TM is recursively enumerable.

PROOF: The proof is suggested by below figure. Suppose language L is accepted by a k-tape TM M. We simulate M with a one tape TM N whose tape we think of as having 2k tracks. Half these tracks hold the tapes of M, and the other half of the tracks each hold only a single marker that indicates where the head for the corresponding tape of M is currently located. Figure assumes k = 2. The second and fourth tracks hold the contents of the first and second tapes of M, track 1 holds the position of the head of tape 1, and track 3 holds the position of the second tape head.

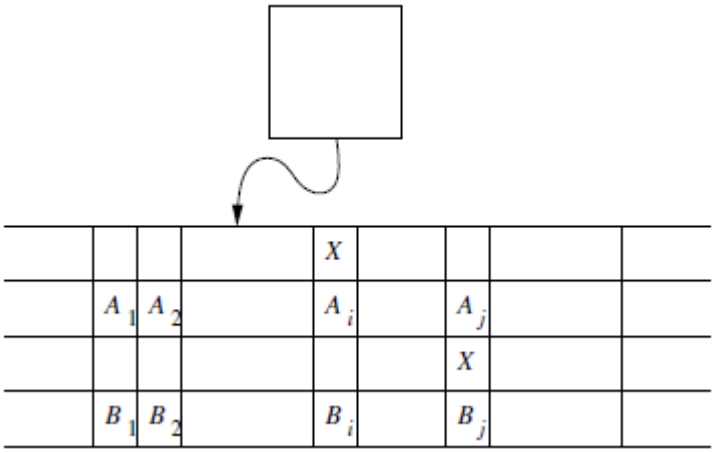


Figure: Simulation of a two tape Turing machine by a one tape Turing machine

To simulate a move of M Ns head must visit the k head markers. So that N not get lost, it must remember how many head markers are to its left at all times; that count is stored as a component of N's finite control. After visiting each head marker and storing the scanned symbol in a component of its finite control. N knows what tape symbols are being scanned by each of M's heads. N also knows the state of M, which it stores in N's own finite control. Thus, N knows what move M will make.

N now revisits each of the head markers on its tape, changes the symbol in the track representing the corresponding tapes of M, and moves the head markers left or right, if necessary. Finally, N changes the state of M as recorded in its own finite control. At this point, N has simulated

one move of M.

We select as N's accepting states all those states that record M's state as one of the accepting states of M. Thus, whenever the simulated M accepts, N also accepts, and N does not accept otherwise.

Nondeterministic Turing Machines

A nondeterministic Turing machine (NTM) differs from the deterministic variety by having a transition function δ such that for each state q and tape symbol X, δ (q, X) is a set of triples

$$\{(q_1, Y_1, D_1),\ (q_2, Y_2, D_2), \ldots, (q_k, Y_k, D_k)\}$$

where k is any finite integer. The NTM can choose, at each step, any of the triples to be the next move. It cannot, however, pick a state from one, a tape symbol from another, and the direction from yet another.

The language accepted by an NTM M is defined in the expected manner, in analogy with the other nondeterministic devices, such as NFA's and PDA's. That is, M accepts an input w if there is any sequence of choices of move that leads from the initial ID with w as input, to an ID with an accepting state. The existence of other choices that do not lead to an accepting state is irrelevant, as it is for the NFA or PDA.

The NTMs accept no languages not accepted by a deterministic TM (or DTM if we need to emphasize that it is deterministic). The proof involves showing that for every NTM M_N, we can construct a DTM M_D that explores the ID's that MN can reach by any sequence of its choices. If M_D finds one that has an accepting state, then M_D enters an accepting state of its own. M_D must be systematic, putting new ID's on a queue, rather than a stack, so that after some finite time M_D has simulated all sequences of up to k moves of M_N, for k = 1,2,……

Theorem: If M_N is a nondeterministic Turing machine, then there is a deterministic Turing machine M_D such that $L(M_N) = L(M_D)$

PROOF: M_D will be designed as a multitape TM, sketched in below figure. The first tape of M_D holds a sequence of ID's of M_N, including the state of M_N. One ID of M_N is marked as the "current" ID, whose successor ID's are in the process of being discovered. In figure the third ID is marked by an x along with the inter ID separator, which is the *. All ID's to the left of the current one have been explored and can be ignored subsequently.

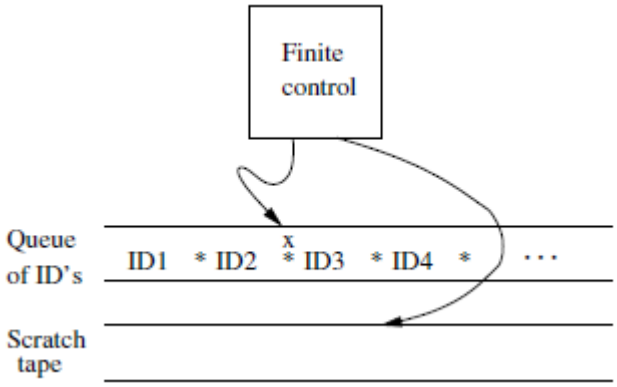


Figure: Simulation of an NTM by a DTM

To process the current ID, M_D does the following:

1. M_D examines the state and scanned symbol of the current ID. Built into the finite control of M_D is the knowledge of what choices of move M_N, has for each state and symbol. If the state in the current ID is accepting, then M_D accepts and simulates M_N no further.
2. However, if the state is not accepting, and the state symbol combination has k moves, then M_D uses its second tape to copy the ID and then make k copies of that ID at the end of the sequence of ID's on tape 1.
3. M_D modifies each of those k ID's according to a different one of the k choices of move that M_N has from its current ID.
4. M_D returns to the marked, current ID, erases the mark, and moves the mark to the next ID to the right. The cycle then repeats with step (1).

It should be clear that the simulation is accurate, in the sense that M_D will only accept if it finds that MN can enter an accepting ID. However, we need to confirm that if M_N enters an accepting ID after a sequence of n of its own moves, then M_D will eventually make that ID the current ID and will accept.

Suppose that m is the maximum number of choices M_N has in any configuration. Then there is one initial ID of M_N, at most m ID's that M_N can reach after one move, at most m^2 ID's MN can reach after two moves, and so on. Thus, after n moves, M_N can reach at most nm^m ID's.

Undecidability

A language that is not recursively enumerable & An Undecidable Problem That Is RE

Decidable

A problem P is decidable if it can be solved by a Turing machine T that always halt. (We say that P has an effective algorithm.)

Note that the corresponding language of a decidable problem is recursive.

Undecidable

A problem is undecidable if it cannot be solved by any Turing machine that halts on all inputs.

Note that the corresponding language of an undecidable problem is non-recursive.

Complements of Recursive Languages

Theorem: If L is a recursive language, L is also recursive.

Proof: Let M be a TM for L that always halt. We can construct another TM M from M for L that always halts as follows:

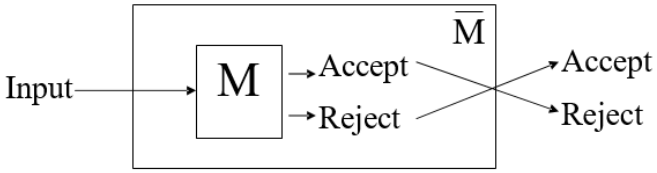


Figure: Construction of a TM accepting the complement of a recursive language

Complements of RE Languages

Theorem: If both a language L and its complement L are RE, L is recursive.

