UNIT-I

AUTOMATA

Introduction

Why do we study Theory of Computation ?

- Importance of Theory of Computation
- Languages
- Languages and Problems

What is Computation ?

Sequence of mathematical operations ?

  - What are, and are not, mathematical operations?
  -
- Sequence of well-defined operations
-
  - How many operations ?
    - The fewer, the better.
  - Which operations ?
  -
    - The simpler, the better.

What do we study in Theory of Computation ?

- What is computable, and what is not ?

- Basis of
  - Algorithm analysis
  -
  - Complexity theory

- What a computer can and cannot do
- Are you trying to write a non-existing program?
- Can you make your program more efficient?

What do we study in Complexity Theory ?

- What is easy, and what is difficult, to compute ?
- What is easy, and what is hard for computers to do?
- Is your cryptograpic scheme safe?

Applications in Computer Science

- Analysis of algorithms
- Complexity Theory
- Cryptography
- Compilers
- Circuit design

History of Theory of Computation

- 1936 Alan Turing invented the *Turing machine*, and proved that there exists an *unsolvable problem*.
- 1940's Stored-program computers were built.
- 1943 McCulloch and Pitts invented *finite automata*.
- 1956 Kleene invented *regular expressions* and proved the equivalence of regular expression and finite automata.
- 1956 Chomsky defined *Chomsky hierarchy*, which organized languages recognized by different automata into hierarchical classes.
- 1959 Rabin and Scott introduced *nondeterministic finite automata* and proved its equivalence to (deterministic) finite automata.
- 1950's-1960's More works on languages, grammars, and compilers

-
- 1965 Hartmantis and Stearns defined *time complexity*, and Lewis, Hartmantis and Stearns defined *space complexity*.
-
- 1971 Cook showed the first *NP-complete problem*, the *satisfiability* prooblem.
-
- 1972 Karp Showed many other NP-complete problems.

Alphabet and Strings

- An *alphabet* is a finite, non-empty set of symbols.
  - {*0,1* } is a binary alphabet.
  - { *A*, *B*, …, *Z*, *a*, *b*, …, *z* } is an English alphabet.
  -
- A *string* over an alphabet Σ is a sequence of any number of symbols from Σ.
  - *0*, *1*, *11*, *00*, and *01101* are strings over {*0*, *1* }.
  - *Cat*, *CAT*, and *compute* are strings over the English alphabet.
  -
- An *empty string*, denoted by ε, is a string containing no symbol.
-
  - ε is a string over any alphabet.

- The length of a string *x*, denoted by *length*(*x*), is the number of positions of symbols in the string.
-

Let Σ = {*a*, *b*, …, *z*}

length(*automata*) = 8

length(*computation*) = 11
length(ε) = 0

- *x*(*i*), denotes the symbol in the *ith* position of a string *x*, for 1≤ *i* ≤ *length*(*x*).

String Operations

- Concatenation
- Substring
- Reversal

- The concatenation of strings *x* and *y*, denoted by *x·y* or *x y*, is a string *z* such that:
  - $z(i) = x(i)$ for $1 \le i \le length(x)$
  - $z(i) = y(i)$ for $length(x) < i \le length(x) + length(y)$

- Example
  - *automata·computation = automatacomputation*

The concatenation of string *x* for *n* times, where $n \ge 0$, is denoted by x^n

- $x^0 = \varepsilon$
- $x^1 = x$
- $x^2 = x\,x$
- $x^3 = x\,x\,x$
- ...

Substring

Let *x* and *y* be strings over an alphabet Σ

The string *x* is a substring of *y* if there exist strings *w* and *z* over Σ such that *y = w x z*.

- ε is a substring of every string.
- For every string *x*, *x* is a substring of *x* itself.

Example

- ε, *comput* and *computation* are substrings of *computation*.

Reversal

Let *x* be a string over an alphabet Σ

The reversal of the string *x*, denoted by *x r*, is a string such that

- if *x* is ε, then *xr* is ε.
- If *a* is in Σ, *y* is in Σ* and *x = a y*, then *xr = yr a*.

(*automata*)*r*
= (*utomata*)*r a*
= (*tomata*)*r ua*
= (*omata*)*r tua*
= (*mata*)*r otua*
= (*ata*)*r motua*
= (*ta*)*r amotua*
= (*a*)*r tamotua*
= (*ε*)*r atamotua*

- = *atamotua The set of strings created from any number (0 or 1 or ...) of symbols in an alphabet Σ is denoted by Σ*.*
-
- *That is, Σ* = ∪i=∞0 Σi*
  - *Let Σ = {**0, 1**}.*
  - *Σ* = {ε, **0, 1, 00, 01, 10, 11, 000, 001, 010, 011**, ... }.*
  -

- *The set of strings created from at least one symbol (1 or 2 or ...) in an alphabet Σ is denoted by Σ+.*

- *That is, Σ+ = ∪i=∞1 Σ i*

  = ∪i=0..∞ Σi - Σ0
  = ∪ i=0..∞ Σi - {ε}
- *Let Σ = {**0, 1**}. Σ+ = {**0, 1, 00, 01, 10, 11, 000, 001, 010, 011**, ... }.*
  Σ and Σ+ are infinite sets.*

- A language over an alphabet Σ is a set of strings over Σ.
  - Let Σ = {*0, 1*} be the alphabet.
  - *Le* = {∈ωΣ* | the number of *1*'s in ω is even}.
  - ε, *0, 00, 11, 000, 110, 101, 011, 0000, 1100, 1010, 1001, 0110, 0101, 0011,* … are in *Le*

- **Operations on LanguagesComplementation**

- **Union**
- **Intersection**
- **Concatenation**
- **Reversal**
- **Closure**

Complementation

Let *L* be a language over an alphabet Σ.

The complementation of *L*, denoted by $\overline{L}$, is Σ*–*L*.

Example:
Let Σ = {*0, 1*} be the alphabet.
Le = {∈ωΣ* | the number of 1's in ω is even}.
$\overline{Le}$= {∈ωΣ* | the number of 1's in ω is not even}.
$\overline{Le}$= {∈ωΣ* | the number of 1's in ω is odd}.

Union

Let *L1* and *L2* be languages over an alphabet Σ.

The union of *L1* and *L2*, denoted by *L1*∪*L2*, is {*x* | *x* is in *L1* or *L2*}.

Example:
{*x*∈{0,1}*|*x* begins with 0} ∪ {*x*∈{0,1}*|*x* ends with 0}

= {$x \in \{0,1\}^*$| x begins or ends with 0}

Intersection

Let *L1* and *L2* be languages over an alphabet Σ.

The intersection of *L1* and *L2*, denoted by $L1 \cap L2$, is { x | x is in *L1* and *L2*}.

Example:

{ $x \in \{0,1\}^*$| x begins with 0} ∩ { $x \in \{0,1\}^*$| x ends with 0}

= { $x \in \{0,1\}^*$| x begins and ends with 0}

Concatenation

Let *L1* and *L2* be languages over an alphabet Σ.

The concatenation of *L1* and *L2*, denoted by $L1 \cdot L2$, is {$w1 \cdot w2$| *w1* is in *L1* and *w2* is in *L2*}.

Example

{ $x \in \{0,1\}^*$| x begins with 0}·{$x \in \{0,1\}^*$| x ends with 0}

= { $x \in \{0,1\}^*$| x begins and ends with 0 and $length(x) \geq 2$}

{ $x \in \{0,1\}^*$| x ends with 0}·{$x \in \{0,1\}^*$| x begins with 0}

= { $x \in \{0,1\}^*$| x has 00 as a substring}

Reversal

Let *L* be a language over an alphabet Σ.

The reversal of *L*, denoted by *Lr*, is {*wr*| *w* is in *L*}.

Example

{$x \in \{0,1\}^*$| x begins with 0} r

= {$x \in \{0,1\}^*$| x ends with 0}

{$x \in \{0,1\}^*$| x has 00 as a substring} r

= {$x \in \{0,1\}^*$| x has 00 as a sub**string}**

Closure

Let *L* be a language over an alphabet Σ.

The closure of *L*, denoted by *L+*, is { x |for an integer $n \geq 1$, $x = x1x2...xn$ and *x1*, *x2* , ..., *xn* are in *L*}

That is, $L+ = \cup i\infty = 1 \quad Li$

Example:

Let $\Sigma = \{0, 1\}$ be the alphabet.
$Le = \{\in\omega\Sigma^* \mid$ the number of 1's in ω is even$\}$
$\boldsymbol{Le}+ = \{\in\omega\Sigma^* \mid$ the number of 1's in ω is even$\} = \boldsymbol{Le}^*$

Observation about Closure

$L+ = L^* - \{\varepsilon\}$?

Example:

$L = \{\in\omega\Sigma^* \mid$ the number of 1's in ω is even$\}$
$\boldsymbol{L}+ = \{\in\omega\Sigma^* \mid$ the number of 1's in ω is even$\} = \boldsymbol{Le}^*$

Why?
$L^* = L+ \cup \{\varepsilon\}$?

- **Languages and ProblemsProblem**
  - **Example: What are prime numbers > 20?**
  -
- **Decision problem**
  - **Problem with a YES/NO answer**
  - **Example: Given a positive integer *n*, is *n* a prime number > 20?**
  -
- **Language**
  - **Example: {*n* | *n* is a prime number > 20}**

Finite Automata

A simple model of computation

- Deterministic finite automata (DFA)
  - How a DFA works
  - How to construct a DFA
- Non-deterministic finite automata (NFA)
  - How an NFA works
  - How to construct an NFA
- Equivalence of DFA and NFA
- Closure properties of the class of languages accepted by FA

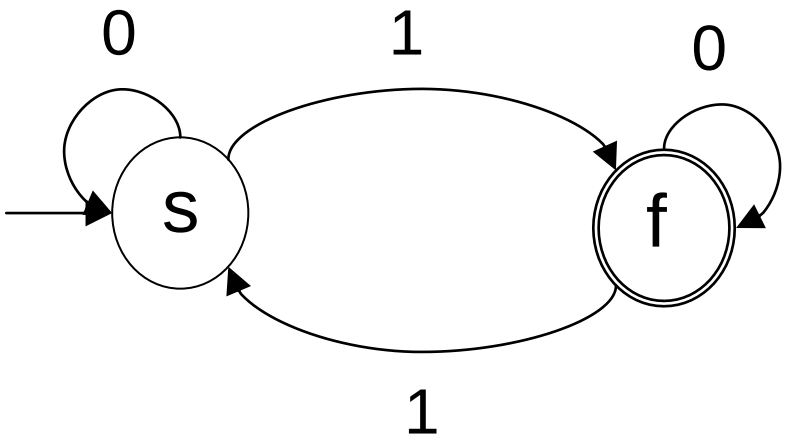
Finite Automata (FA)

- Read an input string from tape
- Determine if the input string is in a language
- Determine if the answer for the problem is "YES" or "NO" for the given input on the tape

How does an FA work?

