CSC 432 Principles of Programming Languages II

LANGUAGE SEMANTICS

Semantics is the study of the relationship between words and how we draw meaning from those words.

Semantics involves the deconstruction of words, signals, and sentence structure. It influences our reading comprehension as well as our comprehension of other people's words in everyday conversation. Semantics play a large part in our daily communication, understanding, and language learning without us even realizing it.

Since meaning in language is so complex, there are actually different theories used within semantics, such as formal semantics, lexical semantics, and conceptual semantics etc

- **Formal Semantics** is the study of grammatical meaning in natural languages using formal tools from logic and theoretical computer science. - Formal semantics uses techniques from math, philosophy, and logic to analyze the broader relationship between language and reality, truth and possibility. Has your teacher ever asked you to use an "if… then" question? It breaks apart lines of information to detect the underlying meaning or consequence of events.
- **Lexical Semantics** - Lexical semantics deconstruct words and phrases within a line of text to understand the meaning in terms of context. This can include a study of individual nouns, verbs, adjectives, prefixes, root words, suffixes, or longer phrases or idioms.
- **Conceptual Semantics** - Conceptual semantics deals with the most basic concept and form of a word before our thoughts and feelings added context to it.
  For example, at its most basic we know a cougar to be a large wild cat. But, the word cougar has also come to indicate an older woman who's dating a younger man. This is where context is important.
- **Denotation** A denotational definition of a language consists of three parts: the abstract syntax of the language, a semantic algebra defining a computational model, and valuation functions. The valuation functions map the syntactic constructs of the language to the

semantic algebra. Recursion and iteration are defined using the notion of a limit. The programming language constructs are in the syntactic domain while the mathematical entity is in the semantic domain and the mapping between the various domains is provided by valuation functions. Denotational semantics relies on defining an object in terms of its constituent parts.

- **Connotation**, on the other hand, refers to the associations that are connected to a certain word or the emotional suggestions related to that word. The connotative meanings of a word exist together with the denotative meanings. The connotations for the word snake could include evil or danger. Connotation is created when you mean something else, something that might be initially hidden.
- **Axiomatic semantics** - The axiomatic semantics of a programming language are the assertions about relationships that remain the same each time the program executes. Axiomatic semantics are defined for each control structure and command. The axiomatic semantics of a programming language define a mathematical theory of programs written in the language.

A mathematical theory has three components:

- *Syntactic rules:* These determine the structure of formulas which are the statements of interest;
- *Axioms:* These describe the basic properties of the system;
- *Inference rules:* These are the mechanisms for deducing new theorems from axioms and other theorems

Operational Semantics-An operational definition of a language consists of two parts: an abstract syntax and an interpreter. An interpreter defines how to perform a computation. When the interpreter evaluates a program, it generates a sequence of machine configurations that define the program's operational semantics. The interpreter is an evaluation relation that is defined by rewriting rules. The interpreter may be an abstract machine or recursive functions.

Basic Declarations and Expressions

Elements of a Program

If you are going to construct a building, you need two things: the bricks and a blueprint that tells you how to put them together. In computer programming, you need two things: data (variables) and instructions (code or functions). Variables are the basic building blocks of a program. Instructions tell the computer what to do with the variables.

Comments are used to describe the variables and instructions. They are notes by the author documenting the program so that the program is clear and easy to read. Comments are ignored by the computer.

In most programming languages, we must declare our variables before we can use them. After our variables are defined, we can begin to use them. The basic structure is a function. Functions can be combined to form a program.

Basic Program Structure

The basic elements of a program are the data declarations, functions, and comments. Let's see how these can be organized into a simple C program.

The basic structure of a one-function program is:

```
/*****************************************************
 * ...Heading comments...                            *
 *****************************************************/
...Data declarations...
int main()
{
   ...Executable statements...
   return (0);
}
```

Heading comments tell the programmer about the program, and *data declarations* describe the data that the program is going to use.

Our single function is named main. The name main is special, because it is the first function called. Other functions are called directly or indirectly from main. The function main begins with:

```
int main()
{
```

and ends with:

```
return (0);
}
```

The line return(0); is used to tell the operating system (UNIX or MS-DOS/Windows) that the

program exited normally (Status=0). A nonzero status indicates an error—the bigger the return value, the more severe the error. Typically, a status of 1 is used for the most simple errors, like a missing file or bad command-line syntax.

Basic structure of a C++ program

Structure is a collection of variables of different data types under a single name. It is similar to a class in that, both holds a collection of data of different data types.

For example: You want to store some information about a person: his/her name, citizenship number and salary. You can easily create different variables name, citNo, salary to store these information separately.

However, in the future, you would want to store information about multiple persons. Now, you'd need to create different variables for each information per person: name1, citNo1, salary1, name2, citNo2, salary2

You can easily visualize how big and messy the code would look. Also, since no relation between the variables (information) would exist, it's going to be a daunting task.

A better approach will be to have a collection of all related information under a single name Person, and use it for every person. Now, the code looks much cleaner, readable and efficient as well.

This collection of all related information under a single name Person is a structure.

C++ Basic Syntax

Comments in C++ Program

C++ Comments are a set of statements that are not executed by the C++ compiler. The use of comments makes it easy for humans to understand the source code. Usually comments give you inside or explanation about the variable, method, class or any statement that exists in source code. The comment statements are ignored during the execution of the program.