Proof: Let M1 and M2 be TM for L and L respectively. We can construct a TM M from M1 and M2 for L that always halt as follows:

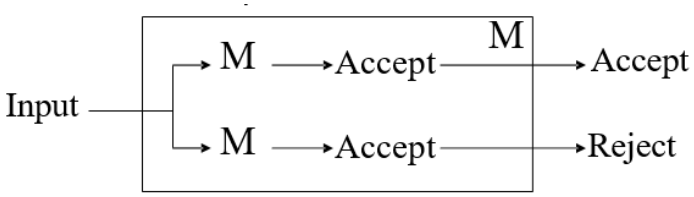


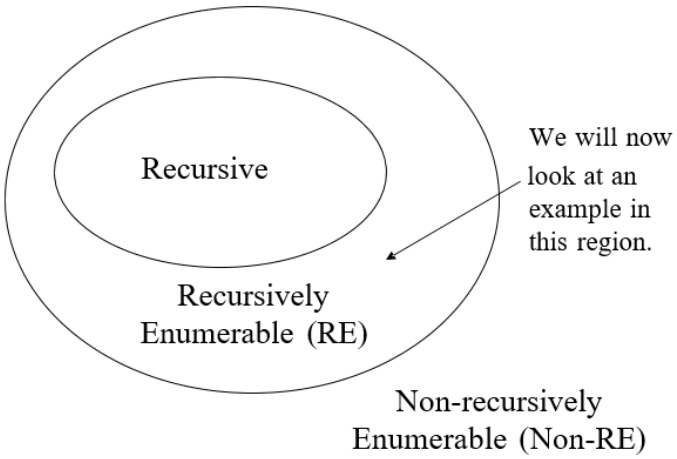
Figure: Simulation of two TM's accepting a language and its complement

A Non-recursive RE Language

- We are going to give an example of a RE language that is not recursive, i.e., a language L that

can be accepted by a TM, but there is no TM for L that always halt.

- Again, we need to make use of the binary encoding of a TM.



A Non-recursive RE Language

- Recall that we can encode each TM uniquely as a binary number and enumerate all TM's as T_1, T_2, …, T_k, … where the encoded value of the kth TM, i.e., T_k, is k.
- Consider the language Lu:

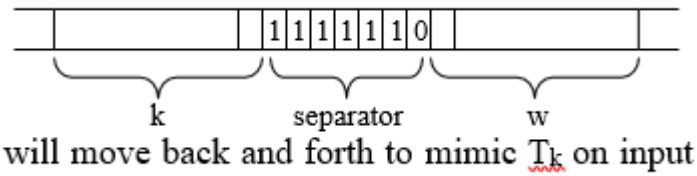
  Lu = {(k, w) | Tk accepts input w} This is called the *universal language*.

Universal Language

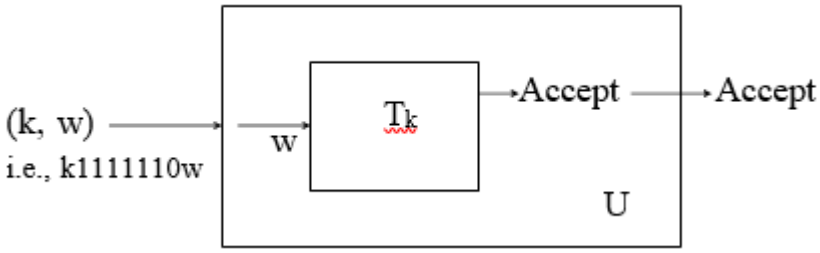
- Note that designing a TM to recognize L_u is the same as solving the problem of *given k and w, decide whether T_k accepts w as its input*.
- We are going to show that Lu is RE but non-recursive, i.e., Lu can be accepted by a TM, but there is no TM for Lu that always halt.

Universal Turing Machine

- To show that L_u is RE, we construct a TM U, called the *universal Turing machine*, such that L_u = L(U).
- U is designed in such a way that given k and w, it will mimic the operation of T_k on input w:



U will move back and forth to mimic T_k on input w.



Why cannot we use a similar method to construct a TM for L_d?

Universal Language

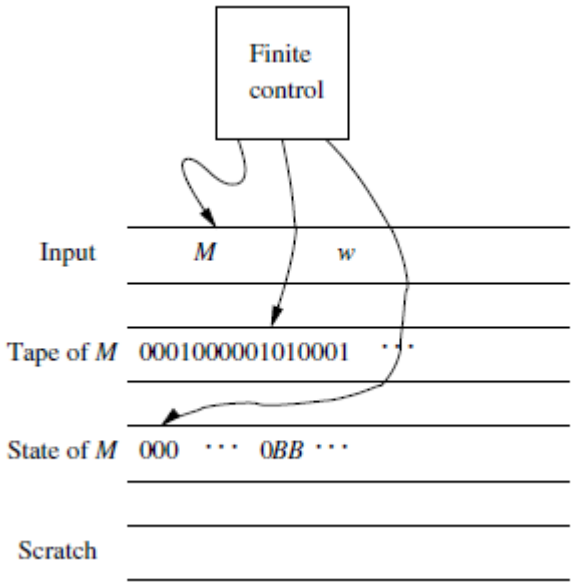
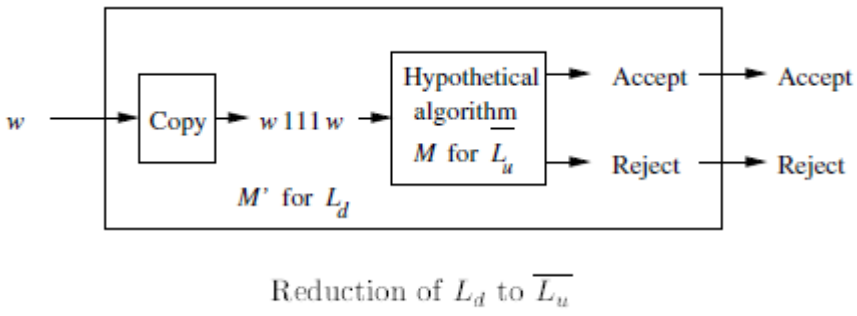


Figure: Organization of a universal Turing machine

- Since there is a TM that accepts L_u, L_u is RE. We are going to show that L_u is non- recursive.

If L_u is recursive, there is a TM M for L_u that always halt. Then, we can construct a TM M' for L_d as follows:



Reduction of L_d to $\overline{L_u}$

A Non-recursive RE Language

- Since we have already shown that Ld is non-recursively enumerable, so M' does not exist and there is no such M.
- Therefore the universal language is recursively enumerable but non-recursive.

Halting Problem

Consider the halting problem:

Given (k,w), determine if T_k halts on w.

