Study Guide

For

Formal Language Theory and Compiler Design and Analysis

Department of Computer Science

Faculty of Informatics

Prepared by

Sebsibe Hailemariam (PhD)

January 2012

Summary

Computer Science deals with the theoretical and practical foundations of computation as well as their implementation in computer system so as to address critical and vital issues of the discipline. Computer Science gives more emphasis to language representation and analysis among the various issues that the field is concerned on. Language as a means of communication and formalism enables standard mechanism of data representation and processing that plays major role in the successful trends in computing discipline.

Formal language is a mathematical formulation of language representation which is the most appropriate model to be adopted for computational purpose. Compiler is a language translator which translates a set of statement organized in some high level programming language into machine language that can be executed by the computing devices. These courses are corner stone to Computer Science students as they provide diverse insight into the field.

This guideline is not a reference that students should use during the due time of the courses formal language and compiler design. Rather it can be used for the students as a checklist to see what is expected from the students to know and what they already know. Hence, this guideline clearly shows the students the existence of any gap among the expected knowledge, skill and attitude to be achieved at the end of the course and the reality.

This guideline is also relevant not only to students but also for instructors that teaches these courses. Instructors should use this guideline as a check list to periodically assess their status while teaching the course and plan a head how to cover the important topics identified and needed to be known by the students.

In addition, the guideline is an important tool for anyone who seeks to study these courses independently. Finally, this guideline is an important asset to the Department to give a direction to instructors what needs to be covered while he/she is teaching the course.

PART I

Formal Language Theory

Summary

Formal language is theoretical definition of languages and tools that define them. Formal language is the basis to study the behavior of languages, machines that define some languages. Tools such as regular expressions, finite state automata, finite state transducer, context free grammars, pushdown automata, Turing machine and others are studied well using these concepts. Natural languages processing is attempted as formal language theory nowadays get more advancement as natural languages are fairly modeled using the formal languages as a modeling tool.

This part of the guideline deals more about the introductory concepts of formal language; finite state automata (deterministic and non-deterministic); regular expressions; context free grammar; pushdown automata language; and languages defined by these concepts.

Unit one

Fundamentals of Formal languages

Summary

Formal language is a set of strings where the strings are formed from sequences of alphabets. Most operations defined in formal language theory are operations of a set. Set theory is the building block of formal language theory. Hence, this unit is intended to provide a guideline for the students on this introductory concept of formal language theory.

General Objective

The general objective of this chapter is to revise the basic concepts for the foundation of formal language theory such as set theory, graph theory, formal proof methods and searching and traversing graphs.

Specific Objectives

At the end of this unit, students should be able to:

- Define a set
- Identify different notations to represent set
- Distinguish complete listing method, partial listing method, and set builder method.
- Describe the power set of a set
- Distinguish finite versus infinite set
- Describe the union, intersection, difference and complement operations
- Describe the subset, equivalence, proper subset of a set
- Describe the commutative, associative, distributive properties of a set
- Describe the De Morgan's law
- Describe the formal proof methods
- Describe proof by deduction
- Use proof by deduction
- Describe proof by induction
- Use proof by induction
- Describe proof by contradiction
- Use proof by contradiction
- Describe the terms alphabets, string, language in formal language analysis

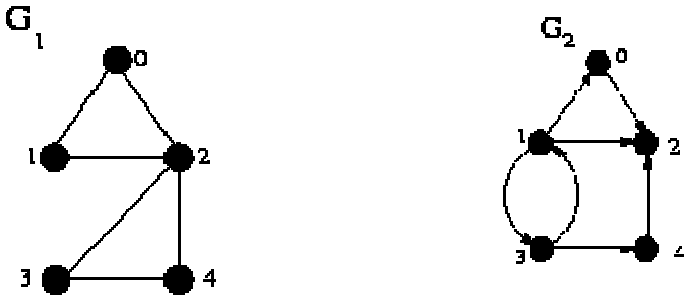
- Identify empty string
- Describe union operation in a string
- Describe concatenation operation in a string
- Describe exponentiation operation in a string
- Describe Kleene closure operator of a set of strings/symbols
- Describe basics of graphs
- Distinguish order of a graph and size of a graph
- Distinguish between cycle graph, cycle in a graph, loop in a graph
- Distinguish between complete graph and connected graph
- Distinguish di-graph from a graph
- Identify degree of a graph
- Identify in-degree, out-degree, degree of a di-graph
- Distinguish a walk, path, cycle in a graph
- Describe disconnected components
- Describe graph representation techniques (Adjacency matrix and Adjacency list)
- Describe graph traversal methods (depth first and breadth first approach)

Self-test Questions without answer key

Attempt all of these questions and make sure that you answer them correctly.

1. Give an example of a set that can be represented using complete listing method
2. Give an example of a set that can be represented using partial listing method
3. Give an example of a set that can be represented using set builder method
4. An infinite set can be represented using complete listing method (TRUE or FALSE)
5. $A \subseteq B$, if and only if all elements of A are also elements of B (TRUE or FALSE)
6. $A \subseteq B$, if and only if all elements of A are also elements of B (TRUE or FALSE)
7. Two sets A and B are said to be equivalent if and only if all elements of A are in B (TRUE or FALSE)
8. Given two set A and B each having N and M elements each, what are the maximum and minimum number of elements in $A \cup B$.