- At the beginning,
  - an FA is in the *start state* (*initial state*)
  - its tape head points at the first cell
- For each move, FA
  - reads the symbol under its tape head
  - changes its state (according to the *transition function*) to the *next state* determined by the symbol read from the tape and its current state
  - move its tape head to the right one cell

- **When does an FA stop working?**
- **When it reads all symbols on the tape**
- **Then, it gives an answer if the input is in the specific language:**
  - **Answer "YES" if its last state is a *final state***
  - **Answer "NO" if its last state is not a *final state***



Q	*a*	δ(*q*,*a*)
S	0	s
S	1	f
F	0	f
F	1	s

- **How to define a DFA**
-
- a 5-tuple (Q, Σ, δ, s, F), where
  - a set of states Q is a finite set
  - an alphabet Σ is a finite, non-empty set
  - a start state s in Q
  - a set of final states F contained in Q
  - a transition function δ is a function $Q \times \Sigma \rightarrow Q$
  -
- *See formal definition*

How an FA works

Definition

- Let $M = (Q, \Sigma, \delta, s, F)$ be a DFA, and $\omega \in \Sigma^*$. We say M accepts ω if $(s, \omega) \vdash^*_M (f, \varepsilon)$, when $f \in F$. Otherwise, we say M rejects ω.

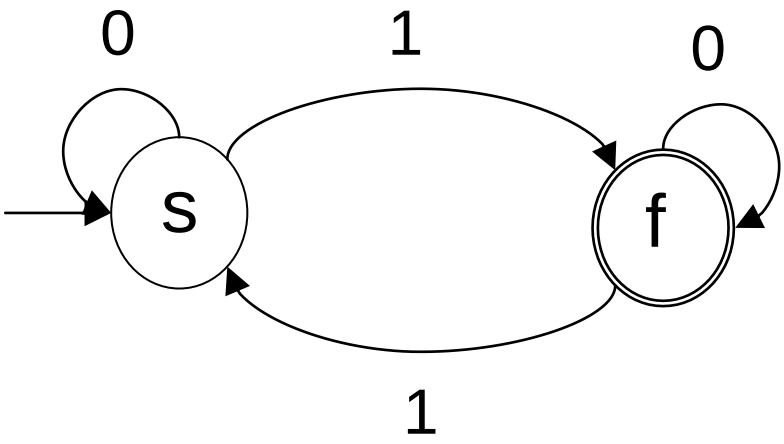
  $(s, 001101) \vdash^*_M (f, \varepsilon)$ $\therefore$ M accepts 001101
  $(s, 01001) \vdash^*_M (s, \varepsilon)$ $\therefore$ M rejects 01001

Language accepted by a DFA

Let $M = (Q, \Sigma, \delta, s, F)$ be a DFA. The language accepted by M, denoted by $L(M)$ is the set of strings accepted by M. That is, $L(M) = \{\Sigma \in \omega^* | (s, \omega) \vdash^*_M (f, \varepsilon)$ for some $f \in F\}$

Example:

- $L(M) = \{x \in \{0,1\}^* \mid$ the number of 1's in x is odd$\}$.



How to construct a DFA

- Determine what a DFA need to memorize in order to recognize strings in the language.
  - Hint: the property of the strings in the language
- Determine how many states are required to memorize what we want.
  - final state(s) memorizes the property of the strings in the language.
- Find out how *the thing we memorize* is changed once the next input symbol is read.
  - From this change, we get the transition function.

Constructing a DFA: Example

- Consider $L = \{\in\omega\{0,1\}^* | \omega$ has both 00 and 11 as substrings$\}$.
- Step 1: decide what a DFA need to memorize
- Step 2: how many states do we need?
- Step 3: construct the transition diagram

Constructing a DFA: Example

- **Consider $L = \{\in\omega\{0,1\}^* | \omega$ represents a binary number divisible by 3$\}$.**
  - ***L* = {0, 00, 11, 000, 011, 110, 0000, 0011, 0110, 1001, 00000, ...}.**
  -
- Step 1: **decide what** a DFA **need to memorize**
  - **remember**ing **that the portion of the string that has been read** so far **is divisible by 3**
  -
- Step 2: how many states do we need?
-
  - 2 states remembering that
    - **the string that has been read is divisible by** 3
    - **the string that has been read is** in**divisible by** 3**.**
  - **3 states remembering that**
    - **the string that has been read is divisible by** 3
    - **the string that has been read** - 1 **is divisible by** 3**.**
    - **the string that has been read** - 2 **is divisible by** 3**.**

Using 2 states

- Reading **a string *w* representing a number divisible by 3.**
  - Next symbol is 0. ***w* 0, which is 2*w, is also divisible by 3.**
    - If *w*=9 is divisible by 3**,** so is 2**w*=18.
  - Next symbol is 1. ***w* 1, which is 2*w +1,** may or may not be divisible **by 3.**
    - I**f** 8 **is indivisible by** 3, so is 17.
    - If **4 is indivisible by 3,** but 9 is divisible.
- U**sing these two states is not sufficient.**
- Using 3 states
- Each **state remembers the remainder** of the number divided by 3**.**
- I**f the portion of the string that has been read so far, say *w*, represents the number whose remainder is 0 (or, 1, or 2),**
  - If the next symbol is 0, what is the remainder of *w* 0?
  - If the next symbol is 1, what is the remainder of *w* 1?

Current number	Current remainder	Next symbol	New number	New remainder
3n	0	0	6n	0
3n	0	1	6n+1	1
3n+1	1	0	6n+2	2
3n+1	1	1	6n+3	0
3n+2	2	0	6n+4	1
3n+2	2	1	6n+5	2

How to define an NFA

- a 5-tuple (Q, Σ, δ, s, F), where
  - a set of states Q is a finite set
  - an alphabet Σ is a finite, non-empty set
  - a start state s in Q
  - a set of final states F contained in Q
  - a transition function δ is a function $Q\times(\cup\Sigma\{\varepsilon\})\rightarrow 2Q$
- *See formal definition*

Definition

- *Let M = (Q, Σ, δ, s, F) be a non-deterministic finite automaton, and (q0, ω0) and (q1, ω1) be two configurations of M.*
- *We say (q0, ω0) yields (q1, ω1) in one step, denoted by (q0, ω0) —/M (q1, ω1), if q1 ∈ δ (q0, a,), and ω0=a ω1, for some a ∈ Σ ∪ {ε}.*

Definition

- *Let M = (Q, Σ, δ, s, F) be an NFA, and (q0, ω0) and (q1, ω1) be two configurations of M. (q0, ω0) yields (q1, ω1) in zero step or more, denoted by (q0, ω0) —/*M (q1, ω1), if*
  - *q0= q1 and ω0 = ω1, or*
  - *(q0, ω0) —/M (q2, ω2) and (q2, ω2) —/*M (q1, ω1) for some q2 and ω2.*

Definition

- *Let M = (Q, Σ, δ, s, F) be an NFA, and ω ∈ Σ*. We say M accepts ω if (s, ω) —/*M (f, ε), when f ∈ F. Otherwise, we say M rejects ω.*

Language accepted by an NFA

- *Let M = (Q, Σ, δ, s, F) be an NFA.*
- *The language accepted by M, denoted by L(M) is the set of strings accepted by M. That is, L(M) = {Σ∈ω*| (s,ω) —/*M (f, ε) for some f∈F}*

DFA and NFA are equivalent

Md and Mn are equivalent $\leftrightarrow$ *L(Md) = L(Mn).*

DFA and NFA are equivalent $\leftrightarrow$

- *For any DFA Md, there exists an NFA Mn such that Md and Mn are equivalent. (part 1)*
- *For any NFA Mn, there exists a DFA Md such that Md and Mn are equivalent. (part 2)*

Part 1 of the equivalence proof

- *For any DFA Md, there exists an NFA Mn such that Md and Mn are equivalent*

Proof: Let Md be any DFA. We want to construct an NFA Mn such that L(Mn) = L(Md).
From the definitions of DFA and NFA, if M is a DFA then it is also an NFA.
Then, we let Mn = Md.
Thus, L(Md) = L(Mn). ν

- *For any NFA Mn, there exists a DFA Md such that Md and Mn are equivalent.*

Proof: Let Mn = (Q, Σ, δ, s, F) be any NFA. We want to construct a DFA Md such that L(Md) = L(Mn).
First define the closure of q, denoted by E(q).
Second, construct a DFA Md=(2Q, Σ, δ', E(s), F')
Finally, prove $\forall \Sigma \in \omega^* \ \exists f \in F \ (s,\omega) \vdash^*_{Mn} (f, \varepsilon) \leftrightarrow \exists f' \in F' \ (E(s), \omega) \vdash^*_{Md} (f', \varepsilon)$. ν

Closure of state q

- *Let M = (Q, Σ, δ, s, F) be an NFA, and q∈Q.*
- *The closure of q, denoted by E(q), is*
  - *the set of states which can be reached from q without reading any symbol.*
  - *{p∈Q| (q, ε) $\vdash^*_M$ (p, ε)}*
- *If an NFA is in a state q, it can also be in any state in the closure of q without reading any input symbol.*

Example of closure

Constructing the equivalent DFA

Let Mn = (Q, Σ, δ, s, F) be any NFA. We construct a DFA Md =(2Q, Σ, δ', E(s), F'), where :

- $\delta'(q',a) = \cup\{r \in E(p) \mid p \in \delta(q,a)\}$ *and*
- $F' = \{f \subseteq Q \mid f \cap F \neq \varnothing\}$

E(q0)	E(q1)	E(q2)	E(q3)	E(q4)
q0, q1, q2, q3	q1, q2, q3	q2	q3	q3,q4

Prove property of δ and δ'

Let Mn = (Q, Σ, δ, s, F) be any NFA, and Md = (2Q, Σ, δ', E(s), F') be a DFA, where

- $\delta'(q', a) = \cup\{r \in E(p) \mid p\delta(q,a)\}$ *and*
- $F' = \{f \subseteq Q \mid f \cap F \neq \varnothing\}$

Prove $\forall \Sigma \in \omega^*, \exists f \in F\ (s,\omega) \vdash^*_{Mn} (f, \varepsilon) \leftrightarrow \exists f' \in F'\ (E(s), \omega) \vdash^*_{Md} (f', \varepsilon)$ *and* $f \in f'$ *by induction.*

Prove a more general statement $\forall \Sigma \in \omega^*, \forall p, q \in Q\ (p,\omega) \vdash^*_{Mn} (q, \varepsilon) \leftrightarrow (E(p), \omega) \vdash^*_{Md} (q', \varepsilon)$ *and* $q \in q'$.

Proof

Part I:

For any string ω *in* Σ^*, *and states q and r in Q, there exists* $R \subseteq Q$ *such that*

$(q, \omega) \vdash^*_{Mn} (r, \varepsilon) \rightarrow (E(q), \omega) \vdash^*_{Md} (R, \varepsilon)$ *and* $r \in R$.

Basis:

Let ω *be a string in* Σ^*, *q and r be states in Q, and* $(q, \omega) \vdash^*_{Mn} (r, \varepsilon)$ *in 0 step.*

Because $(q, \omega) \vdash^*_{Mn} (r, \varepsilon)$ *in 0 step, we know (1) q=r , and (2)* $\omega = \varepsilon$.

Then, $(E(q), \omega) = (E(r), \varepsilon)$.

Thus, $(E(q), \omega) \vdash^*_{Md} (E(r), \varepsilon)$.

That is, there exists R=E(r) such that $r \in R$ *and* $(E(q),\omega) \vdash^*_{Md} (R, \varepsilon)$.

