**2019 final**

**Question 1 – Finite State Machines 1**

Prove by construction that **if the languages A and B are regular then the language that is the intersection of A and B is also regular**.

The intuition behind the proof is to run both automata at the same time and accept if they both accept. Let M1 and M2 denote the automata that recognize A and B respectively. They have sets of states S1 and S2, initial states s10 and s20 , and so on. Define a new automaton M as follows:

S = S1 ×S2, s0 = (s10, s20), Syes =S1yes ×S2yes, δ((s1, s2),a)= (δ1(s1,a),δ2(s2,a)).

Thus, M runs both two automata in parallel, updating both at once, and accepts w if they both end in an accepting state. To complete the proof in gory detail, by induction on the length of w we have

δ\*(s0, w) = [δ1\*(s10, w),δ2\*(s20 ,w)],

since δ\*(s0, w) ∈ Syes if and only if δ1\*(s10, w) ∈ S1yes and δ2\*(s20, w) ∈ S2yes , in which case w ∈ A and w ∈ B, M recognizes A ∩ B.

**Question 2 – Finite State Machines 2**

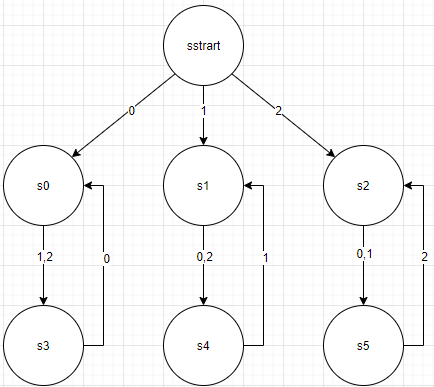
Design a **deterministic finite state machine that recognizes the language of all strings from the alphabet {0, 1, 2} that do not have two identical consecutive characters**. For example, it should not recognize the strings 00 or 11 etc. Ensure that your design shows either a transition table or transition diagram, as well as a formal description of the machine.

Brainstorm:

Accept strings are: 012, 210201,10202012021020210, …

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Sstart | S0 | S1 | S2 | S3 | S4 | S5 |
| 0 | S0 |  | S4 | S5 | S0 |  |  |
| 1 | S1 | S3 |  | S5 |  | S1 |  |
| 2 | S2 | S3 | S4 |  |  |  | S2 |

Formally, a DFA M consists of a set of states S, an input alphabet A, an initial state s0 ∈ S, a subset Syes ⊆ S of accepting states, and a transition function δ : S × A → S,



**Question 3 – Turing Machines 1**

Design a Turing machine that **accepts the set of strings with an equal number of 1’s and 0’s** (in any mixed order). Include a high-level description of its algorithm and draw its transition diagram.

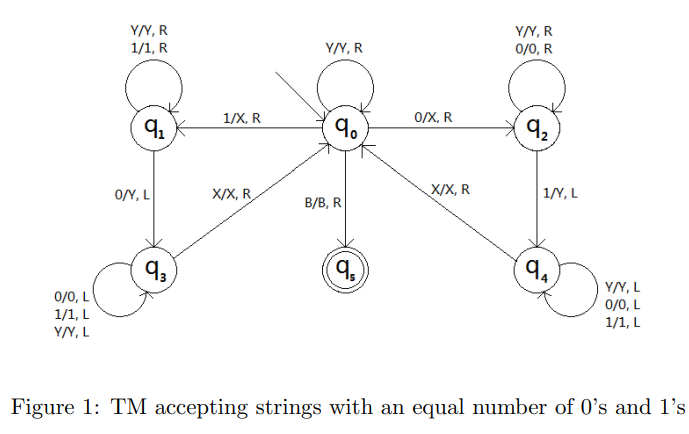
High level description of TM A for { w | w ∈ {0, 1}∗ } A = “On input string w:

1. we start with the first symbol, mark the first symbol as X, move right

2. move right until the corresponding 0 or 1 is found, mark that as Y, move left; otherwise reject