9. Given two set A and B each having N and M elements each, what are the maximum and minimum number of elements in $A \cap B$.
10. What will be $A \cup B$ and $A \cap B$ given A = {1,2,3,8, 11} and B={3, 2, 1, 9, 8}
11. What will be A - B given A = {1,2,3,8, 11} and B={3, 2, 1, 9, 8}
12. What will be the complements of A given A = {1,2,3,8, 11} and U={1,2,3, …, 15}
13. Proof by deduction that the roots of the equation $x^2 - 5x + 6 = 0$ are +2 and +3
14. Proof by deduction that the number of vertices with odd number of degree in a graph are even
15. Proof by induction that the sum of the integers from 1 to N equals (N+1)*N/2
16. The set of ASCII characters of a computer defines an alphabet in formal language theory (TRUE or FALSE)
17. Any string in a natural language are defined from the Latin alphabets{a, b, c, …z, A, B, C, … Z)
18. Define at least three different formal languages from the alphabet {a, b, c, d, e}
19. Differentiate {}, {ε} and ε from formal language perspective
20. Given the two graphs G1 and G2 shown below, answer the following questions from A-Z



a. Identify the degree of all the vertices of G1 and G2
b. Identify the in-degree and out-degree of G2
c. Identify the degree of the graph G1 and G2
d. Distinguish order of a graph and size of a graph
e. Find cycles from G1 and G2
f. Find a walk from G1 which is not a path
g. Is there a loop in G1 and G2

h. How many connected component G1 has
i. Represent G1 and G2 using Adjacency matrix
j. Represent G1 and G2 using Adjacency list

Questions with answer key

1. Given a set A = {C++, C, C#}, list all the power set of A (P(A)). How many elements does P(A) has?
2. According to the De Morgan's law, given A as the set of all even integers, B as the set of integers which are multiple of three and U as the set of all integers answer the following questions
    a. Describe what the complement of $A \cup B$ be verbally as well as using partial listing method
    b. Describe what the complement of $A \cap B$ be verbally as well as using partial listing method
3. Proof by induction that the sum of the degrees of a graph is twice the number of edges

Answer key

1. The number of elements P(A) will have is exactly 2^n where n is the number of elements of A. Hence A has 8 elements in its power set. These are
   {∅,{C++},{C},{C#}, {C++, C}, {C++,C#}, {C,C#}, {C++, C, C#}}
2.
    a. $A \cup B$ is the set of all even integers and all odd integers which are multiple of 3. Hence, the complement of $A \cup B$ becomes a set of non-even integer which are not also multiple of three. Therefore, $(A \cup B)' \equiv$ The set of odd integers which are not multiple of three
    b. $A \cap B$ is the set of integers which are both even and multiple of three. That means, the complement of $A \cap B$ becomes a set of all non-even integer or integers which are not multiple of three. Hence, $(A \cap B)' \equiv$ The set of odd integers or integers which are not multiple of three

3. Proof by induction that the sum of the degrees of a graph is twice the number of edges

   Base phase: assume there is only one edge in the graph. A single edge connects two vertices only say vertices A and B. Hence, the degree of all vertices becomes 0 other than A abd B which have a degree of 1 each. Hence the sum of all the degrees becomes 1 + 1 + 0 + 0 + … + 0 = 2. Hence it is true for N =1.

   Induction phase: Assume it is true for any N such that $N \leq m$, we want to show that it is also true for N= m+1 (i.e the sum of all the degrees of the vertices becomes 2(m+1) = 2m + 2)?

   Let's assume we have a graph G of m+1 edge where one of the edges is from A to B. If we ignore this edge, we will have a Graph G′ with m edge. As G′ has m edges, then the sum of the degrees of all the vertices become 2m (induction assumption above). If we add back the removed edge that links A with B, G′ become modified into G and degree of A and B will increase by one each. Hence, the total sum of degrees of all vertices of G becomes the total sum of degrees of G′ plus 2 (i.e. 2m + 2 = 2(m+1).

Unit Two

Finite State Automata and Regular Languages

Summary

One way to define a language is to construct an automaton: a kind of abstract computer that takes a string as input and produces a yes-or-no answer. The language it defines is the set of all strings for which it says yes.

The simplest kind of automaton is the finite state automaton. More complicated automata have some kind of unbounded memory to work with or use another structure such as stack; in effect, they will be able to grow to whatever size necessary to handle the input string they are given or the stack will be used to memorize important past event happened. PDA and TM have got stack and tape as memory which are infinite but the states in both cases are finite. But this chapter focuses on introducing students about finite state automata (both deterministic and nondeterministic) and the concept of regular languages. A finite automaton has a finite memory that is fixed in advance (finite states) and doesn't use stack memory. Whether the input string is long or short, complex or simple, the finite automaton must reach its decision using the same fixed and finite memory.

General Objective

The purpose of this unit is to introduce the concepts of deterministic finite state automata, regular languages, non-deterministic Finite State Automata (FSA) and their relationships.

Specific Objectives

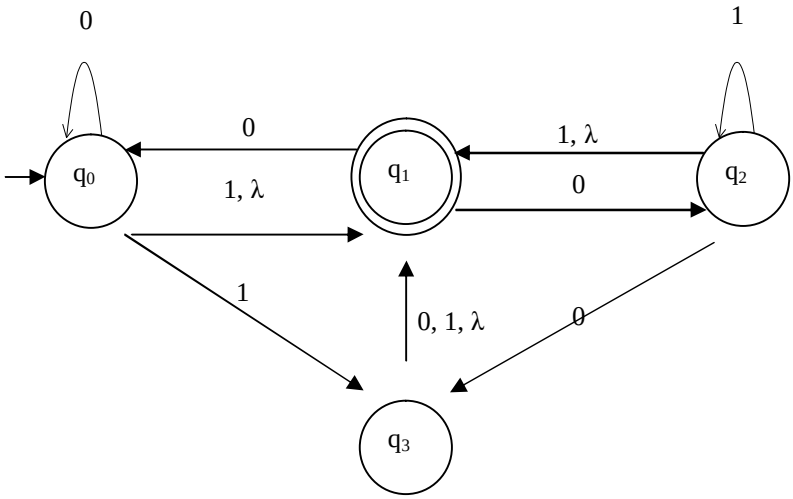
At the end of this unit, students should be able to:

- Identify the concept of states and its interpretation
- Describe what a label refers to in a transition from one state to another state
- Describe FSA in terms of set of states and transitions
- Distinguish start state, final state, and other states
- Describe what do we mean by a string is accepted by a FSA
- Explain a language accepted by the FSA automata
- Identify Deterministic Finite state Automata (DFSA)
- Illustrate DFSA as a five-tuple machine