It's corresponding language is: L_h = { (k, w) | Tk halts on input w}

- The halting problem is also undecidable, i.e., L_h is non-recursive. To show this, we can make use of the universal language problem.
- We want to show that if the halting problem can be solved (decidable), the universal language problem can also be solved.
- So we will try to reduce an instance (a particular problem) in Lu to an instance in L_h in such a way that if we know the answer for the latter, we will know the answer for the former.

Consider a particular instance (k,w) in Lu, i.e., we want to determine if Tk will accept w. Construct an instance I=(k',w') in Lh from (k,w) so that if we know whether Tk' will halt on w', we will know whether Tk will accept w.

Halting Problem: Therefore, if we have a method to solve the halting problem, we can also solve the universal language problem. (Since for any particular instance I of the universal language problem, we can construct an instance of the halting problem, solve it and get the answer for I.) However, since the universal problem is undecidable, we can conclude that the halting problem is also undecidable.

Syntax directed definition

Q: What is syntax directed translation? Discuss S-attributes and L-attributes.

There are two ways to represent the semantic rules associated with grammar symbols.

• Syntax-Directed Definitions (SDD)

• Syntax-Directed Translation Schemes (SDT)

Syntax-directed translation (SDT) refers to a method of compiler implementation where the source language translation is completely driven by the parser, i.e., based on the syntax of the language. The parsing process and parse trees are used to direct semantic analysis and the translation of the source program. Almost all modern compilers are syntax-directed. SDT can be a separate phase of a compiler or we can augment our conventional grammar with information to control the semantic analysis and translation. Such grammars are called **attribute grammars**.

We augment a grammar by associating **attributes** with each grammar symbol that describes its properties. With each production in a grammar, we give **semantic rules/ actions**, which describe how to compute the attribute values associated with each grammar symbol in a production. The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree.

A class of syntax-directed translations called "L-attributed translations" (L for left-to right) includes almost all translations that can be performed during parsing. Similarly, "S-attributed translations" (S for synthesized) can be performed easily in connection with a bottom-up parse.

Syntax-Directed Definitions

A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. An attribute has a name and an associated value: a string, a number, a type, a memory location, an assigned register, strings. The strings may even be long sequences of code, say code in the intermediate language used by a compiler. If X is a symbol and a is one of its attributes, then we write X.a to denote the value of a at a particular parse-tree node labeled X. If we implement the nodes of the parse tree by records or objects, then the attributes of X can be implemented by data fields in the records that represent the nodes for X. The attributes are evaluated by the semantic rules attached to the productions.

Example: PRODUCTION	SEMANTIC RULE
$E \rightarrow E1 + T$	E.code = E1.code || T.code || '+'

SDDs are highly readable and give high-level specifications for translations. But they hide many implementation details. For example, they do not specify order of evaluation of semantic actions.

Syntax-Directed Translation Schemes (SDT)

SDT embeds program fragments called semantic actions within production bodies. The position of semantic action in a production body determines the order in which the action is executed.

Example: In the rule $E \rightarrow E1 + T$ { print '+' }, the action is positioned after the body of the production. SDTs are more efficient than SDDs as they indicate the order of evaluation of semantic actions associated with a production rule. This also gives some information about implementation details.

Inherited and Synthesized Attributes

Terminals can have synthesized attributes, which are given to it by the lexer (not the parser). There are no rules in an SDD giving values to attributes for terminals. Terminals do not have inherited attributes.

A nonterminal A can have both inherited and synthesized attributes. The difference is how they are computed by rules associated with a production at a node N of the parse tree.

- A **synthesized attribute** for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head.

A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.

- An **inherited attribute** for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. Note that the production must have B as a symbol in its body.

An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

An inherited attribute at node N cannot be defined in terms of attribute values at the children of node N. However, a synthesized attribute at node N can be defined in terms of inherited attribute

values at node N itself.

Q: Develop the SDD and construct the annotated parse tree for the input string 3*5+4n for the grammar using synthesized attributes.

S → E n

E → E+T | T

T → T * F | F

F → (E) | digit

The SDD on grammar for arithmetic expressions with operators + and *. It evaluates expressions terminated by an endmarker n. In the SDD, Each of the nonterminals has a single synthesized attribute, called val. We also suppose that the terminal digit has a synthesized attribute lexval, which is an integer value returned by the lexical analyzer.

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

Syntax-directed definition of a simple desk calculator

An SDD that involves only synthesized attributes is called **S-attributed**; the SDD has this property. In an S-attributed SDD, each rule computes an attribute for the nonterminal at the head of a production from attributes taken from the body of the production.

Attribute Grammar: An SDD without side effects is sometimes called an attribute grammar. The rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants.

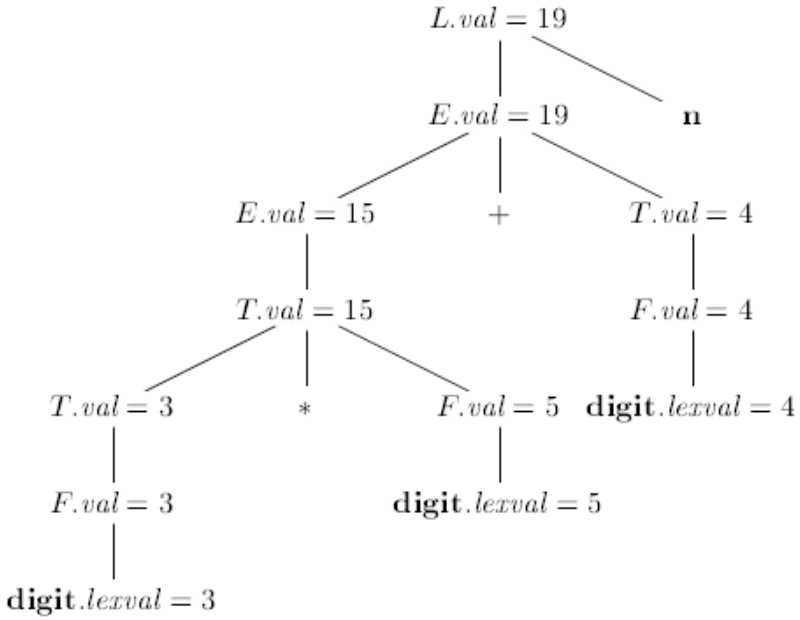
Evaluating an SDD at the Nodes of a Parse Tree

Parse tree helps us to visualize the translation specified by SDD. The rules of an SDD are applied by first constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodes of the parse tree. A parse tree, showing the value(s) of its attribute(s) is called an

annotated parse tree.

With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a postorder traversal of the parse tree.

Annotated Parse Tree for 3*5+4n



Annotated parse tree for 3 * 5 + 4 n

Inherited attributes are useful when the structure of a parse tree does not match the abstract syntax of the source code. They can be used to overcome the mismatch due to grammar designed for parsing rather than translation. In the SDD below, the nonterminal T` has an inherited attribute *inh* as well as a synthesized attribute val. T` inherits F.val from its left sibling F in the production T F T`.

Q: Develop the SDD and construct the annotated parse tree for the input string 3*5 for the grammar with inherited attributes.