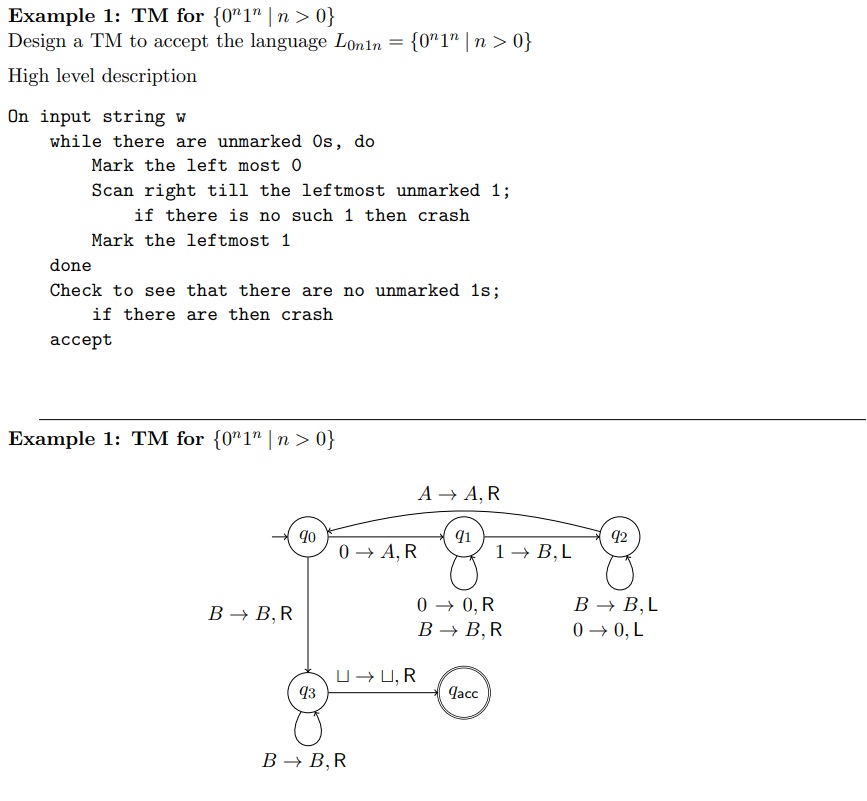
3. move left until a X encountered, move right

4. go stage 1, until we encounter the first symbol as blank, then we accept the string.



**Question 4 – Turing Machines 2**

Show that a **Turing machine is more powerful than a finite state machine by using an example language that is only decidable by a Turing machine**. Provide a high-level description of the Turing machine that decides this language.

****

**Question 5 – Lambda Calculus**

Explain the **concept of Church encoding in the context of Lambda Calculus** and explain **how it is used to create the Church numerals**. Use Church encoding to **derive the Church numerals up to 4** and describe **how they can be used to count in Lambda Calculus**.

**Church encoding** is a means of representing data and operators in the lambda calculus.

The **Church numerals** are a representation of the natural numbers using lambda notation.

**How?** The first step is to create the concept of numbers within λ-Calculus and then define a method of counting between them. Numbers can be encoded by simply applying a function repeatedly n times depending on the number n being represented.

We can then adopt the concept of the natural numbers N as defined by Peano and his so

called Peano numbers to develop counting within the calculus. In Peano numbers, we define the natural numbers N recursively as the successor function S. Thus, we can write that 3 = S(S(S(0))) and likewise for any number required by using the S function recursively. Therefore, we can count in the λ-Calculus by developing the successor “program” or function that takes as input a Church numeral to compute the successor from and the function to use to give us an output Church numeral.