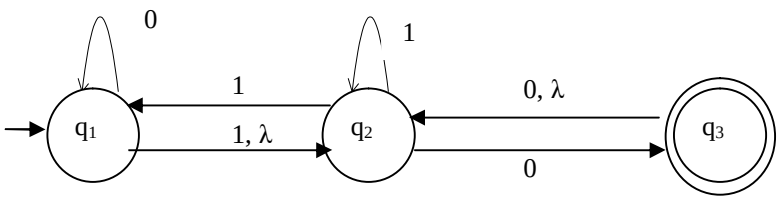
- Describe what transition function be for DFSA
- Describe extended transition function of DFSA
- Provide the definition for the extended transition function of DFSA
- Explain what do we mean by a DFSA machine M accepts a string x
- Define a language defined by a given DFSA machine M
- Define Regular Language
- Identify the closure properties of regular languages
- Describe and prove the closure properties
- Prove that intersection of two regular language is also regular language
- Prove that union of two regular language is also regular language
- Prove that concatenation of two regular language is also regular language
- Prove that Kleene closure of a regular language is also regular language
- Identify Non-deterministic Finite state Automata (NFSA)
- Distinguish NFSA from DFSA
- Illustrate spontaneous transition (**ε-Transition)** in NFSA
- Illustrate NFSA as a five-tuple machine
- Describe how to construct union/concatenation of two NFSA to form complex NFSA
- Describe what transition function be for NFSA
- Describe an instantaneous description of an NFSA
- Describe extended transition function of NFSA
- Provide the definition for the extended transition function of NFSA
- Explain what do we mean by an NFSA machine M accepts a string x
- Define a language defined by a given NFSA machine M
- Explain how to convert DFSA machine into NFSA
- Explain how to convert NFSA machine into DFSA
- Identify the pros and cons of DFSA over NFSA
- Identify the pros and cons of NFSA over DFSA

Self-test Questions

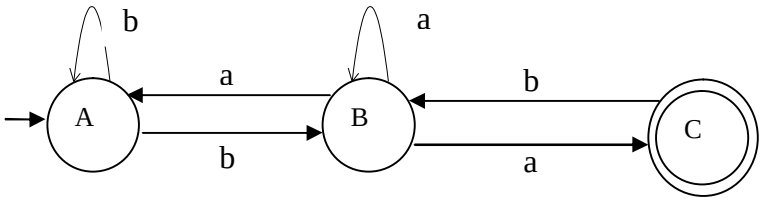
1. Define the language defined by the DFSA shown above
2. Find the DFSA that accept a string **w** of the alphabet **{a,b}** where $\mathbf{N_a \bmod 3 > 1.}$ $\mathbf{N_a}$ is the number of the symbol **a** in **w**.
3. Answer the following questions based on the following FSA.



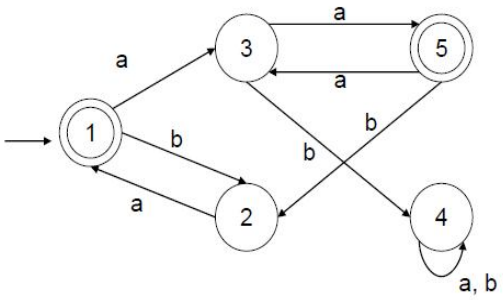
   A. What are Q, ∑ and F?
   B. Is the string 0000111 accepted by the FSA? Justify your answer.
   C. Convert the FSA to an NFSA without **ε-transition** and then check the acceptance of the string 0101.
   D. Convert the NFSA without **ε-transition** moves you obtained above to the corresponding DFSA and then check the acceptance of the string 1100.
   E. Define the language determined by the machine
4. Construct a DFA or NFA that generates a string over the alphabet {a,b} which has m a's followed by n b's for all n and m greater than or equals to 0
5. Discuss the similarities and differences between DFSA and NFSA.
6. Find an equivalent NFSA without λ corresponding to the NFSA shown in the figure below.



7. Find an equivalent DFSA for the following NFSA.



8. Define the language determined by the NFSA machine above
9. Find a regular grammar that generates a language that is accepted by the automaton given in question 11
10. Construct a minimal DFSA for the following DFSA.



11. Given an FSA that accepts binary representation of an integer divisible by three.
12. Given an FSA that accepts binary representation of an integer that ends with 01.
13. Define extended transitions for a DFA machine
14. Describe acceptance a string w by DFS using extended transitions concept
15. Define extended transitions for a NFA machine
16. Describe acceptance a string w by NFS using extended transitions concept
17. Prove that any DFA machine can be converted into NFA machine
18. Prove that any NFA machine can be converted into DFA machine

Questions with answer key

1. Find a DFSA that accept a string **w** of the alphabet **{a}** where $|w| \bmod 3 \neq |w| \bmod 2$.
2. Show that the language $L = \{a^n : n \geq 3\}$ is regular.

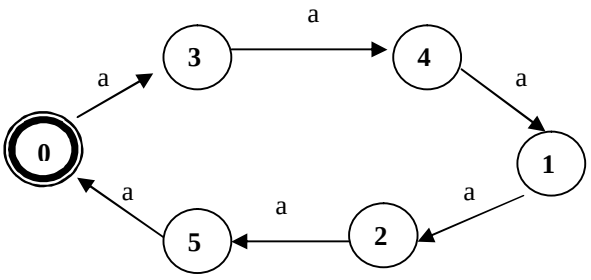
3. Design a DFA that accept only a number (in base 10) which is multiple of 3.

Answer key

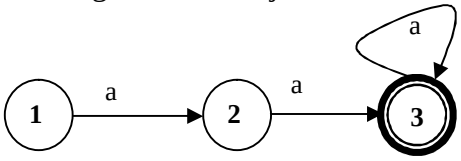
1. To answer this question, we may start by defining a state as an indicator that indicates the remainder for the modulo for the current length when divided by two and three. Hence, I define states as follows:
    a. ***State 0 as a state when length modulo 2 = 0 and length modulo 3 = 0***
    b. State 1 as a state when length modulo 2 = 1 and length modulo 3 = 0
    c. State 2 as a state when length modulo 2 = 0 and length modulo 3 = 1
    d. ***State 3 as a state when length modulo 2 = 1 and length modulo 3 = 1***
    e. State 4 as a state when length modulo 2 = 0 and length modulo 3 = 2
    f. State 5 as a state when length modulo 2 = 1 and length modulo 3 = 2

Hence, except state 0 and 3 are accepting states as per the description.