T → F T'

T' → * F T'

T' → €

F → digit

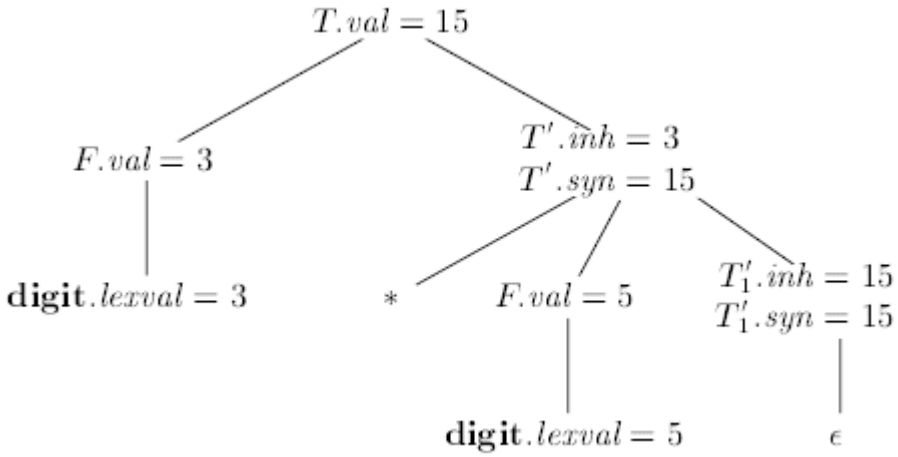
SDD for expression grammar with inherited attributes:

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

An SDD based on a grammar suitable for top-down parsing

Annotated Parse Tree for 3*5 using the above SDD is as below.

An SDD with both inherited and synthesized attributes does not ensure any guaranteed order; even it may not have an order at all.



Annotated parse tree for 3 * 5

Q: Explain how Dependency graphs are used in evaluation orders for SDD's. Explain the construct ion of the Dependency graph from the annotated parse tree in S-attributed and L-attributed SDD with suitable example.

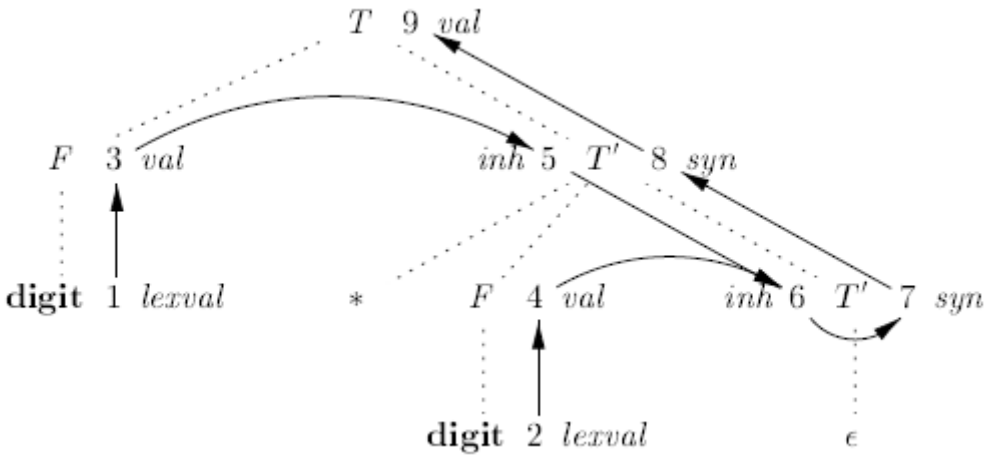
"**Dependency graphs**" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree. While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.

A dependency graph shows the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first

is needed to compute the second. Edges express constraints implied by the semantic rules.

- Each attribute is associated to a node
- If a semantic rule associated with a production *p defines the* value of synthesized attribute *A.b in terms of the value of X.c, then graph has an edge from X.c to A.b*
- If a semantic rule associated with a production *p defines the* value of inherited attribute *B.c in terms of value of X.a, then* graph has an edge from *X.a to B.c*

Dependency graph for the annotated parse tree for 3*5



Topological Sort of the Dependency Graph

A dependency graph characterizes the possible order in which we can evaluate the attributes at various nodes of a parse tree. If there is an edge from node M to N, then attribute corresponding to M first be evaluated before evaluating N. Thus the only allowable orders of evaluation are N1, N2, ..., Nk such that if there is an edge from Ni to Nj 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 are no topological sorts; that is, there is no way to evaluate the SDD on this parse tree. If there are no cycles, however, then there is always at least one topological sort.

S-Attributed Definitions

An SDD is *S-attributed* if every attribute is synthesized. Attributes of an S-attributed SDD can be evaluated in bottom-up order of the nodes of parse tree. Evaluation is simple using post-order traversal.

```
postorder(N) {
        for (each child C of N, from the left)
        postorder(C);
        evaluate attributes associated with node N;
}
```

S-attributed definitions can be implemented during bottom-up parsing since bottom-up parse corresponds to a postorder traversal. Postorder corresponds to the order in which an LR parser reduces a production body to its head.

Example: SDD for simple desk calculator is S-attributed.

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

Syntax-directed definition of a simple desk calculator

L-Attributed Definitions

The idea behind this class is that, between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left(hence "L-attributed"). Each attribute must be either

1. Synthesized, or
2. Inherited,

But with the rules limited as follows. Suppose that there is a production $A \rightarrow X1\ X2 \bullet \bullet \bullet Xn$, and that there is an inherited attribute $Xi.a$ computed by a rule associated with this production. Then the rule may use only:

(a) Inherited attributes associated with the head *A.*

(b) Either inherited or synthesized attributes associated with the occurrences of symbols *X1 X2* • • • *Xi-1* located to the left of *Xi*

(c) Inherited or synthesized attributes associated with this occurrence of Xi itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this Xi.

Example: The following definition shown below is L-attributed. Here the inherited attribute of T` gets its values from its left sibling F. Similarly, T1` gets its value from its parent T` and left sibling F.

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\,F\,T_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

Example: the definitions below are not L-attributed as B.i depends on its right sibling C's attribute.

Production	semantic rules
A→ BC	A.s = B.b;
	B.i = f(C.c, A.s)

Q: Discuss the side effects in evaluation of semantic rules.

Side Effects: Evaluation of semantic rules may generate intermediate codes, may put information into the symbol table, may perform type checking and may issue error messages. These are known as side effects.

Semantic Rules with Controlled Side Effects:

In practice translation involves side effects. Attribute grammars have no side effects and allow any evaluation order consistent with dependency graph whereas translation schemes impose left-to-right evaluation and allow schematic actions to contain any program fragment.

Ways to Control Side Effects

1. Permit incidental side effects that do not disturb attribute evaluation.
2. Impose restrictions on allowable evaluation orders, so that the same translation is produced for any allowable order.