A '//' (double forward slash) is used to specify a **single line comment**, which extends up to the newline character.

```
 // It prints Hello, World!
cout<<"Hello World!";
// It prints Hello, World!
cout<<"Hello World!";
```

If you want to comment **multiple lines** then you can do it using /* and */, everything in between from /* to */ is ignored by the compiler-

```
/* It prints
Hello World! */
cout<<"Hello World!";
```

C++ Tokens

In C++, a program is consists of various tokens. In C++, tokens are smallest individual units can either be a keyword, an identifier, a constant, a string literal, or a symbol. In C++, we have five types of tokens as following:

- Keywords
- Identifiers
- Constants
- Punctuators
- Operators

C++ Punctuators

In C++, a punctuator is a token used to separate two individual tokens used in a c++ program. It is also called as separators. In C++, we have the brace { and }, parenthesis (and), and brackets [and], comma (,), semicolon (;), asterisk (*), colon (:), number sign #(hash) as punctuators.

C++ Semicolon

The semicolon is a statement terminator that is each of the individual statement must be ended with a semicolon. In C++, it is mandatory to put a semicolon (;) at the end of each of the executable statement. Otherwise, the compiler will raise a syntax error.

Curly Braces In C++

In C++, an opening and closing curly braces is used to group all of the statements in a block.

Block in C++

Block is a set of statement grouped together inside opening and closing braces.

Example:-

```
1 {//start of block
2
3  //block of statement(s)
4
5 }//end of block
```

A **C++ program** is structured in a specific and particular manner. In C++, a program is divided into the following three sections:

1. Standard Libraries Section
2. Main Function Section
3. Function Body Section

For example, let's look at the implementation of the `Hello World` program:

```cpp
#include <iostream>
using namespace std;

int main() {
  cout << "Hello World!" << endl;
  return 0;
}
```

Run

Standard libraries section

```cpp
#include <iostream>
using namespace std;
```

- `#include` is a specific preprocessor command that effectively copies and pastes the entire text of the file, specified between the angle brackets, into the source code.
- The file `<iostream>`, which is a standard file that should come with the C++ compiler, is short for **input-output streams**. This command contains code for displaying and getting an input from the user.
- `namespace` is a prefix that is applied to all the names in a certain set. `iostream` file defines two *names* used in this program - **cout** and **endl**.
- This code is saying: Use the cout and endl tools from the std toolbox.

Main function section

```cpp
int main() {}
```

- The starting point of all C++ programs is the `main` function.
- This function is called by the operating system when your program is executed by the computer.
- `{` signifies the start of a block of code, and `}` signifies the end.

Function body section

```
1   cout << "Hello World" << endl;
2   return 0;
```

- The name `cout` is short for **character output** and displays whatever is between the `<<` brackets.
- Symbols such as `<<` can also behave like functions and are used with the keyword `cout`.
- The `return` keyword tells the program to return a value to the function `int main`
- After the return statement, execution control returns to the operating system component that launched this program.
- Execution of the code terminates here.

How to declare a structure in C++ programming?

The `struct` keyword defines a structure type followed by an identifier (name of the structure).
Then inside the curly braces, you can declare one or more members (declare variables inside curly braces) of that structure. For example:

```
struct Person
{
```

```
    char name[50];
    int age;
    float salary;
};
```

Here a structure `person` is defined which has three members: `name`, `age` and `salary`.

When a structure is created, no memory is allocated.

The structure definition is only the blueprint for the creating of variables. You can imagine it as a datatype. When you define an integer as below:

```
int foo;
```

The `int` specifies that, variable `foo` can hold integer element only. Similarly, structure definition only specifies that, what property a structure variable holds when it is defined.
Note: Remember to end the declaration with a semicolon **(;)**

How to define a structure variable?

Once you declare a structure `person` as above. You can define a structure variable as:

```
Person bill;
```

Here, a structure variable `bill` is defined which is of type structure P`erson`. When structure variable is defined, only then the required memory is allocated by the compiler.

Considering you have either 32-bit or 64-bit system, the memory of `float` is 4 bytes, memory of `int` is 4 bytes and memory of `char` is 1 byte.
Hence, 58 bytes of memory is allocated for structure variable `bill`.

How to access members of a structure?

The member of structure variable is accessed using a **dot (.)** operator.
Suppose, you want to access `age` of structure variable `bill` and assign it 50 to it. You can perform this task by using following code below:

```
bill.age = 50;
```

Example: C++ Structure

C++ Program to assign data to members of a structure variable and display it.

```
#include <iostream>
using namespace std;

struct Person
{
    char name[50];
    int age;
    float salary;
};

int main()
{
    Person p1;

    cout << "Enter Full name: ";
    cin.get(p1.name, 50);
    cout << "Enter age: ";
    cin >> p1.age;
    cout << "Enter salary: ";
    cin >> p1.salary;
```

```
    cout << "\nDisplaying Information." << endl;
    cout << "Name: " << p1.name << endl;
    cout <<"Age: " << p1.age << endl;
    cout << "Salary: " << p1.salary;

    return 0;
}
```

Output

```
Enter Full name: Magdalena Dankova
Enter age: 27
Enter salary: 1024.4

Displaying Information.
Name: Magdalena Dankova
Age: 27
Salary: 1024.4
```

Here a structure P`erson` is declared which has three members `name`, `age` and `salary`.