The following DFSA demonstrate the above argument



2. In order to show the language $L = \{a^n: n \geq 3\}$ is regular, we need to construct a DFA that will accept all strings of L and reject others. The following DFA fulfill the requirement

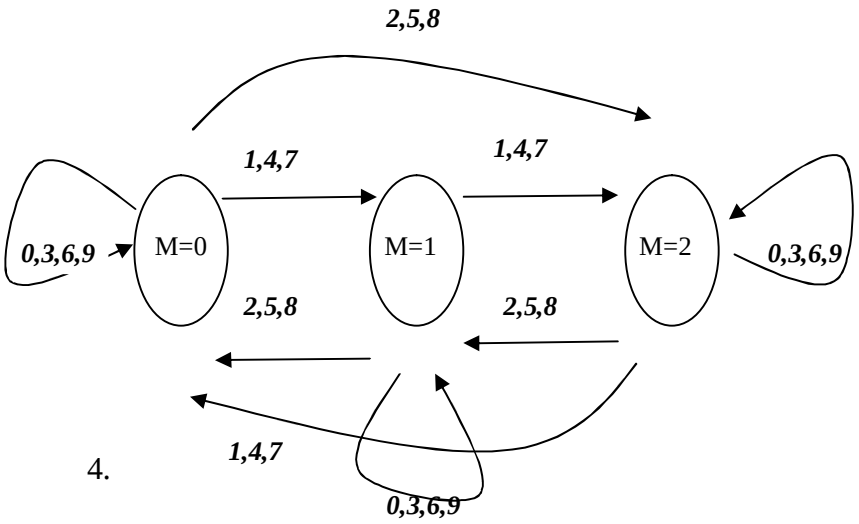


3. Σ={0,1,2,3,4,5,6,7,8,9}. A digit 0, 3, 6 and 9 are all multiple of 3 hence their modulo to 3 is 0. When we compute modulo of the digit 1, 4, 7, 10 to 3, it will be 1 and the remaining digits (2, 5, and 8) has module of 2. If X mod 3 is M for any integer X then Y mod 3 becomes N where Y is X concatenated with some digit d. This relations is shown in the table below for a given X

M=X mod 3	Possible d value	N = Y mod 3
0	0,3,6,9	0
0	1,4,7	1
0	2,5,8	2
1	0,3,6,9	1
1	1,4,7	2
1	2,5,8	0
2	0,3,6,9	2
2	1,4,7	0
2	2,5,8	1

Hence the DFA becomes:

4.



Unit Three

Regular Expressions and Regular Languages

Summary

In the previous unit, study guide were prepared on finite state automata machine that demonstrate both deterministic and non-deterministic finite state automata. In this unit, one of the most powerful mechanisms to define regular language, which is a regular expression, will be presented. Regular expressions are getting popularity in various programming languages and operating system shell commands to process natural languages. Regular expressions will have the same expression power of DFSA and NFSA and hence can be used to express a language called regular language.

General Objective

The general objective of this unit is to guide students on topics such as definition and concepts of regular expression formation, language defined by regular expression, conversion of NFSA and DFSA into regular expression and vise versa and its applications.

Specific Objectives

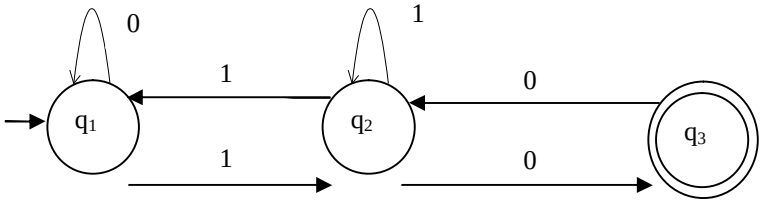
At the end of this unit, students should be able to:

- Define a regular expression
- Define primitive regular expressions
- Describe the operators used in regular expressions which include union (+), concatenation (.), and Kleene closure (*)
- Define complex regular expressions from the primitive ones using the operators
- Define the language that a regular expression denotes
- Show that the language that a regular expression denotes is a regular language by constructing NFSA that accept the languages denoted by primitive regular expressions and complex regular expressions
- Find a regular expression that denotes the language accepted by NFSA
- Know the existence of families of languages that are not regular
- Define Pumping Lemma for regular languages
- Show that some languages are not regular languages by using Pumping Lemma

- Distinguish between the formal regular expression definition and the various regular expression defined by Perl, Python, Java, Unix shell command grep/egrep, etc

Self-test Questions

1. Find an NFSA that accepts L(r) where r = aa*(a + b)
2. Find a regular expression corresponding to the NFSA shown in the figure below.

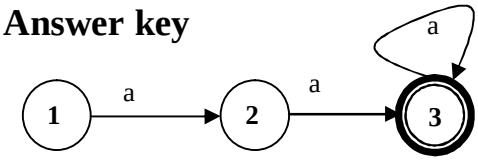


3. State pumping lemma for regular languages
4. Using pumping lemma show that the language L = $\{a^n b 2^n \mid n > 0\}$ is not regular.
5. Write regular expression for the languages all strings ending in 01 given Σ= {0, 1}.
6. Write regular expression for the languages all strings not ending in 01 given Σ= {0, 1}.
7. Find regular expressions for the languages on L = $\{w: n_a(w) \bmod 3 = 0\}$given Σ={a, b}.
   Find regular expressions for the languages on L = $\{w: n_a(w) \bmod 5 > 0\}$given Σ={a, b}.
8. State the precedence rules for the regular expression operators
9. Construct a regular expression r for C++ identifiers (i.e $x \in L(r) \Leftrightarrow$ x is a C++ identifier)
10. Convert the regular expressions (0+1)01 into NFA with ε transitions
11. Convert the regular expressions 00(0+1)* into NFA with ε transitions
12. Prove the following theorems
    a. The union of two regular language is regular
    b. The intersection of two regular language is regular
    c. The complement of a regular language is regular
    d. The difference of two regular language is regular
    e. The reversal of a regular language is regular
    f. The closure (star) of a regular language is regular
    g. The concatenation of two regular language is regular

Questions with answer key

1. Convert the regular expressions 01* into NFA with ε transitions
2. Find regular expressions for the languages on L = {w: |w| mod 3 = 0} given Σ={a, b}.
3. Write regular expression for the languages all strings containing even number of zeros given Σ= {0, 1}.