S := λn. λy. λx. y(nyx)

N0 := λf. λx. x

N1 := λf. λx. f(x)

N2 := λf. λx. f(f(x))

N3 := λf. λx. f(f(f(x)))

N4 := λf. λx. f(f(f(f(x))))

S0 = S(λf. λx. x)

= λn. λy. λx. y(nyx)(λf. λx. x)

= λy. λx. y((λf. λx. x)yx)

= λy. λx. y((λx. x)x)

= λy. λx. y(x)

S1 = S(λf. λx. f(x))

= λn. λy. λx. y(nyx)(λf. λx. f(x))

= λy. λx. y((λf. λx. f(x))yx)

= λy. λx. y((λx. y(x))x)

= λy. λx. y(y(x))

S2 = λy. λx. y(y(y(x)))

S3 = λy. λx. y(y(y(y(x))))

S4 = λy. λx. y(y(y(y(y(x)))))

**How count?** Church numerals use n function applications. The natural number n is represented by the higher order function which applies a function n times to an input. 1 is encoded by a function applied once, 2 by a function applied twice and so on.

#some Church numerals:

ZERO = lambda f: lambda x: x

ONE = lambda f: lambda x: f(x)

TWO = lambda f: lambda x: f(f(x))

THREE = lambda f: lambda x: f(f(f(x)))

**Question 6 – Combinators**

Define and discuss **the significance** of the following **two combinators in Lambda Calculus**:

a) **Composition Combinator**

b) **Y Combinator**

Describe **how** each of these combinators **contribute to the Turing completeness of Lambda Calculus.**

1. composition combinator: λg. λf. λx.g(fx)

It allows us to formally encode the idea of composition. So, you have two functions, g and f. you want to be able to apply them to each other in a structured way. So, by feeding g and f into this combinator.

= λx.g(fx)

Obviously, composition is essential for any kind of model of computation, it's because you need to be able to define things in terms of building blocks of simpler functions. And that's what composition does.

1. In order to make λ-Calculus Turing complete, we need the ability to infinitely loop. One way to obtain infinite loops is to do computations recursively until a stopping condition is reached.

What we need is a recursive combinator that takes in a function f but calls f again with the recursive combinator only if the base condition is not satisfied. If the base condition is satisfied, we would like to stop calling it itself and allow β reduction. This is exactly what the Y combinator allows us to do and is defined as

Y := λf. (λx. f(xx))(λx. f(xx))

:= λf. M(λx. f(Mx))

**Question 7 – Quantum Computation 1**

Describe in a paragraph **how the superposition principle in quantum mechanics is used to construct a qubit** and **how the measurement of qubits give rise to the advantages quantum computation has over conventional computation**.

**How qubit?** A qubit is a is a bit that is in a superposition of two states (say on and off or 1 and 0), rather than exclusively in either one of two states as in a classical or binary bit. In other words, a qubit is in a continuum of the two possible states 0 and 1. A qubit still collapses into a binary bit, but like all quantum systems, the final state has a probabilistic outcome given by its probability density. In our Dirac notation, the qubit can be represented as |q> = (u v)’ where u and v are arbitrary complex numbers, so that <q|q> = 1 and has a probability density of |u|2 + |v|2. In other word, the qubit has a |u|2 probability of collapsing into state 0 and |v|2 probability of collapsing into state 1. Thus, qubits can hold multiple bit states at once during a computation before they collapse when we measure them.

**How measurement?** qubits can hold multiple bit states at once during a computation before they collapse when we measure them. A series of qubits then can-do **parallel computations**, so that each thread of that computation, and its corresponding intermediate results, is represented as one of the many superimposed states held in the qubits. In other words, each thread of the parallel computation exists as a single harmonic across the qubits in a cacophony of harmonics created by superposition that represents the entire computation (reduce computation time).

**Question 8 – Quantum Computation 2**

Describe the **Deutsch problem** and explain **why a quantum computer can solve it much faster than a conventional computer**.

**Problem**: The problem Deutsch’s algorithm tackles can now be stated as follows. Given a block box Uf implementing some unknown function f : {0, 1} → {0, 1}, determine whether f is “constant” or “variable”. Here, constant means f always outputs the same bit, i.e. f(0) = f(1), and variable means f outputs different bits on different inputs, i.e. f(0) not= f(1).

**Why?** Of course, there is an easy way to determine whether f is constant or variable — simply evaluate f on inputs 0 and 1, i.e. compute f(0) and f(1), and then check if f(0) = f(1). This naive (classical) solution, however, requires two queries or calls to Uf (i.e. one to compute f(0) and one to compute f(1)). So certainly, at most two queries to Uf suffice to solve this problem. Can we do it with just one query? Classically, the answer turns out to be no.