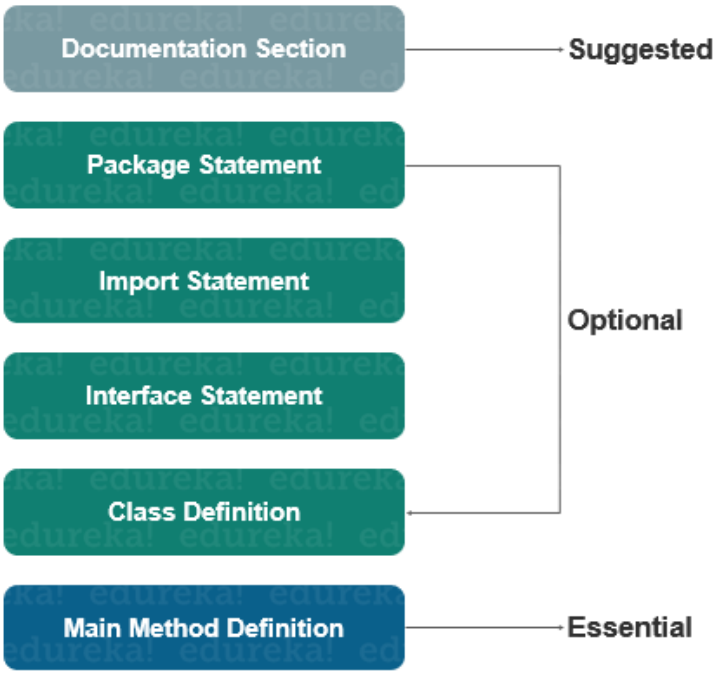
Inside `main()` function, a structure variable `p1` is defined. Then, the user is asked to enter information and data entered by user is displayed.

https://www.programiz.com/cpp-programming/structure-function

Basic Structure of a Java Program

Java programming language is platform-independent and a secure programming language. With a wide variety of applications, Java programming language has been in demand for the last two decades. The out-of-the-box features help java stand apart. Following are the topics discussed in this blog:

- Documentation Section
- Package Statement
- Import Statement
- Interface Section
- Class Definition
- Main Method Class



Documentation Section

It is used to improve the readability of the program. It consists of comments in Java which include basic information such as the method's usage or functionality to make it easier for the programmer to understand it while reviewing or debugging the code. A Java comment is not necessarily limited to a confined space, it can appear anywhere in the code.

The compiler ignores these comments during the time of execution and is solely meant for improving the readability of the Java program.

There are three types of comments that Java supports

- Single line Comment
- Multi-line Comment
- Documentation Comment

Let's take a look at an example to understand how we can use the above-mentioned comments in a Java program.

```
1  // a single line comment is declared like this
2  /* a multi-line comment is declared like this
3  and can have multiple lines as a comment */
4  /** a documentation comment starts with a delimiter and ends with */
```

Package Statement

There is a provision in Java that allows you to declare your classes in a collection called package. There can be only one package statement in a Java program and it has to be at the beginning of the code before any class or interface declaration. This statement is optional; for example, take a look at the statement below.

```
1  package student;
```

This statement declares that all the classes and interfaces defined in this source file are a part of the student package. And only one package can be declared in the source file.

Import Statement

Many predefined classes are stored in packages in Java, an import statement is used to refer to the classes stored in other packages. An import statement is always written after the package statement but it has to be before any class declaration.

We can import a specific class or classes in an import statement. Take a look at the example to understand how import statement works in Java.

```
1  import java.util.Date; //imports the date class
2  import java.applet.*;  //imports all the classes from the java applet package
```

Interface Section

This section is used to specify an interface in Java. It is an optional section which is mainly used to implement multiple inheritance in Java. An interface is a lot similar to a class in Java but it contains only constants and method declarations.

An interface cannot be instantiated but it can be implemented by classes or extended by other interfaces.

```
1  interface stack{
2  void push(int item);
3  void pop();
4  }
```

Class Definition

A Java program may contain several class definitions, classes are an essential part of any Java program. It defines the information about the user-defined classes in a program.

A class is a collection of variables and methods that operate on the fields. Every program in Java will have at least one class with the main method.

Main Method Class

The main method is from where the execution actually starts and follows the order specified for the following statements. Let's take a look at a sample program to understand how it is structured.

```
1  public class Example{
2  //main method declaration
3  public static void main(String[] args){
4  System.out.println("hello world");
5  }
6  }
```

Let's analyze the above program line by line to understand how it works.

public class Example

This creates a class called Example. You should make sure that the class name starts with a capital letter, and the public word means it is accessible from any other classes.

Comments

To improve the readability, we can use comments to define a specific note or functionality of methods, etc for the programmer.

Braces

The curly brackets are used to group all the commands together. To make sure that the commands belong to a class or a method.

public static void main

- When the main method is declared public, it means that it can be used outside of this class as well.
- The word static means that we want to access a method without making its objects. As we call the main method without creating any objects.
- The word void indicates that it does not return any value. The main is declared as void because it does not return any value.
- Main is the method, which is an essential part of any Java program.