Answer key



Unit Four

Context Free Grammar

Summary

So far we have seen finite state automata and regular expression as a mechanism to define regular language and their properties. In this chapter we will look into more complex means of language construction tool called context free grammar. It is a means to define grammar of a language. Context free grammar defines set of strings using set of production rules. It is called context free because each production are independent from each other.

General Objective

The general objective of the chapter is to deal with CFG and the language defined by the grammar. Moreover, links will be established with the concepts that we have discussed so far.

Specific Objectives

At the end of this unit, students should be able to:

- Define what a Context Free Grammar (CFG) is
- Describe role of Context Free Grammar for parsing
- Identify whether a given grammar is a CFG or not
- Define what a Context Free Language (CFL) is
- Define what a derivation or a parse tree is
- Explain how parse tree is generated from a CFG and token sequence
- Identify derivation in a CFG
- Define leftmost derivation
- Define rightmost derivation
- Define extended derivation (n-step derivation)
- Identify single step grammar
- Identify left linear grammar
- Identify right linear grammar
- Identify ambiguous grammar
- Describe a language defined by a grammar

- Define a string from L(G) using extended derivation
- Identify unambiguous grammar
- Show how CFGs can be used in parsing arithmetic expression
- Explain how operator derived ambiguity can be resolved
- Identify what left recursion (immediate and non-immediate left recursion)
- Explain how to remove left recursion
- Explain the purpose of left factoring
- Explain how to do left factoring
- Define λ-productions
- Define nullable nonterminals
- Explain how to remove λ-productions
- Define unit productions
- Explain how to remove unit productions
- Define useless/irrelevant productions
- Explain how to remove irrelevant productions
- Define normal forms
- Define Chomsky Normal Form (CNF)
- Show how a λ-free CFG can be converted into an equivalent CFG in CNF
- Define Greibach Normal Form (GNF)
- Show how a λ-free CFG can be converted into an equivalent CFG in GNF
- Define Pumping Lemma for CFLs
- Show that some languages are not CFLs by using Pumping Lemma for CFLs
- Describe membership algorithm for CFLs
- Show CFLs are closed under union, concatenation and Kleene closure, not intersection
- Show that every regular languages have a CFG
- Show that all CFG do not define regular language
- Show that every single step grammar define regular language
- Show that every right linear grammar define regular language
- Show that every left linear grammar define regular language

Self-test Questions

1. Describe the following grammars using the different grammar characteristics (context free or not, the language it defines, linearity, ambiguity, single step or not, etc)

   B. $S \rightarrow aSb \mid \lambda$

   C:
   $S \rightarrow aB$
   $B \rightarrow bA|b$
   $A \rightarrow a$

   B:
   $S \rightarrow aB|A$
   $aA \rightarrow aA|a|CBA$
   $B \rightarrow \lambda$

   D:
   $A \rightarrow aB$
   $B \rightarrow cA|C$
   $C \rightarrow cC|c$

2. Given a grammar G = (N, T, P, S) with productions:

   $S \rightarrow AB$

   $A \rightarrow aA|a$

   $B \rightarrow bB|b$

   And a string x = aaabbb

   A) find a left most and right most derivations for x.

   B) draw the parse tree for x.

   C) is the grammar ambigious?

3. Given a grammar G = (N, T, P, S) with productions:

   $S \rightarrow SbS|ScS|a$

   and a string x = abaca

   A) find a left most and right most derivations for x.

   B) draw the parse tree for x.

   C) is the grammar ambigious?

4. Consider the following grammar:

   $E \rightarrow T \mid E + T \mid E - T$

   $T \rightarrow F \mid T * F \mid T / F$

   $F \rightarrow a \mid b \mid c \mid (E)$

   Draw parse trees for the following strings

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

5. Given CFG grammar G with productions

   $S \rightarrow aS \mid AB$,

   $A \rightarrow \lambda$,

   $B \rightarrow \lambda$,

   $D \rightarrow b$,

   construct a grammar G' without null productions such that $L(G') = L(G) - \{\lambda\}$.

6. Remove all unit productions from the following CFG.

   $S \rightarrow Aa \mid B$

   $B \rightarrow A \mid bb$

   $A \rightarrow a \mid bc \mid B$

7. Eliminate useless productions from

   $S \rightarrow a \mid aA \mid B \mid C$

   $A \rightarrow aB \mid \lambda$

   $B \rightarrow Aa$

   $C \rightarrow cCD$

   $D \rightarrow ddd$

8. Convert the grammar with productions

   $S \rightarrow ABa \mid ABS$

   $A \rightarrow aab \mid abc$

   $B \rightarrow Ac$

   to Chomsky normal form.

9. Convert the grammar with productions

   $S \rightarrow abSb \mid aa$

   to Greibach normal form.

10. Define a context free grammar for declaration statement for a C++ programming language

Unit Five

Pushdown Automata

Summary

Push down automata is a type of machine that defines context free grammar language. Unlike finite state automata, pushdown automata is an extension to the NDFSA with addition stack that one can push, pop and read from the stack.

General Objective

The general objective the chapter is to introduce pushdown automata and its functionality.

Specific Objectives

At the end of this unit, students should be able to:

- Define Pushdown Automata (PDA)
- Illustrate PDA as a 7-tuple automata
- Describe what happens when a PDA makes a move
- Show the association between PDA and CFL
- Show the differences between FSA and PDA
- Show how PDA accepts language families that cannot be accepted by FSA
- Describe the types of PDA: deterministic PDA (DPDA) and nondeterministic PDA (NPDA)
- Describe an NPDA using transition diagram
- Describe an Instantaneous Description (ID) of a NPDA
- Describe a move of a NPDA using IDs
- Define how strings are accepted by a NPDA
- Show the equivalence between CFGs and NPDAs
- Define a DPDA
- Show that DPDA is not equivalent to NPDA
- Describe deterministic CFL