Induction hypothesis:

For any non-negative integer k, string ω *in* Σ^*, *and states q and r in Q, there exists* $R \subseteq Q$:

$(q, \omega) \vdash^*_{Mn} (r, \varepsilon)$ *in k steps -> (E(q),* ω*)* $\vdash^*_{Md} (R, \varepsilon)$ *and* $r \in R$.

Induction step:

Prove, for any non-negative integer k, string ω in Σ^*, and states q and r in Q, there exists $R \subseteq Q$:

$(q, \omega) \vdash^*_{Mn} (r, \varepsilon)$ in k+1 steps -> $(E(q), \omega) \vdash^*_{Md} (R, \varepsilon)$ and $r \in R$.

Let ω be a string in Σ^*, q and r be states in Q, and $(q, \omega) \vdash^*_{Mn} (r, \varepsilon)$ in k+1 steps.

Because $(q, \omega) \vdash^*_{Mn} (r, \varepsilon)$ in k+1 steps and $k \geq 0$, there exists a state p in Q and a string $\in \alpha \Sigma^*$ such that $(q, \omega) \vdash^*_{Mn} (p, a)$ in k steps and $(p, a) \vdash_{Mn} (r, \varepsilon)$ for some $a \in \Sigma \cup \{\varepsilon\}$.
From the induction hypothesis and $(q, \omega) \vdash^*_{Mn} (p, a)$ in k steps, we know that there

exists $P \subseteq Q$ such that $(E(q), \omega) \vdash^*_{Md} (P, a)$ and $p \in P$.
Since $(p, a) \vdash_{Mn} (r, \varepsilon)$, $r \in \delta(p, a)$.
From the definition of $'\delta$ of Md, $E(\delta(p, a)) \subseteq {}'\delta(P, a)$ because $p \in P$.

Because $r \in \delta(p, a)$ and $E(\delta(p, a)) \subseteq {}'\delta(P, a)$, $r \in {}'\delta(P, a)$.
Then, for $R = {}'\delta(P, a)$, $(P, a) \vdash^*_{Md} (R, \varepsilon)$ and $r \in R$.
Thus, $(E(q), \omega) \vdash^*_{Md} (P, a) \vdash^*_{Md} (R, \varepsilon)$ and $r \in R$.

Part II:

For any string ω in Σ^*, and states q and r in Q, there exists $R \subseteq Q$ such that $r \in R$ and

$(E(q), \omega) \vdash^*_{Md} (R, \varepsilon)$ -> $(q, \omega) \vdash^*_{Mn} (r, \varepsilon)$.

Proof

Basis:
Let ω be a string in Σ^*, q and r be states in Q, R be a subset of Q such that $r \in R$ and $(E(q), \omega) \vdash^*_{Md} (R, \varepsilon)$ in 0 step.
Because $(E(q), \omega) \vdash^*_{Md} (R, \varepsilon)$ in 0 step, $E(q)=R$ and $\omega=\varepsilon$.
From the definition of E, ${}'\delta(q, \varepsilon)=R$ because $E(q)=R$.
Then, for any $r \in R$, $(q, \omega) \vdash^*_{Mn} (r, \varepsilon)$.
That is, there exists $R=E(q)$ such that $r \in R$ and $(q, \omega) \vdash^*_{Mn} (r, \varepsilon)$.
Induction hypothesis:
For any non-negative integer k, string ω in Σ^*, and states q and r in Q, there exists $R \subseteq Q$ such that $r \in R$ and:

$(E(q), \omega) \vdash^*_{Md} (R, \varepsilon)$ in k steps -> $(q, \omega) \vdash^*_{Mn} (r, \varepsilon)$.

Induction step:
Prove, for any non-negative integer k, string ω in Σ^*, and states q and r in Q, there exists $R \subseteq Q$ such that $r \in R$:

$(E(q), \omega) \vdash^*_{Md} (R, \varepsilon)$ in k+1 steps -> $(q, \omega) \vdash^*_{Mn} (r, \varepsilon)$.

Let ω be a string in Σ^*, q and r be states in Q, and $(E(q), \omega) \vdash^*_{Md} (R, \varepsilon)$ in k+1 steps.

*Because (E(q), ω) –/*Md (R, ε) in k+1 steps and k≥0, there exists P∈2Q (i.e. P⊆Q) and a string ∈αΣ* such that ω=αa, (E(q), α) –/*Md (P,ε) in k steps and (P, a) –/Md (R, ε) for some a∈Σ.*
*From the induction hypothesis and (E(q), α) –/*Md (P, ε) in k steps, we know that there exists p∈P such that (q, α)–/*Mn(p,ε) (i.e. (q, αa) –/*Mn (p, a)).*
Since (P, a) –/Md (R, ε), there exists r∈R such that r= δ(p, a).
*Then, for some r∈R, (p, a) –/*Mn (r, ε).*
*Thus, (q, ω) –/*Mn (p, a) –/*Mn (r, ε) for some r∈R.*

Closure Properties

- *The class of languages accepted by FA's is closed under the operations*
  - *Union*
  - *Concatenation*
  - *Complementation*
  - *Kleene's star*
  - *Intersection*

The class of languages accepted by FA is closed under union.
Proof:
Let MA = (QA, Σ, δA, sA, FA) and
MB = (QB, Σ, δB, sB, FB) be any FA.
We construct an NFA M =
(Q, Σ, δ, s, F) such that

- *Q = QA ∪ QB ∪ {s}*
- *δ = δA ∪ δA ∪ {(s, ε, {sA, sB})}*
- *F = FA ∪ FB*

To prove L(M) = L(MA) ∪L(MB), we prove:

I. *For any string ∈ωΣ* ∈ωL(MA) or ∈ωL(MB) → ∈ωL(M) &*
II. *For any string ∈ωΣ* ∉ωL(MA) and ∉ωL(MB). → ∉ωL(M)*

For I, consider (a) ∈ωL(MA) or (b) ∈ωL(MB).
For (a), let ∈ωL(MA).
*From the definition of strings accepted by an FA, there is a state fA in FA such that (sA, ω) |-*MA (fA, ε).*
*Because δAδ⊂, (sA, ω) |-*M (fA, ε) also.*
Because sAδ∈(s,ε), (s, ω) |-M (sA, ω).
*Thus, (s, ω) |-M (sA, ω) |-*M (fA, ε).*
Because fA∈ F, ∈ωL(M).
Similarly for (b).
For (II), let ∉ωL(MA)∪L(MB).
Because (s, ε, {sA, sB})δ∈, either (s,ω) |-M (sA,ω) or (s, ω) |-M (sB, ω) only.
*Because ∉ωL(MA), there exists no fA in FA such that (sA,ω) |-*MA (fA,ε).*
*Because ∉ωL(MB), there exists no fB in FB such that (sB, ω) |-*MB (fB, ε).*
*Since there is no transition between states in QA and QB in M, there exists no state f in F=FA∪FB such that (s, ω) |-M (sA, ω) |-*M (fA, ε) or (s, ω) |-M (sB, ω) |-*M (fB, ε).*

That is, $\notin \omega L(M)$.
Thus, $L(M) = L(MA) \cup L(MB)$.

Closure under concatenation

The class of languages accepted by FA is closed under intersection.
Proof: Let L1 and L2 be languages accepted by FA.
$L1 \cap L2 = (\bar{L1} \cup \bar{L2})$
By the closure property under complementation, there are FA accepting $\bar{L1}$ and $\bar{L2}$.
By the closure property under union, there is an FA accepting $\bar{L1} \cup \bar{L2}$.
By the closure property under complementation, there is an FA accepting$(\bar{L1} \cup \bar{L2})$.
Thus, the class of languages accepted by FA is closed under intersection.
Let MA = (QA, Σ, δA, sA, FA) and
MB = (QB, Σ, δB, sB, FB) be any FA.
We construct an NFA M = (Q, Σ, δ, s, F) such that

- *$Q = QA \times QB$*
- *$\delta = \delta A \times \delta A$ (i.e. $\delta((qA,qB),a) = \delta A(qA,a) \times \delta B(qB,a)$)*
- *$s = (sA, sB)$*
- *$F = FA \times FB$*

- ***Check list***

Basic

- *Explain how DFA/NFA work (configuration, yield next configuration)*
- *Find the language accepted by DFA/NFA*
- *Construct DFA/NFA accepting a given language*
- *Find closure of a state*
- *Convert an NFA into a DFA*
- *Prove a language accepted by FA*
- *Construct FA from other FA's*

Advanced

- *Prove DFA/NFA accepting a language*
- *Prove properties of DFA/NFA*
  - *Configuration change*
  - *Under some modification*
  - *etc.*
- *Prove some properties of languages accepted by DFA/NFA*
  - *Under some modification*
- *Surprise!*

UNIT: II : REGULAR EXPRESSIONS &LANGUAGES

Regular Languages

- *Regular expressions*
- *Regular languages*
- *Equivalence between languages accepted by FA and regular languages*
- *Closure Properties*

Regular Expressions

Regular expression over alphabet Σ

- *$\varnothing$ is a regular expression.*
- *ε is a regular expression.*
- *For any $a\Sigma\in$, a is a regular expression.*
- *If r1 and r2 are regular expressions, then*
  - *(r1 + r2) is a regular expression.*
  - *(r1 $\cdot$ r2) is a regular expression.*
  - *(r1*) is a regular expression.*
- *Nothing else is a regular expression.*

- *$\varnothing$ is a regular language corresponding to the regular expression $\varnothing$.*
- *{ε} is a regular language corresponding to the regular expression ε.*
- *For any symbol $a\Sigma\in$, {a} is a regular language corresponding to the regular expression a.*
- *If L1 and L2 are regular languages corresponding to the regular expression r1 and r2, then*
  - *$L1\cup L2$, $L1\cdot L2$, and L1* are regular languages corresponding to (r1 + r2) , (r1 $\cdot$ r2), and (r1*).*

Simple examples

Let Σ= {0,1}.

- *{Σ∈α*|α does not contain 1's}*
  - *(0*)*
- *{Σ∈α*|α contains 1's only}*
  - *(1·(1*)) (which can can be denoted by (1+))*
- *Σ**
  - *((0+1)*)*
- *{Σ∈α*|α contains only 0's or only 1's}*
  - *((00*)+(11*))*

Some more notations

Let Σ= {0,1}.

- *Parentheses in regular expressions can be omitted when the order of evaluation is clear.*
  - *((0+1)*) ≠ 0+1**
  - *((0*)+(1*)) = 0* + 1**
- *For concatenation, · can be omitted.*
- *r· r· r… r is denoted by rn.*

Let Σ = {0,1}.

- *{Σ∈α*| α contains odd number of 1's}*
  - *0*(10*10*)*10**
- *{Σ∈α*| any two 0's in α are separated by three 1's}*
  - *1*(0111)*01* + 1**
- *{Σ∈α*| α is a binary number divisible by 4}*
  - *(0+1)*00*
- *{Σ∈α*| α does not contain 11}*
  - *0*(10+)* (1+ε) or (0+10)* (1+ε)*

Notation

Let r be a regular expression.
The regular language corresponding to the regular expression r is denoted by L(r).