String[] args

It is an array where each element is a string, which is named as args. If you run the Java code through a console, you can pass the input parameter. The main() takes it as an input.

System.out.println();

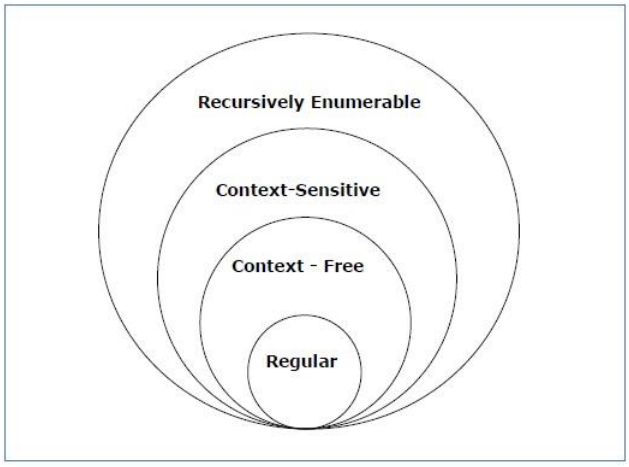
The statement is used to print the output on the screen where the system is a predefined class, out is an object of the PrintWriter class. The method println prints the text on the screen with a new line. All Java statements end with a semicolon.

Chomsky Classification of Grammars

According to Noam Chomosky, there are four types of grammars − Type 0, Type 1, Type 2, and Type 3. The following table shows how they differ from each other −

Grammar Type	Grammar Accepted	Language Accepted	Automaton
Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
Type 2	Context-free grammar	Context-free language	Pushdown automaton
Type 3	Regular grammar	Regular language	Finite state automaton

Take a look at the following illustration. It shows the scope of each type of grammar −



Type - 3 Grammar

Type-3 grammars generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.

The productions must be in the form **X → a or X → aY**

where **X, Y ∈ N** (Non terminal)

and **a ∈ T** (Terminal)

The rule **S → ε** is allowed if **S** does not appear on the right side of any rule.

Example

X → ε
X → a | aY
Y → b

Type - 2 Grammar

Type-2 grammars generate context-free languages.

The productions must be in the form **A → γ**

where **A ∈ N** (Non terminal)

and **γ ∈ (T ∪ N)*** (String of terminals and non-terminals).

These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

Example

S → X a
X → a
X → aX
X → abc
X → ε

Type - 1 Grammar

Type-1 grammars generate context-sensitive languages. The productions must be in the form

α A β → α γ β

where **A ∈ N** (Non-terminal)

and **α, β, γ ∈ (T ∪ N)*** (Strings of terminals and non-terminals)

The strings **α** and **β** may be empty, but **γ** must be non-empty.

The rule **S → ε** is allowed if S does not appear on the right side of any rule. The languages generated by these grammars are recognized by a linear bounded automaton.

Example

AB → AbBc
A → bcA
B → b

Type - 0 Grammar

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.

They generate the languages that are recognized by a Turing machine.

The productions can be in the form of **α → β** where **α** is a string of terminals and nonterminals with at least one non-terminal and **α** cannot be null. **β** is a string of terminals and non-terminals.

Example

S → ACaB
Bc → acB
CB → DB
aD → Db

Regular Expressions

A Regular Expression can be recursively defined as follows −

- **ε** is a Regular Expression indicates the language containing an empty string. **(L (ε) = {ε})**
- **φ** is a Regular Expression denoting an empty language. **(L (φ) = { })**
- **x** is a Regular Expression where **L = {x}**
- If **X** is a Regular Expression denoting the language **L(X)** and **Y** is a Regular Expression denoting the language **L(Y)**, then
    - **X + Y** is a Regular Expression corresponding to the language **L(X) ∪ L(Y)** where **L(X+Y) = L(X) ∪ L(Y)**.
    - **X . Y** is a Regular Expression corresponding to the language **L(X) . L(Y)** where **L(X.Y) = L(X) . L(Y)**
    - **R*** is a Regular Expression corresponding to the language **L(R*)**where **L(R*) = (L(R))***
- If we apply any of the rules several times from 1 to 5, they are Regular Expressions.

Some RE Examples

Regular Expressions	Regular Set
(0 + 10*)	L = { 0, 1, 10, 100, 1000, 10000, … }
(0*10*)	L = {1, 01, 10, 010, 0010, …}

(0 + ε)(1 + ε)	L = {ε, 0, 1, 01}
(a+b)*	Set of strings of a's and b's of any length including the null string. So L = { ε, a, b, aa , ab , bb , ba, aaa…….}
(a+b)*abb	Set of strings of a's and b's ending with the string abb. So L = {abb, aabb, babb, aaabb, ababb, ……..}
(11)*	Set consisting of even number of 1's including empty string, So L= {ε, 11, 1111, 111111, …….}
(aa)*(bb)*b	Set of strings consisting of even number of a's followed by odd number of b's , so L = {b, aab, aabbb, aabbbbb, aaaab, aaaabbb, ……..}
(aa + ab + ba + bb)*	String of a's and b's of even length can be obtained by concatenating any combination of the strings aa, ab, ba and bb including null, so L = {aa, ab, ba, bb, aaab, aaba, ……..}

Regular Sets

Any set that represents the value of the Regular Expression is called a Regular Set.