Q: Give the syntax directed definition to process a sample variable declaration in C and construct the dependency graph for the input float x,y,z

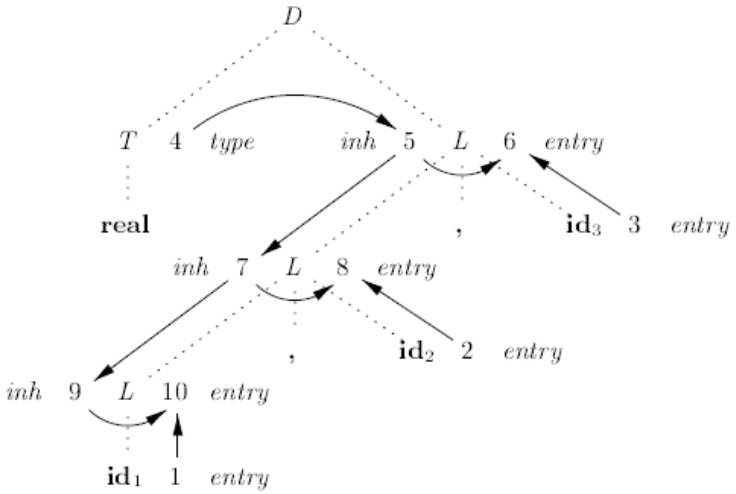
SDD for Simple Type Declarations

	PRODUCTION	SEMANTIC RULES
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow \mathbf{int}$	$T.type = \text{integer}$
3)	$T \rightarrow \mathbf{float}$	$T.type = \text{float}$
4)	$L \rightarrow L_1\ ,\ \mathbf{id}$	$L_1.inh = L.inh$ $addType(\mathbf{id}.entry, L.inh)$
5)	$L \rightarrow \mathbf{id}$	$addType(\mathbf{id}.entry, L.inh)$

Nonterminal *D* represents a declaration, which, from production 1, consists of a type *T* followed by a list *L* of identifiers. *T* has one attribute, *T.type,* which is the type in the declaration *D.* Nonterminal *L* also has one attribute, which we call *ink* to emphasize that it is an inherited attribute. The purpose of *L.inh* is to pass the declared type down the list of identifiers, so that it can be the appropriate symbol-table entries. Productions 2 and 3 each evaluate the synthesized attribute *T.type,* giving it the appropriate value, integer or float. This type is passed to the attribute *L.inh* in the rule for production 1. Production 4 passes *L.inh* down the parse tree. That is, the value L1.*inh* is computed at a parse-tree node by copying the value of *L.inh* from the parent of that node; the parent corresponds to the head of the production. Productions 4 and 5 also have a rule in which a function *addType* is called with two arguments:

1. **id.***entry,* a lexical value that points to a symbol-table object, and
2. *L.inh,* the type being assigned to every identifier on the list.

The function *addType* properly installs the type *L.inh* as the type of the represented identifier. Note that the side effect, adding the type info to the table, does not affect the evaluation order. A dependency graph for the input string **float id1 , id 2, id3** is shown below.



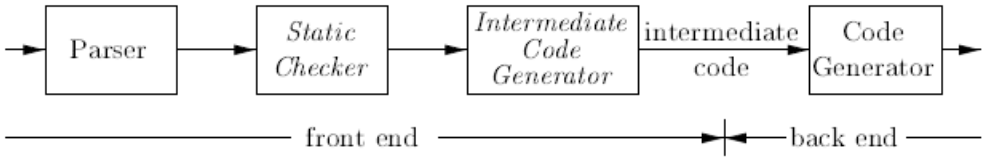
Dependency graph for a declaration float id_1,id_2,id_3

Intermediate Code Generation

Q: Explain intermediate code generation phase in compiler construction.

In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. This facilitates *retargeting*: enables attaching a back end for the new machine to an existing front end.

Logical Structure of a Compiler Front End



A compiler front end is organized as in shown above, where parsing, static checking, and intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing. All schemes can be implemented by creating a syntax tree and then walking the tree.

Static Checking

This includes type checking which ensures that operators are applied to compatible operands. It also includes any syntactic checks that remain after parsing like

• flow–of-control checks

– Ex: Break statement within a loop construct

• Uniqueness checks

– Labels in case statements

• Name-related checks

Intermediate Representations

We could translate the source program directly into the target language. However, there are benefits to having an intermediate, machine-independent representation.

- A clear distinction between the machine-independent and machine-dependent parts of the compiler
- Retargeting is facilitated; the implementation of language processors for new machines will require replacing only the back-end
- We could apply machine independent code optimization techniques Intermediate representations span the gap between the source and target languages.

- *High Level Representations*
  - closer to the source language
  - easy to generate from an input program
  - code optimizations may not be straightforward
- *Low Level Representations*
  - closer to the target machine
  - Suitable for register allocation and instruction selection
  - easier for optimizations, final code generation

There are several options for intermediate code. They can be either

- Specific to the language being implemented

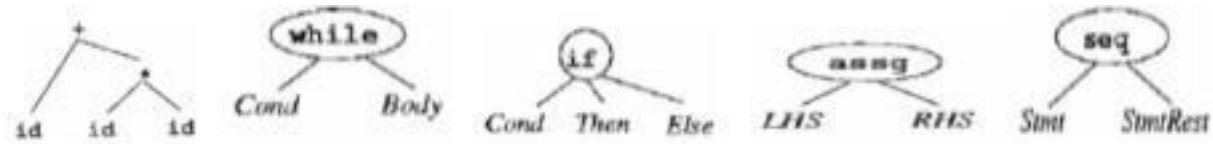
  *P-code for Pascal * Bytecode for Java

- Language independent:

  *3-address code

IR can be either an actual language or a group of internal data structures that are shared by the phases of the compiler. C used as intermediate language as it is flexible, compiles into efficient machine code and its compilers are widely available. In all cases, the intermediate code is a linearization of the syntax tree produced during syntax and semantic analysis. It is formed by breaking down the tree structure into sequential instructions, each of which is equivalent to a single, or small number of machine instructions. Machine code can then be generated (access might be required to symbol tables etc).

Syntax trees:



Syntax trees are high level IR. They depict the natural hierarchical structure of the source program. Nodes represent constructs in source program and the children of a node represent meaningful components of the construct. Syntax trees are suited for static type checking.