Self-test Questions

1. Construct an DPDA that accepts L = $\{a^n b^n: n \geq 0\}$
2. Construct an NPDA that accepts L = $\{a^{2n} b^n: n \geq 0\}$
3. Construct an NPDA that accepts L = $\{a^n b^{2n}: n \geq 0\}$
4. Using instantaneous description show that whether the string **aaaabb** is accepted in question 2
5. Construct an NPDA that accepts the language generated by the following CFG

   E→T | E + T | E – T

   T→F | T * F | T / F

   F→a | b | c | (E)
6. Construct an NPDA that accepts L = $\{ww^R: w \in \{a, b\}^+\}$
7. Construct an NPDA for the language L = $\{w \in \{a, b\}^+: n_a(w) = n_b(w)\}$

Part II

Compiler Design

Summary

Programming machines is one of the most important components for advancement in information communication and technology. Machines solve problem as they are capable of doing several logical computations within a fraction of seconds. Machines can execute instruction when it is represented in the form of machine language which is by far different from human language. The language is defined as a set of instructions that would be executed by the machine so as to solve problems. This fascinates computer scientists to work hard and make machine programming easier and efficient and effective in the day to day human activities.

Programmer could write a non-ambiguous program (set of instructions) in human understandable form, which when executed properly on a given input would produce an output that would be desired solution to a problem. However, such languages couldn't be directly taken by the machine as it is not understandable. Such language should be translated into the machine language with the help of translators. Translator comes in different forms.

This part focus on design and implementation of compilers one of the most common translators from source code into machine language for compiled programming language. The design and implementation of programming language compiler will be introduced in this part. Theoretical aspects of language design and translation are discussed and practically demonstrated by developing a working compiler. Concepts such as lexical analysis, symbol tables, parsing, syntax directed translation, type checking, code generation, and code optimization will be introduced.

Unit One

Fundamentals of Complier Design

Summary

Compiler is software that translator source language written in one language into the desired target language. Compilers use pre-specified grammar from a set of tokens defined using some kind of regular expressions. Compilation passes through a number of phases. The chapter will introduce fundamental concepts in design and implementation of compilers and the different phases so that students will be ready to grasp detailed concepts briefly introduced in this chapter during subsequent chapter discussion.

General Objective

The purpose of this unit is to introduce fundamental concepts of compiler to students such as definition of translators, compiler, phases of compiler design, preprocessor, linker, loader, etc

Specific Objectives

At the end of this unit, students should be able to:

- Describe what a translator is
- Describe what a compiler is
- Distinguish compiler from other translators
- Explain what the output of a compiler be for a given set of characters
- Explain the role of pre-processor, linker and loader while solving problem or during translation
- Identify the two major phases of compiler design
- Distinguish the analysis phase from the synthesis phase
- Describe the steps how a source code written using a compiled language be converted into executable machine code
- Identify and describe the three sub-phases of the analysis phase
- Explain why linear analysis is also called lexical analysis or scanner or tokenizer
- Explain why hierarchical analysis is also called syntax analysis or parser
- Identify and describe the three sub-phases of the synthesis phase

- Describe what a symbol table is in compiler design
- Describe an error handler module in compiler design
- Arrange the entire phases in the process of compiler design showing their I/O relationship
- Demonstrate the role of the above sub-phases using example

Self-test Questions

1. _____ is a program that can read a program written in high-level language and converts it into an equivalent machine language.

2. _____ is a program that translates and then executes a high-level instruction before translating the next instruction.

3. _____ is a program whose task is collecting modules of a program stored in separate files and expanding macros into source language statements.

4. _____ processes assembly language program and produces relocatable machine code.

5. _____ is a program that links together relocatable machine codes with other relocatable object and library files into the code actually runs on the machine.

6. _____ puts all executable programs into memory for execution.

7. List the various phases of a compiler

8. List the phases of a compiler that are termed as front-end of a compiler.

9. List the phases of a compiler that are termed as back-end of a compiler.

10. What are the input to and the output of lexical analysis phase?

11. What are the input to and the output of syntax analysis phase?

12. What are the input to and the output of semantic analysis phase?

13. What are the input to and the output of intermediate code generation phase?

14. What are the input to and the output of code generation phase?

15. What is the purpose of symbol table?

16. Give an example of lexical analyzer generator.

17. Give an example of parser generator.

Unit Two

Lexical Analysis

Summary

Every language has lexical element such as words, numbers, acronyms, symbols etc. In order to convey message these lexical elements should be arranged in their appropriate order. Programming language must define its lexical element before defining its set of instructions. Hence this chapter focuses on how to define lexical element and analyze them for use in subsequent phases. Moreover, before converting source code written in a compiled language, the source code lexical elements (the smallest meaning bearing units of the program) should be analyzed. This unit will provide how to do such analysis (lexical analysis)

General Objective

The purpose of this unit is to algorithmically describe how to identify the optimal sequence of tokens that exist in the source code of the program. Moreover, each of the lexical elements also called tokens will be analyzed for their lexical category, attributes (or resource requirements) for use for later analysis/synthesis phases.

Specific Objectives

At the end of this unit, students should be able to:

- Identify what the role of lexical analyzer is
- Define the terms: symbol, lexeme, token, and pattern
- Provide the set of valid symbols for one of the programming language
- Provide examples of lexemes, tokens and patterns based on the preferred language
- Describe the interaction between lexical analyzer and the parser
- Describe how lexical error be produced by the lexical analyzer
- Identify the role of regular expression for lexical analyzer
- Identify token and token property (attributes)
- Describe why lexical analyzer return token and token property after detecting a lexeme
- Describe when lexical analyzer return NULL as an attribute for a token and why
- Describe when lexical analyzer return the lexeme as an attribute for a token and why

- Describe when lexical analyzer return address of symbol table as an attribute for a token and why
- Distinguish between regular expression and regular definition
- Describe about the role of the lexical analyzer generating tools such as (Lex or Flex)
- Describe how to use Flex or Lex
- Describe the three sections of Lex or Flex source file
- Demonstrate Lex or Flex using examples

Self-test Questions

1. _____ reads the input characters of the source program, groups them into lexemes, and produces as an output a sequence of tokens for each lexeme in the source program.
2. _____ is a description of the form that lexemes of a token may take.
3. _____ is a sequence of characters in the source program that matches the pattern for a token.
4. List three additional tasks performed by a scanner other than identifying tokens.
5. List some of the tokens that exist in a particular programming language.
6. What is/are the purpose/s of attributes of a token?
7. List some of the lexical errors that can occur during scanning.
8. List some of the actions that a lexical analyzer takes in recovering from lexical errors.
9. Define a regular expression for the following tokens in C++ programming
    a. Identifier
    b. Keywords
    c. Operators
    d. Symbols
    e. Integer numbers
    f. Floating point numbers
    g. Strings
    h. Character
10. For the above tokens, what attribute each token to have if the list shows the token identity in group
11. Discuss how input buffering works with and without the use of sentinels.