Some rules for language operations

Let r, s and t be languages over {0,1}

$r + \varnothing = \varnothing + r = r$

$r + s = s + r$

$r\varepsilon = \varepsilon r = r$

$r\varnothing = \varnothing r = \varnothing$

$r(s + t) = rs + rt$

$r+ = r\,r^*$

Rewrite rules for regular expressions

Let r, s and t be regular expressions over {0,1}.

$\varnothing^* = \varepsilon$

$\varepsilon^* = \varepsilon$

$(r + \varepsilon)+ = r^*$

$r^* = r^*(r + \varepsilon) = r^*\,r^* = (r^*)^*$

$(r^*s^*)^* = (r + s)^*$

Closure properties of the class of regular languages (Part 1)

Theorem: *The class of regular languages is closed under union, concatenation, and Kleene's star.*

Proof: *Let L1 and L2 be regular languages over Σ.*
Then, there are regular expressions r1 and r2 corresponding to L1 and L2.
By the definition of regular expression and regular languages, $r1+r2$,$r1 \cdot r2$, and $r1^*$ are regular expressions corresponding to $L1 \cup L2$, $L1 \cdot L2$, and $L1^*$.
Thus, the class of regular languages is closed under union, concatenation, and Kleene's star.

Equivalence of language accepted by FA and regular languages

*To show that the **languages accepted by FA and regular languages** are equivalent, we need to prove:*

- *For any regular language L, there exists an FA M such that L = L(M).*
- *For any FA M, L(M) is a regular language.*

For any regular language L, there exists an FA M such that L = L(M)

Proof:

Let L be a regular language.
Then, ∃ a regular expression r corresponding to L.
We construct an NFA M, from the regular expression r, such that L=L(M).
Basis:
If r = ε, M is
If r = ∅, M is
If r = {a} for some a ∈ Σ, M is

Proof (cont'd)

Induction hypotheses: *Let r1 and r2 be regular expressions with less than n operations. And, there are NFA's M1 and M2 accepting regular languages corresponding to L(r1) and L(r2).*

Induction step: *Let r be a regular expression with n operations. We construct an NFA accepting L(r).*

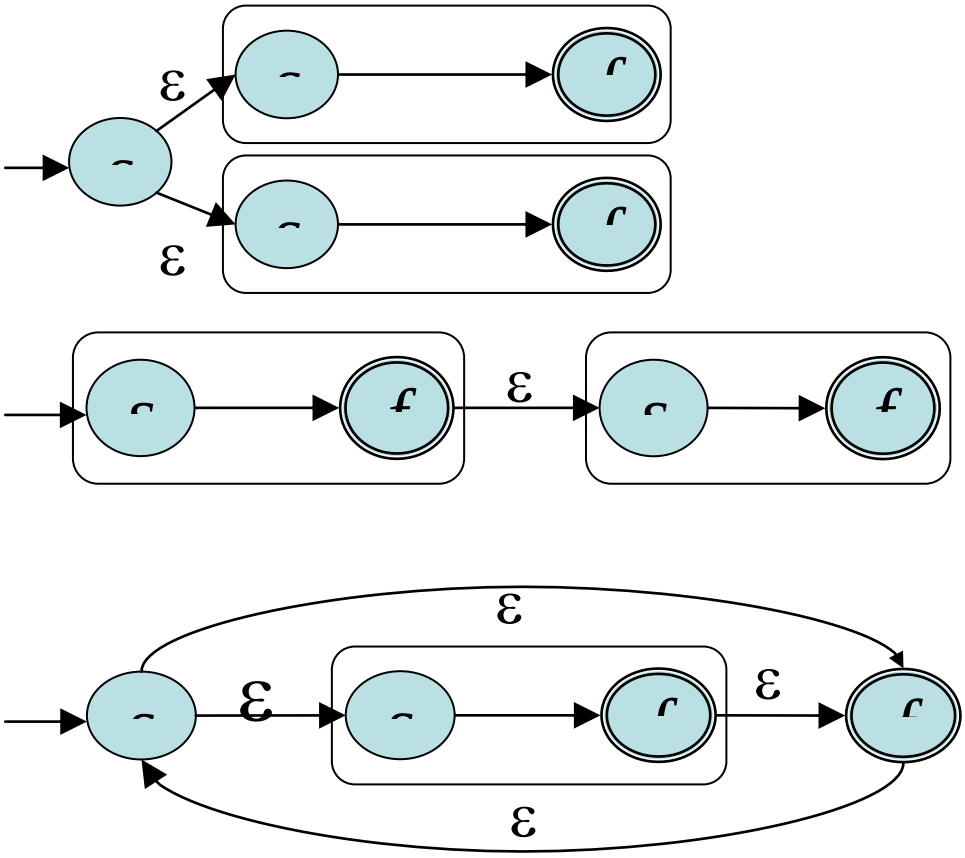
r can be in the form of either r1+r2, r1·r2, or r1, for regular expressions r1 and r2 with less than n operations.*

If r = r1+r2, then M is

If r = r1·r2, then M is

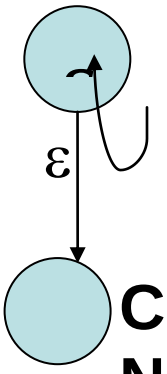
If r = r1, then M is*

Therefore, there is an NFA accepting L(r) for any regular expression r.



Constructing NFA for regular expressions

- $0^*(10^+)^*$ $(1+\varepsilon)$



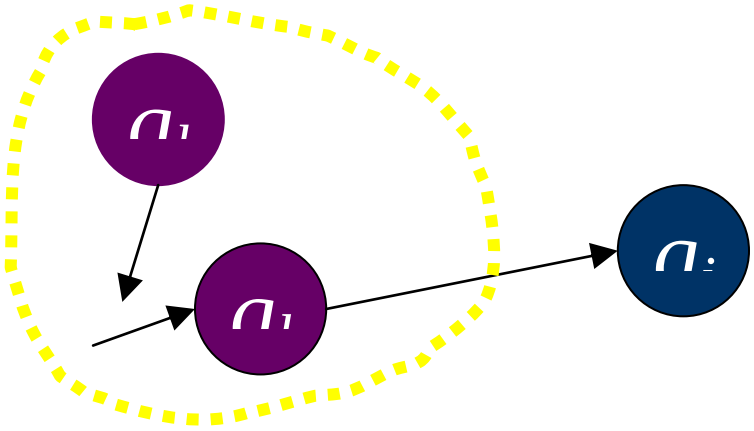
Can these two states be merged? NO

- **Be careful when you decide to merge some**

For any FA *M*, L(*M*) is a regular language

Proof: *Let M = (Q, Σ, δ, q1, F) be an FA, where Q={qi| 1 ≤ i ≤ n} for some positive integer n.*

Let R(i, j, k) be the set of all strings in that drive M from state qi to state qj while passing through any state ql , for l ≤ k. (i and j can be any states)



Proof (cont'd)

We prove that *L*(*M*) is a regular language by showing that there is a regular expression corresponding to *L*(*M*), by induction.

Basis: *R*(*i*, *j*, *0*) corresponds to a regular expression *a* if *i*≠ *j* and *a* + ε if *i*= *j* for some *a*Σ∈.

Induction hypotheses: Let *R*(*i*, *j*,*k-1*) correspond to a regular expression, for any *i*, *j*, *k* ≤ *n*.

Induction step: *R*(*i*, *j*, *k*) = *R*(*i*, *j*, *k-1*) ∪ *R*(*i*, *k*, *k-1*) *R*(*k*, *k*, *k-1*)* *R*(*k*, *j*, *k-1*) also corresponds to a regular expression because *R*(*i*, *j*, *k-1*), *R*(*i*, *k*, *k-1*), *R*(*k*, *k*, *k-1*) and *R*(*k*, *j*, *k-1*) correspond to some regular expressions and union, concatenation, and Kleene's star are allowed in regular expressions.

Therefore, *L*(*M*) is also a regular language because *L*(*M*) = + *R*(*1*, *f*, *n*) for all *qf* in *F*.

Pumping Lemma

Let L be a regular language.
Then, there exists an integer $n \geq 0$ such that for every string x in L that $|x| \geq n$, there are strings u, v, and w such that

- $x = u\ v\ w$,
- $v \neq \varepsilon$,
- $|u\ v| \leq n$, and
- for all $k \geq 0$, $u\ v^k\ w$ is also in L

Any language L is not a regular language *if* for any integer $n \geq 0$, there is a string x in L such that $|x| \geq n$, for any strings u, v, and w,

- $x \neq u\ v\ w$, or
- $v = \varepsilon$, or
- Not ($|u\ v| \leq n$), or
- there is $k \geq 0$, $u\ v^k\ w$ is not in L

Any language L is not a regular language ***if***

- for any integer $n \geq 0$,
- there is a string x in L such that $|x| \geq n$,
- for any strings u, v and w, such that $x = u\ v\ w$, $v \neq \varepsilon$, and $|u\ v| \leq n$,
  - there is $k \geq 0$, $u\ v^k\ w$ is not in L

- Given a language L.
- Let n be any integer ≥ 0 .
- Choose a string x in L that $|x| \geq n$.
- Consider all possible ways to chop x into u, v and w such that $v \neq \varepsilon$, and $|uv| \leq n$.
- For all possible u, v, and w, show that there is $k \geq 0$ such that $u\ v^k\ w$ is not in L.
- Then, we can conclude that L is not regular.

Prove $\{0^i 1^i \mid i \geq 0\}$ is not regular

Let $L = \{0^i 1^i \mid i \geq 0\}$.
Let n be any integer ≥ 0.
Let $x = 0^n 1^n$.
Make sure that x is in L and $|x| \geq n$.
The only possible way to chop x into u, v, and w such that $v \varepsilon \neq$, and $|uv| \leq n$ is:
$u = 0^p$, $v = 0^q$, $w = 0^{n-p-q} 1^n$, where $0 \leq p < n$ and $0 < q \leq n$
Show that there is $k \geq 0$, $uv^k w$ is not in L.
$uv^k w = 0^p 0^{qk} 0^{n-p-q} 1^n = 0^{p+qk+(n-p-q)} 1^n = 0^{n+q(k-1)} 1^n$
If $k \neq 1$, then $n+q(k-1) \neq n$ and $uv^k w$ is not in L.
Then, L is not regular.

Let $L = \{0^i 1^i \mid i \geq 0\}$.
Let n be any integer ≥ 0, and $m = \lceil n/2 \rceil$.
Let $x = 0^m 1^m$.
Make sure that x is in L and $|x| \geq n$.
Possible ways to chop x into u, v, and w such that $v \neq \varepsilon$, and $|uv| \leq n$ are:

- **$u = 0^p$, $v = 0^q$, $w = 0^{m-p-q} 1^m$, where $0 \leq p < m$ and $0 < q \leq m$**
- **$u = 0^p$, $v = 0^{m-p} 1^q$, $w = 1^{m-q}$, where $0 \leq p < m$ and $0 < q \leq m$**
- **$u = 0^m 1^p$, $v = 1^q$, $w = 1^{m-p-q}$, where $0 \leq p < m$ and $0 < q \leq m$**

Show that there is $k \geq 0$, $uv^k w$ is not in L.