Properties of Regular Sets

Property 1. *The union of two regular set is regular.*

Proof −

Let us take two regular expressions

RE_1 = a(aa)* and RE_2 = (aa)*

So, L_1 = {a, aaa, aaaaa,.....} (Strings of odd length excluding Null)

and L_2 ={ ε, aa, aaaa, aaaaaa,.....} (Strings of even length including Null)

$L_1 \cup L_2$ = { ε, a, aa, aaa, aaaa, aaaaa, aaaaaa,.....}

(Strings of all possible lengths including Null)

RE ($L_1 \cup L_2$) = a* (which is a regular expression itself)

Hence, proved.

Property 2. *The intersection of two regular set is regular.*

Proof −

Let us take two regular expressions

RE_1 = a(a*) and RE_2 = (aa)*

So, L_1 = { a,aa, aaa, aaaa, ...} (Strings of all possible lengths excluding Null)

L_2 = { ε, aa, aaaa, aaaaaa,.....} (Strings of even length including Null)

$L_1 \cap L_2$ = { aa, aaaa, aaaaaa,.....} (Strings of even length excluding Null)

RE ($L_1 \cap L_2$) = aa(aa)* which is a regular expression itself.

Hence, proved.

Property 3. *The complement of a regular set is regular.*

Proof −

Let us take a regular expression −

RE = (aa)*

So, L = {ε, aa, aaaa, aaaaaa, ...} (Strings of even length including Null)

Complement of **L** is all the strings that is not in **L**.

So, L' = {a, aaa, aaaaa, ...} (Strings of odd length excluding Null)

RE (L') = a(aa)* which is a regular expression itself.

Hence, proved.

Property 4. *The difference of two regular set is regular.*

Proof −

Let us take two regular expressions −

RE_1 = a (a*) and RE_2 = (aa)*

So, L_1 = {a, aa, aaa, aaaa, ...} (Strings of all possible lengths excluding Null)

L_2 = { ε, aa, aaaa, aaaaaa,.....} (Strings of even length including Null)

$L_1 – L_2$ = {a, aaa, aaaaa, aaaaaaa, ...}

(Strings of all odd lengths excluding Null)

RE ($L_1 – L_2$) = a (aa)* which is a regular expression.

Hence, proved.

Property 5. *The reversal of a regular set is regular.*

Proof −

We have to prove $\mathbf{L^R}$ is also regular if **L** is a regular set.

Let, L = {01, 10, 11, 10}

RE (L) = 01 + 10 + 11 + 10

L^R = {10, 01, 11, 01}

RE (L^R) = 01 + 10 + 11 + 10 which is regular

Hence, proved.

Property 6. *The closure of a regular set is regular.*

Proof −

If L = {a, aaa, aaaaa, ...} (Strings of odd length excluding Null)

i.e., RE (L) = a (aa)*

L* = {a, aa, aaa, aaaa , aaaaa,……} (Strings of all lengths excluding Null)

RE (L*) = a (a)*

Hence, proved.

Property 7. *The concatenation of two regular sets is regular.*

Proof −

Let RE_1 = (0+1)*0 and RE_2 = 01(0+1)*

Here, L_1 = {0, 00, 10, 000, 010, ...} (Set of strings ending in 0)

and L_2 = {01, 010,011,.....} (Set of strings beginning with 01)

Then, L_1 L_2 = {001,0010,0011,0001,00010,00011,1001,10010,.....}

Set of strings containing 001 as a substring which can be represented by an RE − (0 + 1)*001(0 + 1)*

Hence, proved.

Identities Related to Regular Expressions

Given R, P, L, Q as regular expressions, the following identities hold −

- ∅* = ε
- ε* = ε
- RR* = R*R
- R*R* = R*

- (R*)* = R*
- RR* = R*R
- (PQ)*P =P(QP)*
- (a+b)* = (a*b*)* = (a*+b*)* = (a+b*)* = a*(ba*)*
- R + ∅ = ∅ + R = R (The identity for union)
- R ε = ε R = R (The identity for concatenation)
- ∅ L = L ∅ = ∅ (The annihilator for concatenation)
- R + R = R (Idempotent law)
- L (M + N) = LM + LN (Left distributive law)
- (M + N) L = ML + NL (Right distributive law)
- ε + RR* = ε + R*R = R*

Arden's Theorem

In order to find out a regular expression of a Finite Automaton, we use Arden's Theorem along with the properties of regular expressions.

Statement **−**

Let **P** and **Q** be two regular expressions.

If **P** does not contain null string, then **R = Q + RP** has a unique solution that is **R = QP***

Proof **−**

R = Q + (Q + RP)P [After putting the value R = Q + RP]

= Q + QP + RPP

When we put the value of **R** recursively again and again, we get the following equation **−**

$R = Q + QP + QP^2 + QP^3$…..

R = Q (ε + P + P^2 + P^3 + ….)

R = QP* [As P* represents (ε + P + P2 + P3 + ….)]

Hence, proved.