12. Draw a transition diagram that recognizes the operators: +, ++, +=, --, -, -= , = and ==.
13. Distinguish regular definition and regular expression
14. Write a regular definition for integer constants in C++.
15. Give notations used in extensions of regular expressions in specifying one or more instances, zero or one instance, and character classes.
16. Discuss how a lexical analyzer system is built

Unit Three

Syntax Analyzer

Summary

Grammar is the building block to carry message through sequence of tokens. Unless the tokens are arranged in precise order, it would be difficult to decide about the instruction it carries to be performed. Hence, syntax analyzer analyzes the output of the lexical analyzer (sequence of tokens and their associated properties) so as to understand the grammar of the code. In this unit, we will focus on how to analyze the validity of the grammar of the token sequence through syntax analyzer/parser.

General Objective

The purpose of this unit is to parse token sequence using context free grammar defined for the language. Concepts such as CFG, derivation, ambiguity and ambiguity resolution, types of parsing: top-down parsing, bottom up parsing will be discussed

Specific Objectives

At the end of this unit, students should be able to:

- Define parse trees
- Describe role of Context Free Grammar for parsing
- Identify derivation in a CFG
- Distinguish left most derivation and right most derivation
- Explain how parse tree is generated from a CFG and token sequence
- Identify ambiguity in syntax analysis
- Explain how operator derived ambiguity can be resolved
- Identify what left recursion (immediate and non-immediate left recursion)
- Explain how to remove left recursion
- Explain the purpose of left factoring and how to do left factoring
- Describe top-down and bottom up parsing
- Explain what LL parsing is
- Explain the relationship between LL and top down parsing

- Explain what LR parsing is
- Explain the relationship between LR and bottom up parsing
- Define the two functions used in both top-down and bottom-up parsing: FIRST and FOLLOW
- Describe the three types of top-down parsing algorithms (recursive descent parser, recursive predictive parsing, non-recursive predictive parsing)
- Illustrate algorithmically recursive Descent parser, recursive predictive parsing, non-recursive predictive parsing with examples
- Distinguish the strength and weakness of each of the above top-down parsers
- Describe why bottom-up parser is also called shift/reduce parser
- Describe the three types of bottom-up parsing algorithms (Simple LR parser, the canonical LR parsing, look ahead LR parsing)
- Illustrate algorithmically Simple LR parser, the canonical LR parsing, look ahead LR parsing with examples
- Distinguish the strength and weakness of each of the above bottom-up parsers

Self-test Questions

1. What are the input to and the output of a parser?
2. List the two commonly used parsing methods using in compilers.
3. List the goals that an error handler should possess.
4. List four types of error recovery modes.
5. How does a compiler that implements panic mode error recovery method recovery from an error? Discuss
6. FIRST(α), a string of grammars symbols α, is a set of terminals that begins the strings derived from α. (TRUE / FALSE)
7. FOLLOW(A), for a nonterminal A, is the set of terminals α that can appear immediately to the right of A in some sentential form. (TRUE / FALSE)

8. Compute FIRST and FOLLOW of non-terminals in the following grammar:
   E→TE'

E'→+TE' | λ

T→FT'

T'→*FT' | λ

F→(E) | id

9. List two types of top-down parsing techniques.
10. Construct a non recursive predictive parsing table for the grammar in question 8 and then parse the input id + id $.
11. Parse input **efef** using recursive descent parsing technique based the following grammar.
    A → fAeA | eAfA| λ
12. What types of grammars are LL(1) grammars? What does LL(1) stand for?
13. List the procedures employed to check whether a given grammar is an LL(1) grammar or not.
14. What are handle and handle pruning in bottom-up parsing?
15. How does a shift-reduce parser works?
16. List the conflicts that occur in shift-reduce parsing and explain briefly about the conflicts.
17. List the four possible actions that a shift-reduce parser makes.
18. What does LR(K) stand for?
19. Discuss the LR parsing algorithm
20. Define an LR(0) item.
21. What is closure(I), for an LR(0) item I?
22. Define goto(I, X), for an LR(0) item I and a grammar symbol X.
23. Describe the sets of items construction algorithm in SLR.
24. Describe the algorithm for constructing an SLR(1) parsing table.
25. Define an LR(1) item.
26. What is closure(I), for an LR(1) item I?
27. Define goto(I, X), for an LR(1) item I and a grammar symbol X.
28. Describe the sets of items construction algorithm in LR(1).
29. Describe the algorithm for constructing an LR(1) parsing table.

Unit Four

Syntax Directed Translation

Summary

Compilers are responsible to do translation. Syntax directed translation is one of the approaches that do translation assisted by the proposed grammar. Source code will be scanned for identification of token sequence and passed to the parser. Parser succeeds if there is one or more derivations that would accepted the token sequence. Syntax directed translation refers to translating the source code while appropriate productions are selected.

General Objective

The objective of the chapter is to discuss how to approach translation of the source code while parsing. This would require understanding of the semantic of the source code while performing reduction of derivation so that translation can be made at the same time while doing reduction.