Q: Explain the variants of Syntax Trees: DAG. Construct DAG for

a. id+id*id

b. a + a * (b - c) + (b - c) * d

for the grammar

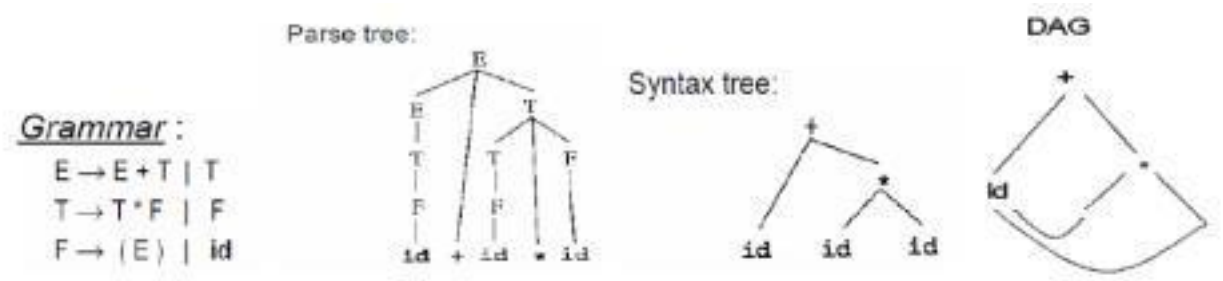
E → E + T | T

T → T * F | F

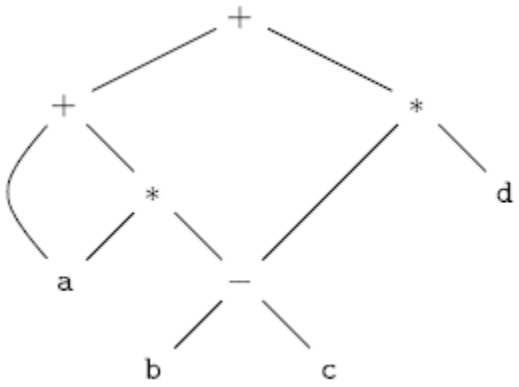
F → (E) | id

A directed acyclic graph (*DAG)* for an expression identifies the *common sub expressions* (sub expressions that occur more than once) of the expression. DAG's can be constructed by using the same techniques that construct syntax trees. A DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators. A node N in a DAG has more than one parent if N represents a common sub expression, so a DAG represents expressions concisely. It gives clues to compiler about the generating efficient code to evaluate expressions.

Given the grammar below, for the input string **id + id * id** , the parse tree, syntax tree and the DAG are as shown.



DAG for the expression a + a * (b - c) + (b - c) * d is shown below.



Dag for the expression a +a*(b-c)+ (b-c)*d

Q: Explain different methods to construct syntax tree/DAG:

1) Using the SDD to draw syntax tree or DAG for a given expression:-

- Draw the parse tree
- Perform a post order traversal of the parse tree
- Perform the semantic actions at every node during the traversal

– Creates a syntax tree if a new node is created each time functions Leaf and Node are called

– Constructs a DAG if before creating a new node, these functions check whether an identical node already exists. If yes, the existing node is returned.

SDD to produce Syntax trees or DAG is shown below.

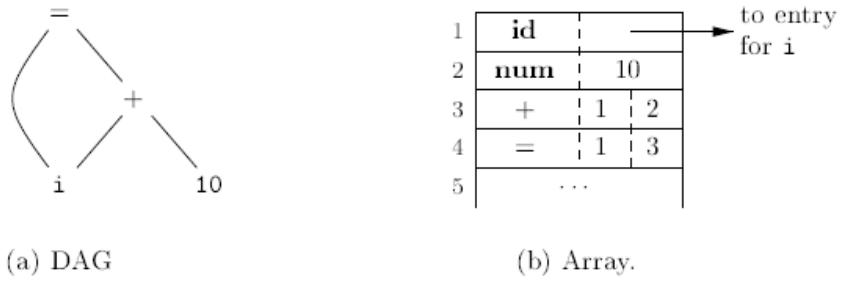
	PRODUCTION	SEMANTIC RULES
1)	$E \to E_1 + T$	$E.node = \textbf{new } Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \textbf{new } Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (\ E \ )$	$T.node = E.node$
5)	$T \to \textbf{id}$	$T.node = \textbf{new } Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \to \textbf{num}$	$T.node = \textbf{new } Leaf(\textbf{num}, \textbf{num}.val)$

For the expression a + a * (b – c) + (b - c) * d, steps for constructing the DAG is as below:

1) $p_1 = Leaf(\textbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\textbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\textbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\textbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\textbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\textbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\textbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

2) 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.



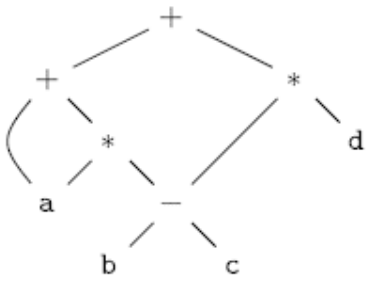
(a) DAG (b) Array.

Nodes of a DAG for i = i + 10 allocated in an array

The signature of a node is a triple < op, l, r> 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.

Q: Define three address code. Explain Three Address Code Instructions

- **Three Address Code** can range from high- to low-level, depending on the choice of operators. In general, it is a statement containing at most 3 addresses or operands.
- The general form is x := y op z, where "op" is an operator, x is the result, and y and z are operands. **x**, **y**, **z** are variables, constants, or "temporaries". A three-address instruction consists of at most 3 addresses for each statement.
- It is a linearized representation of a binary syntax tree. Explicit names correspond to interior nodes of the graph. E.g. for a looping statement, syntax tree represents components of the statement, whereas three-address code contains labels and jump instructions to represent the flow-of-control as in machine language.



(a) DAG

$t_1 = b - c$
$t_2 = a * t_1$
$t_3 = a + t_2$
$t_4 = t_1 * d$
$t_5 = t_3 + t_4$

(b) Three-address code

DAG and its corresponding three-address code

A **Three Address Code** instruction has at most one operator on the RHS of an instruction; no built-up arithmetic expressions are permitted.

Example: x + y * z can be translated as

t1 = y * z

t2 = x + t1

where t1 & t2 are compiler–generated temporary names.

Since it unravels multi-operator arithmetic expressions and nested control-flow statements, it is useful for target code generation and optimization.

Addresses and Instructions

• TAC consists of a sequence of instructions, each instruction may have up to three addresses, prototypically t1 = t2 op t3

a) **Addresses** may be one of:

i) A name. Each name is a symbol table index. For convenience, we write the names as the identifier. Ex: a, b…..z, sum, num, avg etc.

ii) A constant. Ex: 10, 'a', "hello" etc.

iii) A compiler-generated temporary. Each time a temporary address is needed, the compiler generates another name from the stream t1, t2, t3, etc.

Temporary names allow for code optimization to easily move instructions. At target-code generation time, these names will be allocated to registers or to memory.

Three Address Code Instructions

1) Assignment instructions: x = y op z. It includes binary arithmetic and logical operations.
2) Unary assignments: x = op y. This includes unary arithmetic op (-) and logical op (!,&,|) and type conversion.
3) Copy instructions: x = y
4) Unconditional jump: goto L. L is a symbolic label of an instruction
5) Conditional jumps:
   if x goto L If x is true, execute instruction L next
   ifFalse x goto L If x is false, execute instruction L next
6) Conditio
   nal
   jumps: if
   x relop y
   goto L

7) Procedure calls. For a procedure, call

p(x1, …, xn) For

parameters, param x1

…

param xn

for functions, y= call p

return y is optional.

8) Indexed copy instructions: x = y[i] and x[i] = y

• Left: sets x to the value in the location i memory units beyond y

• Right: sets the contents of the location i memory units beyond x to y

9) Address and pointer instructions:

• x = &y sets the value of x to be the location (address) of y.

• x = *y, presumably y is a pointer or temporary whose value is a location. The value of x is set to the contents of that location.

• *x = y sets the value of the object pointed to by x to the value of y.

Example: Given the statement **do i = i+1; while (a[i] < v); ,** the TAC can be written as below in two ways, using either symbolic labels or position number of instructions for labels.

```
L:   t1 = i + 1
     i = t1
     t2 = i * 8
     t3 = a [ t2 ]
     if t3 < v goto L
```

(a) Symbolic labels.

```
100:  t1 = i + 1
101:  i = t1
102:  t2 = i * 8
103:  t3 = a [ t2 ]
104:  if t3 < v goto 100
```

(b) Position numbers.