- **$u=0^p$, $v=0^q$, $w= 0^{m-p-q} 1^m$, where where $0 \leq p < m$ and $0 < q \leq m$**
  $uv^k w = 0^p 0^{qk} 0^{m-p-q} 1^m = 0^{m+q(k-1)} 1^m$ is not in L if $k \neq 1$.
- **$u=0^p$, $v=0^{m-p} 1^q$, $w=1^{m-q}$, where where $0 \leq p < m$ and $0 < q \leq m$**
  $uv^k w = 0^p (0^{m-p} 1^q)^k 1^{m-q}$ is not in L if $k \neq 1$.
- **$u=0^m 1^p$, $v=1^q$, $w=1^{m-p-q}$, where where $0 \leq p < m$ and $0 < q \leq m$**
  $uv^k w = 0^m 1^p 1^{qk} 1^{m-p-q} = 0^m 1^{m+q(k-1)}$ is not in L if $k \neq 1$.

Then, L is not regular.

Prove {1*i*|*i* is prime} is not regular

Let *L* = {*1i*| *i* is prime}.
Let *n* be any integer ≥0.
Let *p* be a prime ≥ *n*, and *w* = *1p*.
Only one possible way to chop *w* into *x*, *y*, and *z* such that *y* ≠ ε, and |*x y*| ≤ *n* is:
x* = *1q*, *y* = *1r*, *z* = *1p-q-r*, where 0≤*q*<*n* and 0<*r*<*n
Show that there is *k* ≥ 0, *x yk z* is not in *L*.
x yk z* = *1q 1rk 1p-q-r* = *1q+rk+(p-q-r)* = *1p+r(k-1)
If *k*=*p*+*1*, then *p*+*r(k-1)* = *p(r+1)*, which is not a prime.
Then, *x yk z* is not in *L*.
Then, *L* is not regular.

Using closure property

Let ∇ be a binary operation on languages and the class of regular languages is closed under ∇. (∇ can be ∪, ∩, or -)

- If L1 and L2 are regular, then L1∇L2 is regular.
- If L1∇L2 is not regular, then L1 or L2 are not regular.
- If L1∇L2 is not regular but L2 is regular, then L1 is not regular.

Let L={w∈{0,1}*| the number of 0's and 1's in w are equal}.
Let R= {0i1i| i ≥ 0}.
R = 0*1* ∩ L
We already prove that R is not regular.
But 0*1* is regular.
Then, L is not regular.
Let ∇ be a unary operation on a language and the class of regular languages is closed under ∇.
(∇ can be complement or *)

- If L is regular, then ∇L is regular.
- If ∇L is not regular, then L is not regular.

Prove that {w∈{0,1}*| the number of 0's and 1's in w are not equal} is not regular

Let L = {w∈{0,1}*| the number of 0's and 1's in w are not equal}.
Let R = $\bar{L}$ = {w∈{0,1}*| the number of 0's and 1's in w are equal}.
We already prove that R is not regular.
Then, L is not regular.

Check list

- **Find the language described by a regular exp.**
- **Construct regular exp. describing a given language**
- **Convert a regular exp. into an FA**
- **Convert an FA into a regular exp.**
- **Prove a language is regular**
  - **By constructing a regular exp.**
  - **By constructing an FA**
  - **By using closure properties**

- **Construct an FA or a regular exp. for the intersection, union, concatenation, complementation, and Kleene's star of regular languages**
- **Prove other closure properties of the class of regular lang**

UNIT-III

CONTEXT FREE GRAMMAR AND LANGUAGES

Pushdown automata

Pushdown automata differ from finite state machines in two ways:

1. They can use the top of the stack to decide which transition to take.
2. They can manipulate the stack as part of performing a transition.

Pushdown automata choose a transition by indexing a table by input signal, current state, and the symbol at the top of the stack. This means that those three parameters completely determine the transition path that is chosen. Finite state machines just look at the input signal and the current state: they have no stack to work with. Pushdown automata add the stack as a parameter for choice.

Pushdown automata can also manipulate the stack, as part of performing a transition. Finite state machines choose a new state, the result of following the transition. The manipulation can be to push a particular symbol to the top of the stack, or to pop off the top of the stack. The automaton can alternatively ignore the stack, and leave it as it is. The choice of manipulation (or no manipulation) is determined by the transition table.

Put together: Given an input signal, current state, and stack symbol, the automaton can follow a transition to another state, and optionally manipulate (push or pop) the stack.

In general pushdown automata may have several computations on a given input string, some of which may be halting in accepting configurations while others are not. Thus we have a model which is technically known as a "nondeterministic pushdown automaton" (NPDA). Nondeterminism means that there may be more than just one transition available to follow, given an input signal, state, and stack symbol. If in every situation only one transition is available as continuation of the computation, then the result is a deterministic pushdown automaton (DPDA), a strictly weaker device.

If we allow a finite automaton access to two stacks instead of just one, we obtain a more powerful device, equivalent in power to a Turing machine. A linear bounded automaton is a device which is more powerful than a pushdown automaton but less so than a Turing machine.

Pushdown automata are equivalent to **context-free grammars:** for every context-free grammar, there exists a pushdown automaton such that the language generated by the grammar is identical with the language generated by the automaton, which is easy to prove. The reverse is true, though harder to prove: for every pushdown automaton there exists a context-free grammar such that the language generated by the automaton is identical with the language generated by the grammar.

Formal Definition

A PDA is formally defined as a 7-tuple:

where

- is a finite set of *states*
- is a finite set which is called the *input alphabet*
- is a finite set which is called the *stack alphabet*
- is a mapping of into , the *transition relation*, where Γ^* means "a finite (maybe empty) list of element of Γ" and denotes the empty string.
- is the *start state*
- is the *initial stack symbol*
- is the set of *accepting states*

An element is a transition of M. It has the intended meaning that M, in state , with on the input and with as topmost stack symbol, may read a, change the state to q, pop A, replacing it by pushing . The letter ε (epsilon) denotes the empty string and the component of the transition relation is used to formalize that the PDA can either read a letter from the input, or proceed leaving the input untouched.

In many texts the transition relation is replaced by an (equivalent) formalization, where

- is the *transition function*, mapping into finite subsets of .

Here $\delta(p,a,A)$ contains all possible actions in state p with A on the stack, while reading a on the input. One writes for the function precisely when for the relation. Note that *finite* in this definition is essential.

Computations

a step of the pushdown automaton

In order to formalize the semantics of the pushdown automaton a description of the current situation is introduced. Any 3-tuple is called an instantaneous description (ID) of M, which includes the current state, the part of the input tape that has not been read, and the contents of the stack (topmost symbol written first). The transition relation δ defines the step-relation of M on instantaneous descriptions. For instruction there exists a step , for every and every .

In general pushdown automata are nondeterministic meaning that in a given instantaneous description (p,w,β) there may be several possible steps. Any of these steps

can be chosen in a computation. With the above definition in each step always a single symbol (top of the stack) is popped, replacing it with as many symbols as necessary. As a consequence no step is defined when the stack is empty.

Computations of the pushdown automaton are sequences of steps. The computation starts in the initial state q_0 with the initial stack symbol Z on the stack, and a string w on the input tape, thus with initial description (q_0,w,Z). There are two modes of accepting. The pushdown automaton either accepts by final state, which means after reading its input the automaton reaches an accepting state (in F), or it accepts by empty stack (), which means after reading its input the automaton empties its stack. The first acceptance mode uses the internal memory (state), the second the external memory (stack).

Formally one defines

1. with and (final state)
2. with (empty stack)

Here represents the reflexive and transitive closure of the step relation meaning any number of consecutive steps (zero, one or more).

For each single pushdown automaton these two languages need to have no relation: they may be equal but usually this is not the case. A specification of the automaton should also include the intended mode of acceptance. Taken over all pushdown automata both acceptance conditions define the same family of languages.

Theorem. For each pushdown automaton M one may construct a pushdown automaton M' such that $L(M) = N(M')$, and vice versa, for each pushdown automaton M one may construct a pushdown automaton M' such that $N(M) = L(M')$

The following is the formal description of the PDA which recognizes the language by final state:

PDA for (by final state)

, where

$Q = \{p,q,r\}$

$\Sigma = \{0,1\}$

$\Gamma = \{A,Z\}$

$F = \{r\}$

δ consists of the following six instructions:

$(p,0,Z,p,AZ)$, $(p,0,A,p,AA)$, (p,ε,Z,q,Z), (p,ε,A,q,A), $(q,1,A,q,\varepsilon)$, and (q,ε,Z,r,Z).

In words, in state p for each symbol 0 read, one A is pushed onto the stack. Pushing symbol A on top of another A is formalized as replacing top A by AA. In state q for each symbol 1 read one A is popped. At any moment the automaton may move from state p to state q, while it may move from state q to accepting state r only when the stack consists of a single Z.

There seems to be no generally used representation for PDA. Here we have depicted the instruction (p,a,A,q,α) by an edge from state p to state q labelled by $a;A / \alpha$ (read a; replace A by α).

Understanding the computation process

accepting computation for 0011

The following illustrates how the above PDA computes on different input strings. The subscript M from the step symbol is here omitted.

(a) Input string = 0011. There are various computations, depending on the moment the move from state p to state q is made. Only one of these is accepting.

(i) . The final state is accepting, but the input is not accepted this way as it has not been read.
(ii) . No further steps possible.
(iii) . Accepting computation: ends in accepting state, while complete input has been read.

(b) Input string = 00111. Again there are various computations. None of these is accepting.

(i) . The final state is accepting, but the input is not accepted this way as it has not been read.
(ii) . No further steps possible.
(iii) . The final state is accepting, but the input is not accepted this way as it has not been (completely) read.

Pushdown Automata

As Fig. 5.1 indicates, a *pushdown automaton* consists of three components: 1) an input tape, 2) a control unit and 3) a stack structure. The input tape consists of a linear configuration of cells each of which contains a character from an alphabet. This tape can be moved one cell at a time to the left. The stack is also a sequential structure that has a first element and grows in either direction from the other end. Contrary to the tape head associated with the input tape, the head positioned over the current stack element can read and write special stack characters from that position. The current stack element is always the top element of the stack, hence the name ``stack''. The control unit contains both tape heads and finds itself at any moment in a particular state.

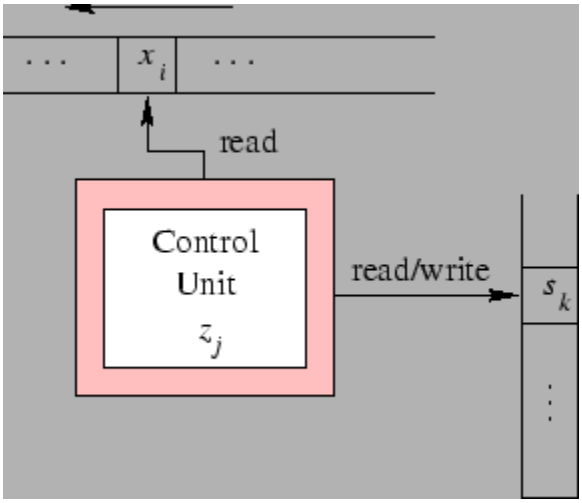


Figure 5.1: Conceptual Model of a Pushdown Automaton

Definition

A (non-deterministic) **finite state pushdown automaton** (abbreviated PDA or, when the context is clear, an automaton) is a 7-tuple $\mathcal{P} = (X, Z, \mathbf{S}, R, z_A, S_A, Z_F)$, where

- $X = \{x_1, \cdots, x_m\}$ is a finite set of *input symbols*. As above, it is also called an *alphabet*. The *empty symbol* λ is *not* a member of this set. It does, however, carry its usual meaning when encountered in the input.
- $Z = \{z_1, \cdots z_n\}$ is a finite set of states.
- $\mathbf{S} = \{s_1, \cdots, s_p\}$ is a finite set of stack symbols. In this case $\lambda \in \mathbf{S}$.
- $R \subseteq ((X \cup \{\lambda\}) \times Z \times \mathbf{S}^*) \times (Z \times \mathbf{S}^*))$ is the *transition relation*.
- z_A is the *initial state*.
- S_A is the *initial stack symbol*.
- $Z_F \subseteq K$ is a distinguished set of *final states*.