Assumptions for Applying Arden's Theorem

- The transition diagram must not have NULL transitions
- It must have only one initial state

Method

Step 1 **−** Create equations as the following form for all the states of the DFA having n states with initial state q_1.

$q_1 = q_1R_{11} + q_2R_{21} + \ldots + q_nR_{n1} + \varepsilon$

$q_2 = q_1R_{12} + q_2R_{22} + \ldots + q_nR_{n2}$

……

……

……

……

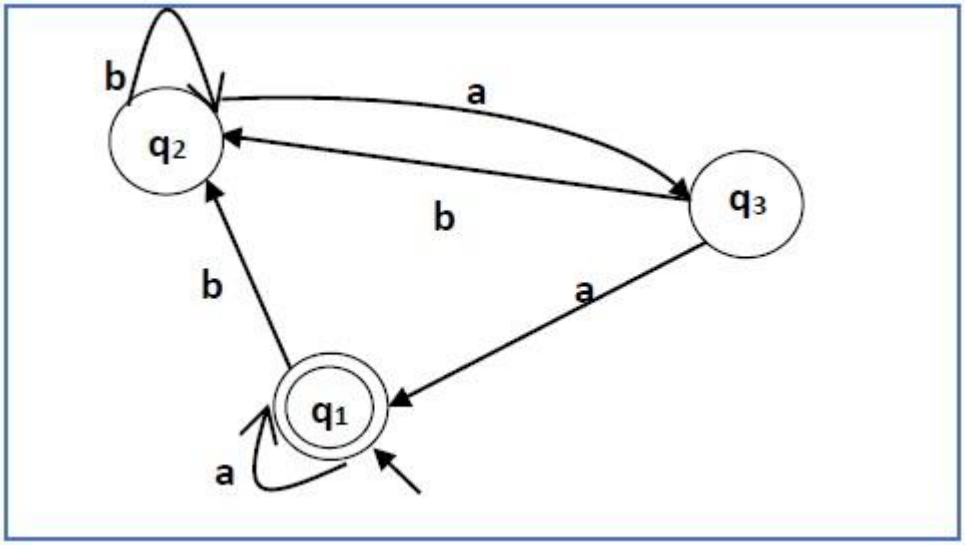
$q_n = q_1R_{1n} + q_2R_{2n} + \ldots + q_nR_{nn}$

$\mathbf{R_{ij}}$ represents the set of labels of edges from $\mathbf{q_i}$ to $\mathbf{q_j}$, if no such edge exists, then $\mathbf{R_{ij} = \emptyset}$

Step 2 − Solve these equations to get the equation for the final state in terms of $\mathbf{R_{ij}}$

Problem

Construct a regular expression corresponding to the automata given below −



Solution −

Here the initial state and final state is $\mathbf{q_1}$.

The equations for the three states q1, q2, and q3 are as follows −

$q_1 = q_1a + q_3a + \varepsilon$ (ε move is because q1 is the initial state0

$q_2 = q_1b + q_2b + q_3b$

$q_3 = q_2a$

Now, we will solve these three equations −

$q_2 = q_1b + q_2b + q_3b$

= $q_1b + q_2b + (q_2a)b$ (Substituting value of q_3)

= $q_1b + q_2(b + ab)$

= $q_1b\ (b + ab)^*$ (Applying Arden's Theorem)

$q_1 = q_1a + q_3a + \varepsilon$

= $q_1a + q_2aa + \varepsilon$ (Substituting value of q_3)

= $q_1a + q_1b(b + ab^*)aa + \varepsilon$ (Substituting value of q_2)