Two ways of assigning labels to three-address statements

Q. Explain the different types of representation of 3-address code

Data structures for representation of TAC can be objects or records with fields for operator and operands.

Representations include **quadruples, triples and indirect triples**.

a. Quadruples

• In the quadruple representation, there are four fields for each instruction: *op, arg1, arg2, result.*

– Binary ops have the obvious representation

– Unary ops don't use arg2

– Operators like param don't use either arg2 or result

– Jumps put the target label into result

• The quadruples in Fig (b) implement the three-address code in (a) for the expression a = b * - c + b * - c

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

Three-address code and its quadruple representation

b. Triples

• A triple has only three fields for each instruction: *op, arg1, arg2*

• The *result* of an operation *x op y* is referred to by its position.

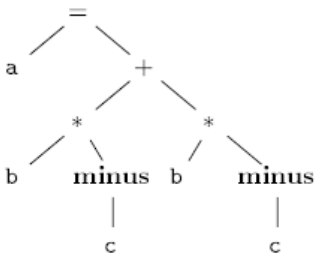
• Triples are equivalent to signatures of nodes in DAG or syntax trees.

• Triples and DAGs are equivalent representations only for expressions; they are not equivalent for control flow.

• Ternary operations like ***x[i]*** = *y* requires two entries in the triple structure, similarly for x = y[i].

• Moving around an instruction during optimization is a problem

Example: Representations of a =b *– c + b * – c



(a) Syntax tree

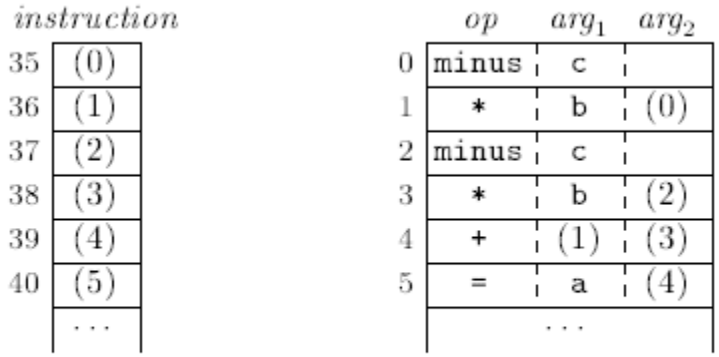
	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

(b) Triples

Representations of a = b * - c + b * - c ;

c. Indirect Triples

These consist of a listing of pointers to triples, rather than a listing of the triples themselves. An optimizing compiler can move an instruction by reordering the instruction list, without affecting the triples themselves.



instruction	
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

Indirect triples representation of three-address code

d. Static Single-Assignment Form

Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations. Two distinctive aspects distinguish SSA from three-address code.

• All assignments in SSA are to variables with distinct names; hence *static singleassignment.*

3 address code	SSA
p = a+b	p1=a+b
q=p-c	q1=p1-c
p=q*d	p2=q1*d
p=e-p	p3=e-p2
q=p+q	q2=p3+q1

Code generation

Q13: Discuss the issues in the design of code generator

The final phase in our compiler model is the code generator. It takes as input the intermediate representation (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces as output a semantically equivalent target program, as shown in Fig. 8.1.

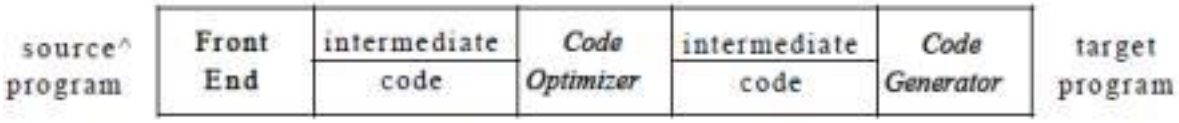


Figure 8.1: Position of code generator

The requirements imposed on a code generator are severe. The target program must preserve the semantic meaning of the source program and be of high quality; that is, it must make effective use of the available resources of the target machine. Moreover, the code generator itself must run efficiently.

Issues in the Design of a Code Generator

1. Input to the Code Generator

- The input to the code generator is the intermediate representation of the source program produced by the front end, along with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the IR.
- The many choices for the IR include three-address representations such as quadruples, triples, indirect triples; virtual machine representations such as byte codes and stack-machine code; linear representations such as postfix notation; and graphical representations such as syntax trees and DAG's.
- We assume that the front end has scanned, parsed, and translated the source program into a relatively low-level IR, so that the values of the names appearing in the IR can be represented by quantities that the target machine can directly manipulate, such as integers and floating-point numbers. We also assume that all syntactic and static semantic errors have been detected, that the necessary type checking has taken place, and that type conversion operators have been inserted wherever necessary. The code generator can therefore proceed on the assumption that its input is free of these kinds of errors.

2. The Target Program

The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code. The most common

target-machine architectures are

1) RISC (reduced instruction set computer): A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture.
2) CISC (complex instruction set computer): CISC machine typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.
3) Stack based: In a stack-based machine, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack. To achieve high performance the top of the stack is typically kept in registers. Stack-based machines almost disappeared because it was felt that the stack organization was too limiting and required too many swap and copy operations.
    - However, stack-based architectures were revived with the introduction of the Java Virtual Machine (JVM). The JVM is a software interpreter for Java bytecodes, an intermediate language produced by Java compilers. The inter preter provides software compatibility across multiple platforms, a major factor in the success of Java.
    - To overcome the high performance penalty of interpretation, which can be on the order of a factor of 10, *just-in-time* (JIT) Java compilers have been created. These JIT compilers translate bytecodes during run time to the native hardware instruction set of the target machine. Another approach to improving Java performance is to build a compiler that compiles directly into the machine instructions of the target machine, bypassing the Java bytecodes entirely.
    - Producing an absolute machine-language program as output has the advantage that it can be placed in a fixed location in memory and immediately executed. Programs can be compiled and executed quickly.
    - Producing a relocatable machine-language program (often called an *object module)* as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader. Although we must pay the added expense of linking and loading if we produce relocatable object modules, we gain a great deal of flexibility in being able to compile subroutines separately and to call other previously compiled programs from an object

module. If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled program modules.

3. Instruction Selection

The code generator must map the IR program into a code sequence that can be executed by the target machine. The complexity of performing this mapping is determined by a factors such as

- The level of the IR: If the IR is high level, the code generator may translate each IR statement into a sequence of machine instructions using code templates. Such statement by- statement code generation, however, often produces poor code that needs further optimization. If the IR reflects some of the low-level details of the underlying machine, then the code generator can use this information to generate more efficient code sequences.
- The nature of the instruction-set architecture: The nature of the instruction set of the target machine has a strong effect on the difficulty of instruction selection. For example, the uniformity and completeness of the instruction set are important factors. If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling. On some machines, for example, floating-point operations are done using separate registers. Instruction speeds and machine idioms are other important factors. If we do not care about the efficiency of the target program, instruction selection is straightforward. For each type of three-address statement, we can design a code skeleton that defines the target code to be generated for that construct. For example, every three-address statement of the form x = y + z, where x, y, and z are statically allocated, can be translated into the code sequence

```
LD RO, y          // RO = y (load y into register RO)
ADD RO, RO, z     // RO = RO + z (add z t o RO)
ST x, RO          // x = RO (store RO into x)
```

This strategy often produces redundant loads and stores. For example, the sequence of three-address statements

```
a = b + c
d = a + e
```

would be translated into

```
LD RO, b  / / RO = b
ADD RO, RO, c  / / RO = RO
```

ST **a,** RO // **a** = = RO

LD RO, **a** / / RO = **a**

ADD RO, RO, e / / RO = RO

ST **d,** RO // **d** = = RO

Here, the fourth statement is redundant since it loads a value that has just been stored, and so is the third if **a** is not subsequently used.