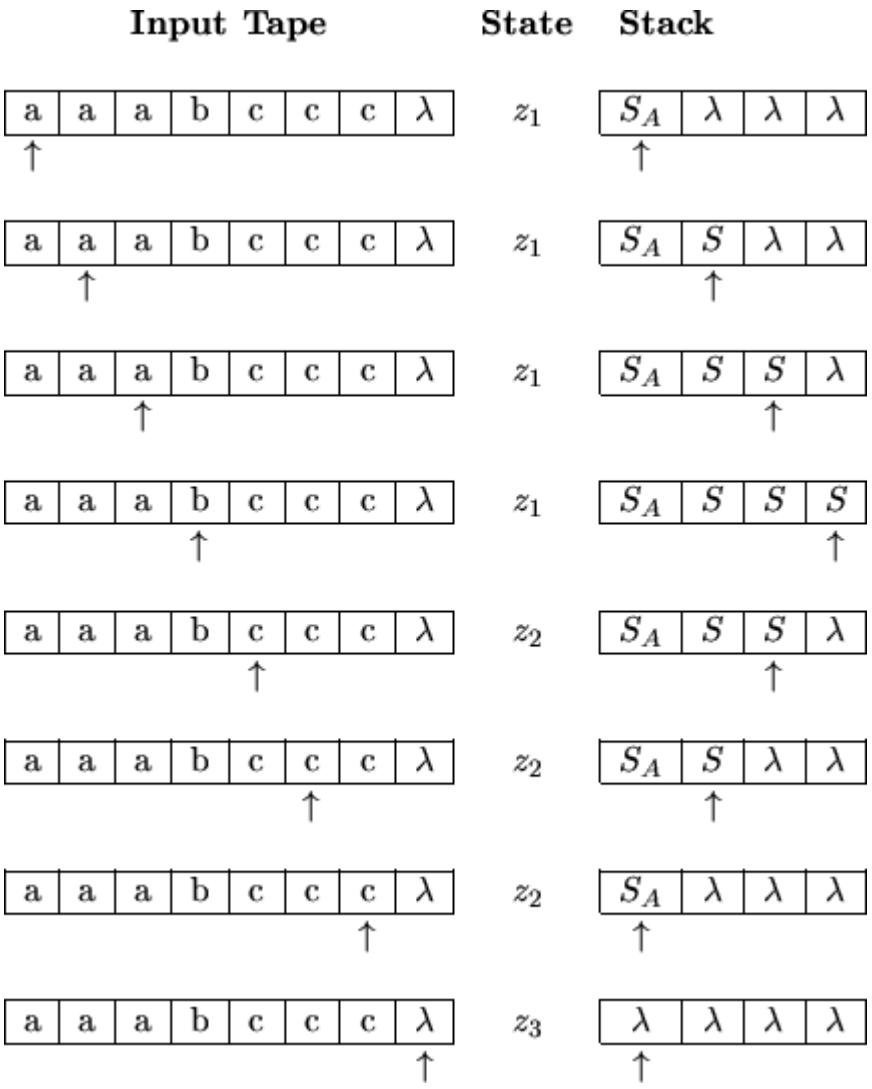


Figure 5.3: Derivation of the String a^3bc^3

Context-Free Languages

As will be recalled from the last chapter there were two basic ways to determine whether a given string belongs to the language generated by some finite state automaton: One could verify that the string brings the automaton to a final state or one could derive, or, better, produce, the string in the regular grammar corresponding to the automaton. The same option holds for PDAs.

Definition

A **context-free grammar** is a grammar $\mathcal{G} = (X, T, S, R)$ for which all rules, or productions, in R have the special form $A \Rightarrow \alpha$, for $A \in X - T$ and $\alpha \in X^*$.

Additionally, for any two strings $u, v \in X^*$ write $u \Rightarrow v$ (u **directly produces** v) if and only if (1) $u = u_1Au_2$ for $u_1, u_2 \in X^*$ and $A \in X - T$ and (2) $v = v_1 \alpha v_2$ and $A \Rightarrow \alpha$, $\alpha \in X^*$, is a production from R. The reduction $u \Rightarrow v$ is also called a **direct production**. Finally, write $u \Rightarrow v$ for two strings $u, v \in X^*$ (u **derives** v) if there is a sequence $u = u_0 \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_n = v$ of direct productions $u_i \Rightarrow u_{i+1}$ from R. The **length** of the derivation is n. The **language generated by** $\mathcal{G}$ is $\{x \in T^* \mid S \Rightarrow^* x\}$.

Thus, the definition just articulates the reduction of A to α in *any context* in which A occurs. It is trivial that every regular language is context-free. The obverse, as will be seen presently, is not true. Before proving the central theorem for this section two typical examples are given.

Example 1

Consider $\mathcal{G} = (X, T, R, S)$ with $T = \{a, b\}$ and $X = \{S, a, b, \lambda\}$. The productions, or grammar rules, are: $S \Rightarrow aSb \mid \lambda$. Then it is clear that $L(\mathcal{G}) = \{a^n b^n \mid n \geq 0\}$. From the previous chapter it is known that this language is not regular.

Example 2: A Grammar for Arithmetic Expressions

Let

$$X = \{E, T, F, id, +, -, *, /, (,), a, b, c\}$$

and $T = \{a, b, c, +, -, *, /, (,)\}$. The start symbol S is E and the productions are as follows:

$E \Rightarrow E + T \mid E - T \mid T$

$T \Rightarrow T*F \mid T/F \mid F$

$F \Rightarrow (E) \mid id$

$id \Rightarrow a \mid b \mid c$

Then the string $(a + b)*c$ belongs to $L(\mathcal{G})$. Indeed, it is easy to write down a derivation of this string:

$$E \Rightarrow T \Rightarrow T*F \Rightarrow F*F \Rightarrow (E)*F \Rightarrow (E + T)*F$$
$$\Rightarrow (T + T)*F \Rightarrow (F + T)*F \Rightarrow (id + T)*F \Rightarrow (a + T)*F$$
$$\Rightarrow (a + F)*F \Rightarrow (a + id)*F \Rightarrow (a + b)*F \Rightarrow (a + b)*id \Rightarrow (a + b)*c$$

The derivation just adduced is *leftmost* in the sense that the leftmost nonterminal was always substituted. Although derivations are in general by no means unique, the leftmost one is. The entire derivation can also be nicely represented in a tree form, as Fig. 5.4 suggests.

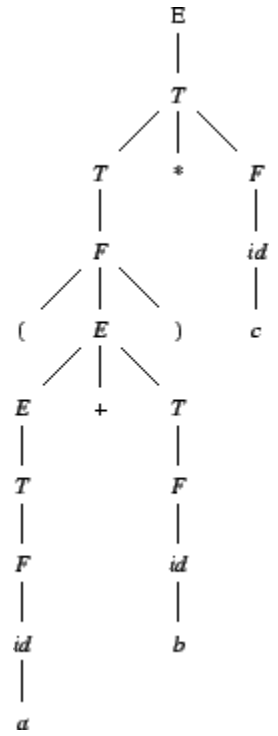


Figure 5.4: Derivation Tree for the Expression $(a + b)^*c$

The internal nodes of the derivation, or syntax, tree are nonterminal symbols and the frontier of the tree consists of terminal symbols. The start symbol is the *root* and the derived symbols are *nodes*. The *order* of the tree is the maximal number of successor nodes for any given node. In this case, the tree has order 3. Finally, the *height* of the tree is the length of the longest path from the root to a leaf node, *i.e.* a node that has no successor. The string $(a + b)^*c$ obtained from the concatenation of the leaf nodes together from left to right is called the *yield* of the tree.

The expected relation between pushdown automata and context-free languages is enunciated in the following theorem.