= $q_1(a + b(b + ab)^*aa) + \varepsilon$

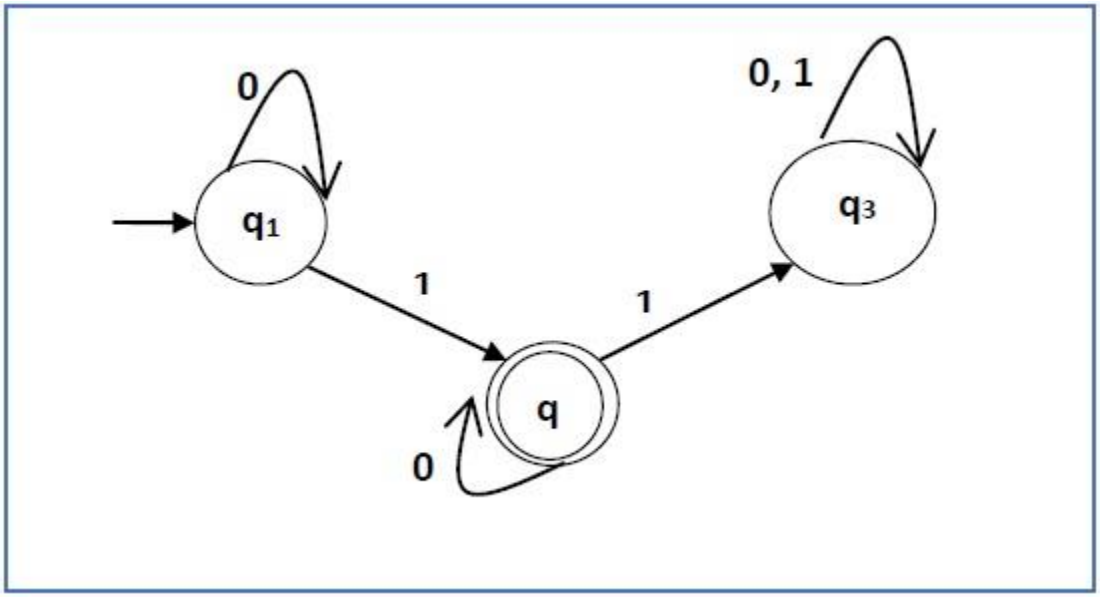
= $\varepsilon\ (a + b(b + ab)^*aa)^*$

= $(a + b(b + ab)^*aa)^*$

Hence, the regular expression is $(a + b(b + ab)^*aa)^*$.

Problem

Construct a regular expression corresponding to the automata given below −



Solution −

Here the initial state is q_1 and the final state is q_2

Now we write down the equations −

$q_1 = q_10 + \varepsilon$

$q_2 = q_11 + q_20$

$q_3 = q_21 + q_30 + q_31$

Now, we will solve these three equations −

$q_1 = \varepsilon 0^*$ [As, $\varepsilon R = R$]

So, $q_1 = 0^*$

$q_2 = 0^*1 + q_20$

So, $q_2 = 0^*1(0)^*$ [By Arden's theorem]

Hence, the regular expression is 0^*10^*.

Simple Expressions

Computers can do more than just print strings—they can also perform calculations. Expressions

Table: Simple Operators

Operator	**Meaning**
*	Multiply
/	Divide
+	Add
-	Subtract
%	Modulus (return the remainder after division)

Multiply (*), divide (/), and modulus (%) have precedence over add (+) and subtract (-). Parentheses, ( ), may be used to group terms. Thus:
(1 + 2) * 4
yields 12, while:
1 + 2 * 4
yields 9.

Example 4-1 computes the value of the expression (1 + 2) * 4.

Example: simple/simple.c

```
int main()
{
   (1 + 2) * 4;
   return (0);
}
```

Although we calculate the answer, we don't do anything with it. (This program will generate a "null effect" warning to indicate that there is a correctly written, but useless, statement in the program.)

Think about how confused a workman would be if we were constructing a building and said,

"Take your wheelbarrow and go back and forth between the truck and the building site."

"Do you want me to carry bricks in it?"

"No. Just go back and forth."

We need to store the results of our calculations.

Variables and Storage

C allows us to store values in *variables*. Each variable is identified by a *variable name*.

In addition, each variable has a *variable type*. The type tells C how the variable is going to be used and what kind of numbers (real, integer) it can hold. Names start with a letter or underscore (_), followed by any number of letters, digits, or underscores. Uppercase is different from lowercase, so the names sam, Sam, and SAM specify three different variables. However, to avoid confusion, you should use different names for variables and not depend on case differences.

Nothing prevents you from creating a name beginning with an underscore; however, such names are usually reserved for internal and system names.

Most C programmers use all-lowercase variable names. Some names like **int**, **while**, **for**, and **float** have a special meaning to C and are considered *reserved words*. They cannot be used for variable names.

The following is an example of some variable names:

```
average         /* average of all grades */
pi              /* pi to 6 decimal places */
```

number_of_students /* number students in this class */

The following are *not* variable names:

```
3rd_entry   /* Begins with a number */
all$done    /* Contains a "$" */
the end     /* Contains a space */
int         /* Reserved word */
```

Avoid variable names that are similar. For example, the following illustrates a poor choice of variable names:

```
total       /* total number of items in current entry */
totals      /* total of all entries */
```

A much better set of names is:

```
entry_total /* total number of items in current entry */
all_total   /* total of all entries */
```

Variable Declarations

Before you can use a variable in C, it must be defined in a *declaration statement*.
A variable declaration serves three purposes:

1. It defines the name of the variable.
2. It defines the type of the variable (integer, real, character, etc.).
3. It gives the programmer a description of the variable. The declaration of a variable answer can be:

```
int answer;     /* the result of our expression */
```

The keyword **int** tells C that this variable contains an integer value. (Integers are defined below.) The variable name is answer. The semicolon (;) marks the end of the statement, and the comment is used to define this variable for the programmer. (The requirement that every C variable declaration be commented is a style rule. C will allow you to omit the comment. Any experienced teacher, manager, or lead engineer will not.)

The general form of a variable declaration is:

type name; /**comment */*

where *type* is one of the C variable types (**int**, **float**, etc.) and *name* is any valid variable name. This declaration explains what the variable is and what it will be used for.

Variable declarations appear just before the main() line at the top of a program.

Integers

One variable type is integer. Integer numbers have no fractional part or decimal point. Numbers such as 1, 87, and -222 are integers. The number 8.3 is not an integer because it contains a decimal point. The general form of an integer declaration is:
int*name*; /* *comment* */

A calculator with an 8-digit display can only handle numbers between 99999999 and -99999999. If you try to add 1 to 99999999, you will get an overflow error. Computers have similar limits. The limits on integers are implementation dependent, meaning they change from computer to computer.