But quantumly, Deutsch showed how to indeed achieve this with a single query. Suppose one measures the first qubit of the output state |ψ> (this qubit marks which term in the superposition we have) with a standard basis measurement {|0><0|, |1><1|}. Show that the probability of outcome 0 or 1 is |α|2 or |β|2, respectively, and that in each case, the state collapses to either |0>|f(0)> or |1>|f(1)>, respectively. Thus, only one answer f(0) or f(1) can be extracted this way. *The intuition here is you are moving into a superposition of state. So, you have another degree of freedom, you're feeding through possibilities of ones and zeros, rather than just a one or just a zero and that gives you additional information.*

**Tutorial lambda calculus Problem 3**

Describe **how Boolean logic can be defined in Lambda Calculus** and therefore the **three main operations of AND, OR and NOT** can be encoded into it making sure to **explain all the relevant combinators, their arguments and function bodies**.

In λ-Calculus, we define Boolean logic by taking two arguments and defining that if the first is chosen, it represents true and if the second is chosen, then it is false. In other words, we may define true T and false F as the following in λ-Calculus:

T := λx. λy. x == f(x, y) = x

F := λx. λy. y == f(x, y) = y

**The NOT operation** toggles the value, so we need to select the opposite value based on the current argument. The inputs are Booleans p and q, i.e. selector functions, so we can then define the NOT as an opposing selector function

NOT := λp. pFT

NOT T == (λp.pFT)T = T(FT) = F

NOT F == (λp.pFT)F = F(FT) = T

In the body, we can see the Boolean p will select either F if it is true and T if it is false according to our convention for true and false.

Likewise, we can define **the AND operation** as the selector function that requires both p and q are both true, so we can define it as

AND := λp. λq. pqF

AND T T = (λp. λq.pqF)TT = T(TF) = T

AND F T = (λp. λq.pqF)FT = F(TF) = F

In the body, we can see if p is false, return false because both Booleans need to be true and the value of q is not necessary to make the decision, otherwise return the value of q that will decide the final value of the operation.

we can also build **the OR operation** based on similar principles as

OR := λp. λq. pT q

OR T F = (λp. λq.pTq)TF = (λq.TTq)F = T(TF) = T

We select true if p is true, if it is not true then perhaps q is true, so return q. The value of q will decide the result.

**Tutorial lambda calculus Problem 5**

In a paragraph and using a few equations, describe **how one can compute the factorial function using recursion in Lambda Calculus**. Make sure to mention all the relevant combinators and theorems required to achieve your result.

The usual recursive form of the factorial function in most procedural languages would be:

int factorial (int n) {

if n == 0

return 1

else

return n\*fac(n-1) <-- in programming language

}

We can attempt to convert the above procedural ‘function’ directly to λ-Calculus for which we might (incorrectly) write:

FAC = λn. If ISZERO(n)(1)MUL(n)(FAC(PRED(n))))

Instead, because functions in λ-Calculus are anonymous, we can write the above correctly using the Y combinator while setting Z for ISZERO as

FAC = Y(λf. λn.Z(n)(1)MUL(n)(f(PRED(n)))) = YR

and set R = λf. λn. Z(n)(1) MUL(n)(f(PRED(n))).

Relevant combinators:

Y := λf. (λx. f(xx))(λx. f(xx))

YR = (λx.R(xx))( λx.R(xx)) = R((λx.R(xx)( λx.R(xx)) = R(YR) = R(R(YR)) …

PRED - predecessor, the factorial function applied to the predecessor

Z(n) - the predicate is zero

Z := λn. λf. λx.n(λx.F)T

MULT := (λn. λk. λf.n(kf)) 🡪 returns the product of two numbers

PRED := (λn. λf. λx.n(λg. λh.h(gh))( λu.x)( λu.u)) 🡪 returns the predecessor of a number by subtracting 1