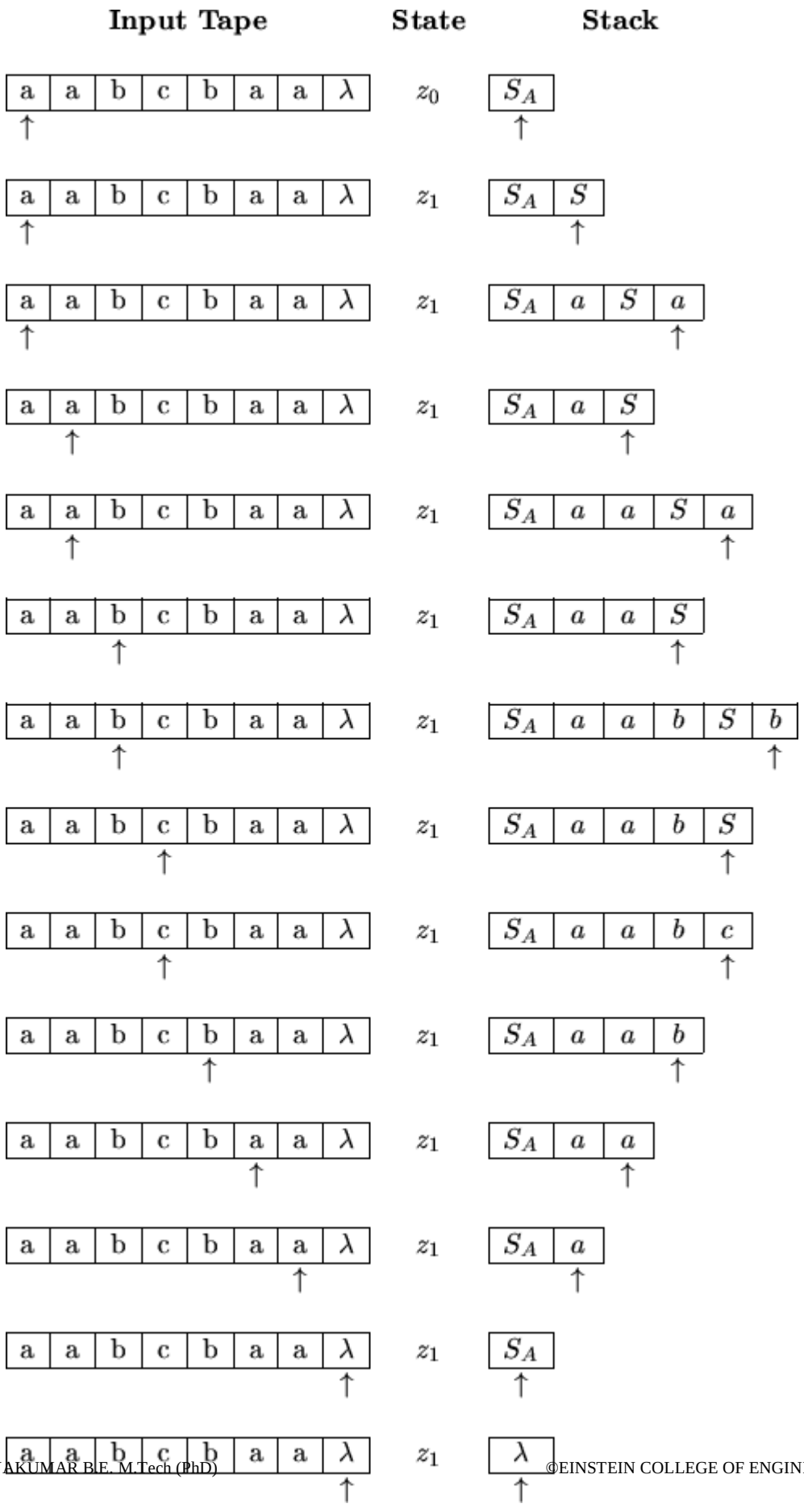

Input Tape
State
Stack
a a b c b a a λ
z_0
S_A
z_1
S
a
b
c
λ

Figure 5.5: Derivation of the String a^2bcba^2

Conversely, assume $\mathcal{P}$ is a PDA. To clarify the subsequent definitions the following discussion on the internal operation of $\mathcal{P}$ is offered. The goal is, of course, to concoct a context-free grammar that executes a leftmost derivation of every string that $\mathcal{P}$ accepts. If $\mathcal{P}$ were as simple as the example in the first part of this proof, namely, that after pushing the very first nontrivial symbol (not S_A) onto the stack $\mathcal{P}$ remains in a single state z_1, then it would be very straightforward to reverse the above process and construct $\mathcal{G}$ from $\mathcal{P}$. Basically, if x is the input string write $x = x'ax''$, where x' is that part of x that has already been processed (a so-called *prefix* of x) and ax'' is the rest of x whose first input symbol is a. Then the direct production of configurations of $\mathcal{P}$ of the form $(ax'', z_1, AA') \Rightarrow (x'', z_1, \alpha A')$ corresponds to the grammar rule $A \Rightarrow a\alpha$, resulting in the reduction $x'AA' \Rightarrow xa\alpha A'$. Thus the sequence of stack moves from the above-mentioned example commences with S_A and, after popping that symbol, derives the string a^2bcba^2, as can be seen by inspecting the stack column in Fig. 5.5.

Unfortunately, the general case is considerably more complicated, because $\mathcal{P}$'s state transitions also enter into the picture. Proceeding naively, one could reduce $\mathcal{P}$ to a 2 state PDA $\mathcal{P}'$ of the aforementioned type by pushing *pairs* (z, A) of states and stack symbols from $\mathcal{P}$ onto $\mathcal{P}'$'s stack, thus imitating $\mathcal{P}$'s calculation of input strings. Thus, when $\mathcal{P}$ is in state z and pushes A onto the stack, $\mathcal{P}'$ pushes (z, A) onto its stack. The reader is invited to pause to discover the fatal shortcoming of this method before reading further.

The problem becomes immediately transparent when one considers what happens when $\mathcal{P}'$ pops a stack element (z, A). State z is no longer relevant for $\mathcal{P}'$'s further operation- $\mathcal{P}$ *was* in state z when A got pushed, but what state was $\mathcal{P}$ in when the pop occurred? Therefore, it is necessary to push triples (z, A, z'), where z' is $\mathcal{P}$'s state when the pop takes place. Since it is not known what $\mathcal{P}$'s state z' is going to be when it pops A, $\mathcal{P}$ has to guess what it is going to be, *.i.e.* it nondeterministically pushes (z, A, z'), where $z' \in Z$ is arbitrary. The only restriction is that when executing two (or more) push operations the unknown state z' must be manipulated consistently. This means if A_1A_2 is pushed, then after $\mathcal{P}$ pops A_1, or, equivalently, $\mathcal{P}'$ pops (z_1, A_1, z_1'), then $\mathcal{P}$ finds itself in state z_1'. Since $\mathcal{P}'$ does not use its own state information in imitating $\mathcal{P}$'s state transitions, $\mathcal{P}$'s current state must be available in describing the next element of $\mathcal{P}'$'s stack, or, in other words, $\mathcal{P}$ better be in state z_1' when popping A_2 from its stack, and so must be of the form (z_1', A_2, z_2) for some (predicted) $z_2 \in Z$. This train of thought will now be formalized.

For simplicity, assume that $\mathcal{P}$ pushes at most two symbols and that it has a single acceptance statei z_F. A moment's reflection shows that these assumptions are not restrictive; but they do eliminate some extra preprocessing. The nonterminals of G are triples (z, A, z') of states z, z' and a stack symbol A. The basic idea is to imitate what

the machine undergoes in state z finally to pop symbol A and to wind up thereby in state z', having processed some string of input characters. Thus the rules for the sought-after context-free grammar are posited as follows:

1. For the (extra) start symbol put $S \Rightarrow (z_A, S_A, z_F)$.
2. For each transition $((a, z, B),(z', C)) \in R$ put for each $z_1 \in Z$

$$(z, B, z_1) \Rightarrow a(z', C, z_1)$$

3. In case two symbols are pushed, *i.e.* $((a, z, B),(z', C_1C_2)) \in R$, then put for each pair $z_1, z_2 \in Z$

$$(z, B, z_1) \Rightarrow a(z', C_1, z_2)(z_2, C_2, z_1).$$

4. For each $z \in Z$ put $(z, \lambda, z) \Rightarrow \lambda$.

It is important to notice the free choice of z_1 and z_1, z_2 in rules 2. and 3. Consider, for example, processing the string a^2bc^2 from the PDA from Section 5.1. Then posit the start rule

$$S \Rightarrow (z_1, S_A, z_3),$$

since there is only one final state. Now mechanically translate each of the transitions from this PDA into their grammatical equivalents as shown in Table 5.1.

Table 5.1: Translation of the PDA $\mathcal{P}$ Transition Rules into Grammatical Productions

Nr.	Transition Function	Nr	Production
1	$((a, z_A, S_A),(z_A, SS_A))$	1'	$(z_A, S_A, z') \Rightarrow a(z_A, S, z'')(z'', S_A, z')$
2	$((a, z_A, S),(z_A, SS))$	2'	$(z_A, S, w') \Rightarrow a(z_A, s, w'')(w'', s, w')$
3	$((b, z_A, S),(z_2, \lambda))$	3'	$(z_A, S, v') \Rightarrow b(z_2, \lambda, v')$
4	$((c, z_2, S),(z_2, \lambda))$	4'	$(z_2, S, u') \Rightarrow c(z_2, \lambda, u')$
5	$((c, z_2, S_A),(z_3, \lambda))$	5'	$z_2, S_A, t') \Rightarrow c(z_3, \lambda, t')$

It is important to note that states z', z'', w', w'', v', u', t' can be chosen at will. Hopefully, a proper choice will lead to success in accordance with the philosophy of nondeterminism.

Properties of Context-Free Langauges

Syntax Trees

Tree representations of derivations, also known as syntax trees, were briefly introduced in the preceding section to promote intuition of derivations. Since these are such important tools for the investigation of context-free languages, they will be dealt with a little more systematically here.

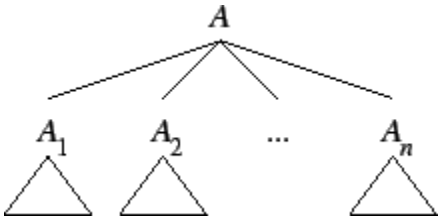
Definition

Let $\mathcal{G}$ = (X, T, R, S) be a context-free grammar. A **syntax tree** for this grammar consists of one of the following

1. A single node x for an $x \in T$. This x is both root and leaf node.
2. An edge

corresponding to a production $A \Rightarrow \alpha \in R$.

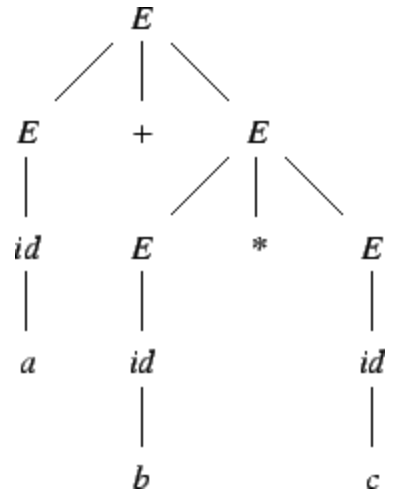
3. A tree



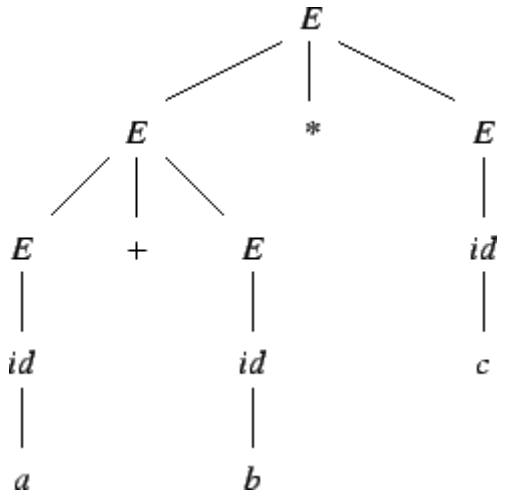
where the A_1, A_2, ⋯ , A_n are the root nodes of syntax trees. Their yields are read from left to right.

Ambiguity

Until now the syntax trees were uniquely determined-even if the sequence of direct derivations were not. Separating the productions corresponding to the operator hierarchy, from weakest to strongest, in the expression grammar + , - ,*,/,() preserves this natural hierarchy. If this is not done, then syntax trees with a false evualation sequence are often the result. Suppose, for instance, that the rules of the expression grammar were written $E \Rightarrow E + E \mid E^*E \mid id$, then two *different* syntax trees are the result. If the first production $E \Rightarrow E + E$ were chosen then the result would be the tree



On the other hand, choosing the production $E \Rightarrow E*E$ first results in a syntax tree of an entirely different ilk.



Thus this grammar is *ambiguous*, because it is possible to generate two different syntax trees for the expression $a + b*c$.

Chomsky Normal Form

Work with a given context-free grammar is greatly facilitated by putting it into a so-called *normal form*. This provides some kind of regularity in the appearance of the right-hand sides of grammar rules. One of the most important normal forms is the *Chomsky normal form*.

Definition

The context-free Grammar $\mathcal{G} = (X, T, R, S)$ is said to be in **Chomsky normal form** if all grammar rules have the form

$$A \Rightarrow a \mid BC, \tag{5.1}$$

for $a \in T$ and $B, C \in X - T$. There is one exception. If $\lambda \in L(\mathcal{G})$, then the single extra rule

$$S \Rightarrow \lambda \tag{5.2}$$

is permitted. If $\lambda \notin L(\mathcal{G})$ then production rule 5.2 is not allowed.