Calculators use decimal digits (0-9). Computers use binary digits (0-1) called bits. Eight bits make a byte. The number of bits used to hold an integer varies from machine to machine. Numbers are converted from binary to decimal for printing.

On most UNIX machines, integers are 32 bits (4 bytes), providing a range of 2147483647 (2^{31}-1) to -2147483648. On the PC, most compilers use only 16 bits (2 bytes), so the range is 32767 (2^{15}-1) to -32768. These sizes are typical. The standard header file *limits.h* defines constants for the various numerical limits. (See Chapter 18, for more information on header files.)

The C standard does not specify the actual size of numbers. Programs that depend on an integer being a specific size (say 32 bits) frequently fail when moved to another machine.

Assignment Statements
Variables are given a value through the use of assignment statements. For example:
answer = (1 + 2) * 4;
is an assignment. The variable answer on the left side of the equal sign (=) is assigned the value of the expression (1 + 2) * 4 on the right side. The semicolon (;) ends the statement.
Declarations create space for variables. Variable not yet assigned value is known as an *uninitialized variable*. The question mark indicates that the value of this variable is unknown.
Assignment statements are used to give a variable a value. For example:
answer = (1 + 2) * 4; is an assignment. The variable answer on the left side of the equals operator (=) is assigned the value of the expression (1 + 2) * 4.

The general form of the assignment statement is:

variable =*expression*;
The = is used for assignment. It literally means: Compute the expression and assign the value of that expression to the variable. (In some other languages, such as PASCAL, the = operator is used to test for equality. In C, the operator is used for assignment.)

Characters

The type **char** represents single characters. The form of a character declaration is:

```
char variable;  /* comment */
```

Characters are enclosed in single quotes ('). 'A', 'a', and '!' are character constants. The backslash character (\) is called the *escape character*. It is used to signal that a special character follows. For example, the characters \" can be used to put a double quote inside a string. A single quote is represented by \'. \n is the newline character. It causes the output device to go to the beginning of the next line (similar to a return key on a typewriter). The characters \\ are the backslash itself. Finally, characters can be specified by *nnn*, where *nnn* is the octal code for the character. Table 4-3 summarizes these special characters. Appendix A contains a table of ASCII character codes.

Table 4-3. Special Characters

Character	Name	Meaning
\b	Backspace	Move the cursor to the left by one character
\f	Form Feed	Go to top of new page
\n	Newline	Go to next line
\r	Return	Go to beginning of current line
\t	Tab	Advance to next tab stop (eight column boundary)
\'	Apostrophe	Character '

NOTE

While characters are enclosed in single quotes (`'`), a different data type, the string, is enclosed in double quotes (`"`). A good way to remember the difference between these two types of quotes is to remember that single characters are enclosed in single quotes. Strings can have any number of characters (including one), and they are enclosed in double quotes.

Declarations

In programming, a **declaration** is a statement describing an identifier, such as the name of a variable or a function. Declarations are important because they inform the compiler or interpreter what the identifying word means, and how the identified thing should be used.

A declaration may be optional or required, depending on the programming language. For example, in the C programming language, all variables must be declared with a specific data type before they can be assigned a value.

Declarations provide information about the name and type of data objects needed during program execution.

Two types of declaration:

- implicit declaration

- explicit declaration

Implicit declaration or default declaration:

They are those declaration which is done by compiler when no explicit declaration or user defined declaration is mentioned.

Example

```
$abc='astring';

$abc=7;
```

In 'perl' compiler implicitly understand that

$abc ='astring' is a string variable and
$abc=7; is an integer variable.

Explicit declaration of data object:

Float A,B;

It is an example of Float A,B, of c language. In explicit we or user explicitly defined the variable type. In this example it specifies that it is of float type of variable which has name A & B.

A "Declaration" basically serves to indicate the desired lifetime of data objects.

Declarations of operations:

- Compiler need the signature of a prototype of a subprogram Or function so it can determine the type of argument is being used and what will be the result type.

*** Before the calling of subprogram, Translator need to know all these information. ***

Example in C language

Float sub(int z, float y)

It declares sub to have the signature

Sub: int xfloat-> float

Purpose of Declarations:

1) Choice of storage representation: AS Translator determines the best storage representation of data types that why it needs to know primarily the information of data type and attribute of a data object.

2) Storage Management: It make to us to use best storage management for data object by providing its information and these information as tells the lifetime of a data object.

For Example:-

In C language we have many options for declaration for elementary data type.

1) Simple Declaration: Like float A,B;

It tells lifetime is only at the end of execution **as lifetime of every data objects can be maximum to end of execution time.**

But simple declaration tells the single block of memory will be allocated.

2) Runtime Declaration: C language and many more language provide us the feature of dynamic memory allocation by keywords "Malloc and Calloc."

So in this special block of memory is allocated in memory and their lifetime is also different.

3) Polymorphic operations: In most language, some special symbol like + to designate any one of the several different operation which depends on type of data or argument is provided.

In this operation has some name like as we discussed + in this case operation symbol is said to be overloaded because it does not designate one specific operation.

Ada: allows programmer to overload subprograms.

ML: Expands this concept with full polymorphism where function has one name but variety of implementation depending on the types of arguments.

4) Type checking:- Declaration is basically for static type checking rather than dynamic.