Specific Objectives

At the end of this unit, students should be able to:

- Define syntax directed translation
- Define syntax directed definition
- Describe the applications of syntax directed translation using examples
- Define attributes of grammar symbols
- Define the two types of attributes: synthesized and inherited
- Discuss the differences between synthesized and inherited attributes
- Define annotated / decorated parse tree
- Define dependency graph
- Define evaluation order and the various methods for evaluating semantic rules
- Define S-attributed grammars
- Discuss bottom up evaluation of S-attributed grammar
- Discuss top-down evaluation of S-attributed grammar
- Define L-attributed grammars

- How to define a translation scheme with inherited attribute
- How to eliminate left recursion from translation scheme

Self-test Questions

1. _____ is a generalization of the CFG in which each grammar symbol has an associated set of attributes (synthesized and inherited) and rules.
2. List the two types of attributes in syntax directed definition.
3. What is an annotated parse tree?
4. Given the following syntax directed definition, draw annotated parse tree for 1101.11

Syntax Rules	**Semantic Rules**
$N \rightarrow L_1.L_2$	$N.v = L_1.v + L_2.v / (2^{L2.l})$
$L_1 \rightarrow L_2\ B$	$L_1.v = 2 * L_2.v + B.v$
	$L_1.l = L_2.l + 1$
$L \rightarrow B$	$L.v = B.v$
	$L.l = 1$
$B \rightarrow 0$	$B.v = 0$
$B \rightarrow 1$	$B.v = 1$

5. What types of attributes are v, l, and s in question 4? (synthesized / inherited)
6. Draw a dependency graph for question 4.
7. Write at least two topological sort for question 4.
8. What is an S-attributed grammar?
9. What is an L-attributed grammar?
10. What is the difference between attributes in an L-attributed grammar and inherited attributes?
11. Give an example of S-attributed grammar.
12. Give an example of L-attributed grammar.

Unit Five

Type checking

Summary

Some programming are strongly typed (require all its identifiers to be declared before use and implicitly type casting is prohibited), some are weakly typed (require all its identifiers to be declared before use but implicitly or explicit type casting is possible among compatible types) and others are un-typed (doesn't require identifiers to be declared and they can play any type role at any point in the program). Type checking is required for strongly or weakly typed languages to conform the design policy.

General Objective

The objective of this chapter is to discuss the issue of type checking and detect invalid operation from the perspective of object type so that corrective measure can be made at early stage.

Specific Objectives

At the end of this unit, students should be able to:

- Define what type checking is
- Describe the various types of static checks: type checks, flow-of-control checks, name-related checks, uniqueness checks
- Define type expressions
- Define type constructors
- Show how type constructors can be used in defining type expressions
- Define type systems
- Describe error recovery in type checking
- Show assigning types to the various parts of a program using syntax directed definition
- Describe equivalence of types
- Describe structural equivalence of types
- Describe name equivalence of types
- Describe type conversions

Self-test Questions

1. List some of the activities that a compiler performs during type checking.
2. What does uniqueness check mean?
3. List some of the type expressions in C++.
4. List some of the type constructors in C++.
5. Describe the difference between structural equivalence and name equivalence in type expressions.
6. Given the following syntax directed definition for assigning types for program parts, draw an annotated parse tree for **a mod b** assuming that a and b have been declared integers.

E→literal	{E.type := char}
E→num	{E.type := integer}
E→id	{E .type := lookup (**id**.entry)}
E→E1 mod E2	{E.type := **if** E1.type = integer and E2.type := integer **then** integer **else** type_error}
E→E1[E2]	{E.type := **if** E2.type = integer and E1.type := array(s, t) **then** t **else** type_error}
E→^E1	{E.type := **if** E2.type = pointer(t) **then** t **else** type_error}

Unit Six

Intermediate Code Generation

Summary

Usually compilers don't generate the machine code or object code on the fly. It should pass through phases such as intermediate code generation. Intermediate codes are codes that split statements into a set of smaller instructions that could be directly mapped into the desired object code. This permits to easily generate the final code and to undertake code optimization

General Objective

The main objective of this chapter is to generate an equivalent code as the source code which can be easily mapped into the desired object codes.

Specific Objectives

At the end of this unit, students should be able to:

- Define intermediate code
- Describe the use of generating machine-independent intermediate code
- List various types of intermediate languages: syntax trees, postfix notation, three – address code
- List the various types of three address statements: binary assignment, unary assignment, copy statement, unconditional jump, conditional jump, function call, indexed arguments, and address and pointer assignments
- Use syntax directed translation to generate three-address code statements for: declarations, assignment statements, and addressing array elements
- Define backpatching

Self-test Questions

1. What are the uses of generating an intermediate code?
2. List three intermediate languages used in intermediate code generation.
3. Give the format of three-address code for function call.
4. Give the format of three-address code for conditional jump.

5. Describe the uses of the functions: **newtemp**, **newlabel**, and **gen** in generating three-address code statement using syntax directed translation.
6. Given the following syntax directed definition, generate three-address code statements for the assignment statement: a:=x+y*z.

Production	**Semantic Rules**
$S \rightarrow id := E$	S.code := E.code || gen (id.lexeme, :=, E.place)
$E \rightarrow E_1 + E_2$	E.place := newtemp; E.code := E_1.code || E_2.code || gen (E.place, ':=', E_1.place, '+', E_2.place)
$E \rightarrow E_1 * E_2$	E.place := newtemp; E.code := E_1.code || E_2.code || gen (E.place, ':=', E_1.place, '*', E_2.place)
$E \rightarrow - E_1$	E.place := newtemp; E.code := E_1.code || gen (E.place, ':= uminus ', E_1.place)
$E \rightarrow (E_1)$	E.place := newtemp; E.code := E_1.code
$E \rightarrow id$	E.place := id.lexeme; E.code := '' /* empty code */

Unit Seven

Code generation

Summary

Code generation is the final phase in compiler implementation which generates the object code from the intermediate codes. Code generator may be undertaken before or after code optimization.

General Objective

The objective of this chapter is to discuss how code can be generated from the intermediate code.

Specific Objectives

At the end of this unit, students should be able to:

- Define code generation
- Describe the properties that the generated code should possess
- Describe the issues in code generation including: input, output, memory management, instruction selection, and register allocation
- Describe how the simple code generation algorithm works

Self-Test Questions

1. What are the input to and the output of a code generator?
2. What properties are required of the generated code?
3. Give an example on how selection of instructions affects the efficiency of a generated code.
4. Describe the differences between register allocation and register assignment.
5. Given the following three-address statements for the assignment statement d := (a – b) + (a – c) + (a – c), describe how the simple code generation algorithm works assuming that d is live.

   t := a – b

   u := a – c

   v := t + u

   d := v + u