- The desired quality of the generated code: The quality of the generated code is usually determined by its speed and size. On most machines, a given IR program can be implemented by many different code sequences, with significant cost differences between the different implementations. A naive translation of the intermediate code may therefore lead to correct but unacceptably inefficient target code. For example, if the target machine has an "increment" instruction (INC), then the three-

  address statement a = a + 1 may be implemented more efficiently by the single instruction INC a, rather than by a more obvious sequence that loads a into a register, adds one to the register, and then stores the result back into **a:**

```
LD R0, a          // RO = a
ADD RO, RO, #1    // RO = RO + 1
ST a, RO          // a = RO
```

4. Register Allocation

A key problem in code generation is deciding what values to hold in what registers. Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values. Values not held in registers need to reside in memory. Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.

The use of registers is often subdivided into two sub problems:

1. ***Register allocation,*** during which we select the set of variables that will reside in registers at each point in the program.
2. ***Register assignment***, during which we pick the specific register that a variable will reside in.

Finding an optimal assignment of registers to variables is difficult, even with single-register machines. Mathematically, the problem is NP-complete. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register-

usage conventions be observed.

Example: Certain machines require *register-pairs* (an even and next odd numbered register) for some operands and results. For example, on some machines, integer multiplication and integer division involve register pairs. The multiplication instruction is of the form M x, y

where x, the multiplicand, is the even register of an even/odd register pair and y, the multiplier, is the odd register. The product occupies the entire even/odd register pair. The division instruction is of the form **D** x, y where the dividend occupies an even/odd register pair whose even register is x; the divisor is y. After division, the even register holds the remainder and the odd register the quotient. Now, consider the two three-address code sequences shown below in which the only difference in (a) and (b) is the operator in the second statement. The shortest assembly-code sequences for (a) and (b) are given

t = a + b	t = a + b
t = t * c	t = t + c
t = t / d	t = t / d
(a)	(b)

Two three-address code sequences

L Rl,a	L RO, a
A Rl,b	A RO, b
M R0,c	A RO, c
D R0,d	SRDA RO, 32
ST Rl,t	D RO, d
ST Rl, t	
(a)	(b)

Optimal machine-code sequences

Ri stands for register *i.* SRDA stands for Shift-Right-Double-Arithmetic and SRDA RO, 32 shifts the dividend into Rl and clears RO so all bits equal its sign bit. L, ST, and A stand for load, store, and add, respectively. Note that the optimal choice for the register into which a is to be loaded depends on what will ultimately happen to t.

5. Evaluation Order

The order in which computations are performed can affect the efficiency of the target code. As we shall see, some computation orders require fewer registers to hold intermediate results than others. However, picking a best order in the general case is a difficult NP-complete problem. Initially, we shall avoid the problem by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator.

University Questions/Question bank

1 Construct a Turning machine that accept the language 0^n, 1^n where n > 1 and draw transition for Turning Machine.

2 Explain the working principle of Turning Machine with diagram. Design a Turing Machine to accept strings formed on {0, 1} and ending with 000. Write transition diagram and ID for w = 101000

3. Design a Turing machine to accept L={ $0^n1^n2^n$ | n>=0}. Draw the transition diagram.

4. Design a Turing Machine to accept the language L={a^n b^n c^n | n>=1}. Draw the transition diagram. Show the moves made by this turing machine for the string aabbcc.

5. Design a turing machine to accept the language L={0^n 1^n 2^n | n>=1}.

6. Design a turing machine to accept strings of a's and b's ending with ab or ba.

7. Obtain a Turing machine to accept the language L = { 0^n 1^n | n>=1 }.

8.Design a turing machine to accept the following language L={o^n1^n|n≥1}. Write its transition diagram and given instantaneous description for input 0011.

9. Design a turing machine to accept the language L={ww^R : w€ (a, b)*} Write its transition diagram. Also show the sequence of moves made by the TM for the string "aabbaa".

10. Design a Turing machine to accept L = { $a^nb^nc^n$ | n ≥ 0 }

11 Design a Turing machine to accept the set of all palindromes over { 0 ,1}*. Write the transition diagram for the constructed Turing machine and write the sequence of ID's for the input string '1001' .

12. Define a turing machine and explain with neat diagram, the working of a basic turing machine.

13. Design a turing machine to accept the set of all palindrome over {a,b}*. Also, indicate the moves made by turing machine for the string aba.

14. Explain restricted Turing machines.

15. Explain the following with example:

i.Decidability

ii.Decidable languages

iii.Undecidable languages

16. Write short notes on:

a. Multi-tape turing machine.

b. Non-deterministic turing machine.

17. Write short notes on:

a. Undecidable languages.

b. Halting problem of turing machine.

18. a) Explain the following:

i.Non deterministic turing machine

ii.Multi-tape turing machine

b) Define the following:

i.Recursively enumerable language

ii.Decidable language

19. Write a detailed note on halting problem of Turing machine.

20. Write a note on universal Turing machine and show that simulate a computer.

21: What is syntax directed translation? Discuss S-attributes and L-attributes.

22: Develop the SDD and construct the annotated parse tree for the input string 3*5+4n for the grammar using synthesized attributes.

$S \rightarrow E\ n$

$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid digit$

23: Develop the SDD and construct the annotated parse tree for the input string 3*5 for the grammar with inherited attributes.

$T \rightarrow F\ T'$

$T' \rightarrow * F\ T'$

$T' \rightarrow €$

$F \rightarrow digit$

24: Explain how Dependency graphs are used in evaluation orders for SDD's. Explain the construct ion of the Dependency graph from the annotated parse tree in S-attributed and L-attributed SDD with suitable example.

25: Discuss the side effects in evaluation of semantic rules.

26: Give the syntax directed definition to process a sample variable declaration in C and construct the dependency graph for the input float x,y,z

27. What is syntax tree? Construct Syntax tree for a-4+c using the suitable SDD

28: Explain intermediate code generation phase in compiler construction.

29: Explain the variants of Syntax Trees: DAG. Construct DAG for

a. id+id*id

b. a + a * (b - c) + (b - c) * d

for the grammar

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

30: Explain different methods to construct syntax tree/DAG:

31: Define three address code. Explain Three Address Code Instructions

32. Explain the different types of representation of 3-address code

33: Discuss the issues in the design of code generator

34. Explain the construction of a Simple Target Machine Model

35. Explain Program and Instruction Costs