1. $vy \neq \lambda$ (that is, $v \neq \lambda$ or $y \neq \lambda$).
2. The length of vwy satisfies $|vwy| \leq n$.
3. For each integer $k \geq 0$, it follows that $uv^k wy^k z \in L(\mathcal{G})$.

Proof

Assume that $\mathcal{G}$ is in Chomsky normal form. For $x \in L(\mathcal{G})$ consider the (binary) syntax tree for the derivation of x. Assume the height of this tree is h as illustrated in Fig. 5.6.

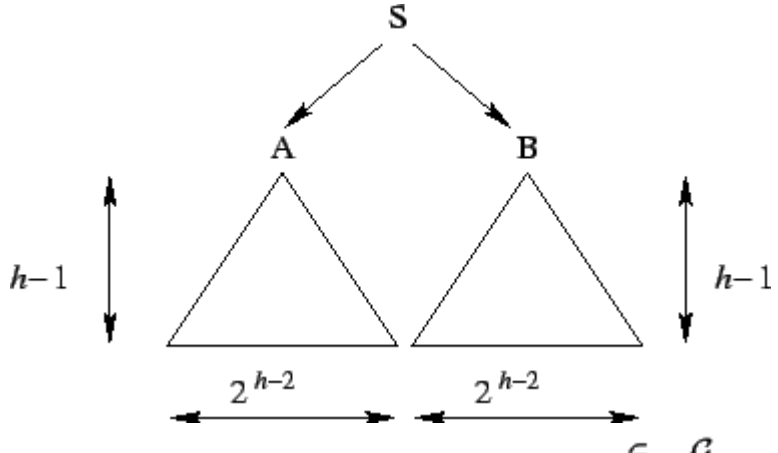


Figure 5.6: Derivation Tree for the string $x \in L(\mathcal{G})$

Then it follows that $|x| \leq 2^{h-2} + 2^{h-2} = 2^{h-1}$, *i.e.* the yield of the tree with height h is at most 2^{h-1}. If $\mathcal{G}$ has k nonterminal symbols, let $n = 2^k$. Then let $x \in L(\mathcal{G})$ be a string with $|x| \geq n$. Thus the syntax tree for x has height at least $k + 1$, thus on the path from the root downwards that defines the height of the tree there are at least $k + 2$ nodes, *i.e.* at

least $k + 1$ nonterminal symbols. It then follows that there is some nonterminal symbol A that appears at least twice. Consulting Fig. 5.7, it is seen that the partial derivation $S \Rightarrow^* uAz \Rightarrow^* uvAyz$ obtains.

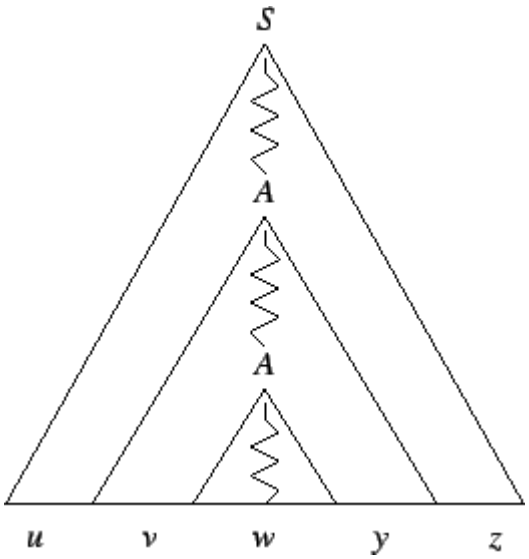


Figure 5.7: Nonterminal A appears twice in the derivation of x

If, now, both u and z were empty, then derivations of the form $S \Rightarrow uAz \Rightarrow A$ would be possible, contrary to the assumption of Chomsky normal form. For the same reason either v or y are nonempty. If $|\ vwy\ | > n$ then apply the procedure anew until the condition $|\ vwy\ | \leq n$ holds. Finally, since the derivation $A \Rightarrow vAy$ can be repeated as often as one pleases, it follows that $S \Rightarrow^* uAz \Rightarrow^* uvAyz \Rightarrow^* uv^2Ay^2z \Rightarrow uv^2wy^2z$i, *etc.* can be generated. This completes the proof.

Example 1

The language $L = \{a^i b^i c^i \mid i \geq 1\}$ is not context free.

Proof

Assume L were context-free. Then let n be the n from the preceding theorem and put $x = a^n b^n c^n$. Ogden's lemma then provides the decomposition $x = uvwyz$ with the stated properties. There are several cases to consider.

Case 1 The string vy contains only a's. But then the string $uwz \in L$, which is impossible, because it contains fewer a's than b's and c's. .
Case 2,3 vy contains only b's or c's. This case is similar to case 1.
Case 4,5 vy contains only a's and b's or only b's or c's. Then it follows that uwz contains

more c's than a's and b's or more a's than b's and c's. This is again a contradiction.

Since $|\ vwy\ | \leq n$ it is not possible that vy contain a's and c's.

it is seen that the complements $\overline{L_1}$ and $\overline{L_2}$ are not in general context-free.

Push Down Automata and Context-Free Grammars

Definition

An algorithm is called **polynomial** in case there is an integer $k \geq 2$ such that the number of steps after which the algorithm halts is $\mathcal{O}(n^k)$. The argument n depends only on the input.

Theorem 5..7 There is a polynomial algorithm that constructs to any given push down automaton $\mathcal{P}$ a context-free grammar $\mathcal{G}$ with $L(\mathcal{G}) = L(\mathcal{A})$. Conversely, there is a polynomial algorithm that constructs to any given context-free grammar $\mathcal{G}$ a push down automaton $\mathcal{P}$ with $L(\mathcal{P}) = L(\mathcal{G})$.

Theorem 5..8 There is a polynomial algorithm that decides, given any context-free grammar $G = (X, T, R, S)$ and $x \in T^*$ whether $x \in L(\mathcal{G})$.

Proof

The proof of this theorem sometimes goes under the name *CYK algorithm* after their discoverers Cocke, Younger and Kasami. It proceeds as follows:

1. Rewrite $\mathcal{G}$ in Chomsky normal form. It is easily seen that this can be done in polynomial time.
2. If $x = x_1x_2 \cdots x_n$, then for $0 \leq i, j \leq n$ put $x_{ij} = x_ix_{i+1} \cdots x_{i+j-1}$. It is noteworthy that $|\ x_{ij}\ | = j$. The idea is to determine all $A \in X - T$ for which $A \Rightarrow^* x_{ij}$. Thus set

$$V_{ij} = \{A \in X - T \mid A \Rightarrow x_{ij}\}.$$

   1. For $j = 1$ it is readily seen that $V_{i1} = \{A \in X - T \mid A \Rightarrow x_i\}$.
   2. For general j it is also seen that $A \in V_{ij} \iff A \Rightarrow^* x_ix_{i+1} \cdots x_{i+j-1} \iff A \Rightarrow BC$ is a rule from R and $B \Rightarrow^* x_i \cdots x_{i+k-1}$ and $C \Rightarrow^* x_{i+k} \cdots x_{i+j-1}$ for some $k = 1, 2, \cdots j - 1$.

Thus the algorithm can be formulated as follows:

for $i := 1$ *to* n *do*

$V_{i1} := \{A \in X - T \mid A \Rightarrow x_i \in R\}$;

for $j := 2$ *to* n *do*

for $i := 1$ *to* $n - j + 1$ *do begin*

$V_{ij} := \emptyset$;

for $k := 1$ *to* $j - 1$ *do*

$V_{ij} := V_{ij} \cup \{A \in X - T \mid A \Rightarrow BC,\ B \in V_{ik},\ C \in V_{i+k,\, j-k}\}$;

end

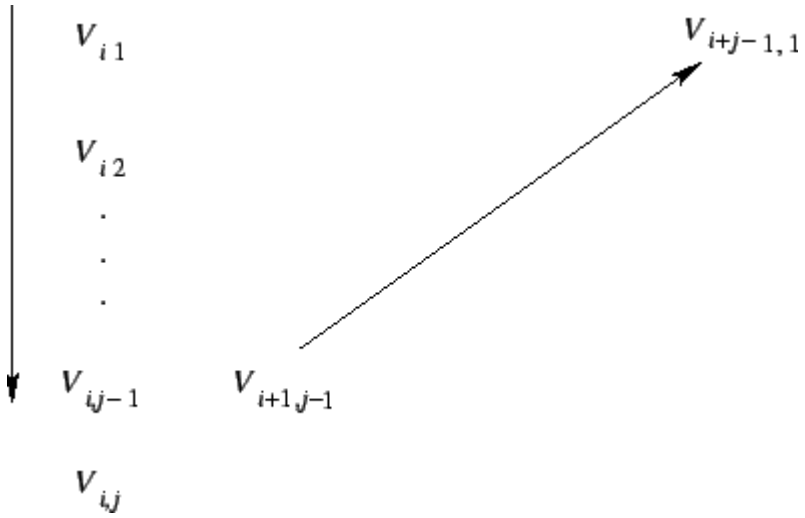


Figure 5.8: Diagonal Procedure for CYK Algorithm

There is a nice interpretation of the innermost *for* loop. Formally one processes the pairs $V_{i1}V_{i+1,\,j-1}$, $V_{i2}V_{i+2,\,j-2}$, $\cdots$, $V_{i,\,j-1}V_{i+j-1,\,1}$. As evidenced in Fig. 5.8 go down the ith column and simultaneously traverse the diagonal from $V_{i+1,\,j-1}$ up and to the right. The corresponding elements are compared with each other.

Finally, it is seen that $x \in L(G) \iff S \in V_{1,\,n}$, because then $S \Rightarrow x_1 \cdots x_n$, where $n =$ length(x).

This technique of producing increasingly larger solutions from smaller ones is called *dynamic programming*.

Example

Consider the Grammar

$S \Rightarrow AB \mid BC$

$A \Rightarrow BA \mid a$

$B \Rightarrow CC \mid b$

$C \Rightarrow AB \mid a$

and the string $x = baaba$ with $n = 5$. Then proceeding as above, the following triangular matrix results:

b	a	a	b	a
B	A, C	A, C	B	A, C
S, A	B	S, C	S, A	
$\emptyset$	B	B		
$\emptyset$	S, A, C			
S, A, C				

Since $S \in V_{15}$ it follows that $x \in L(G)$. It is quite remarkable that the algorithm time is $\mathcal{O}(n^3)$. It is also remarkable that the CYK algorithm actually shows how to construct the derivation, which has great practical importance.

Then it is easy to derive the string abc:

$S \Rightarrow aBC \Rightarrow abC \Rightarrow abc$

Similarly, one derives the string $a^2b^2c^2$:

$S \Rightarrow aSBC \Rightarrow a^2BCBC$

$\Rightarrow a^2B^2C^2 \Rightarrow a^2bBC^2$

$\Rightarrow a^2b^2C^2 \Rightarrow a^2b^2cC \Rightarrow a^2b^2c^2$

$\Rightarrow a^2b^2C^2 \Rightarrow a^2b^2cC \Rightarrow a^2b^2c^2$

It is then a routine application of mathematical induction to prove the general formula $S \Rightarrow a^n b^n c^n$.

$$\Rightarrow E + b*c \Rightarrow T + b*c \Rightarrow F + b*c \Rightarrow id + b*c \Rightarrow a + b*c.$$

At each stage of the derivation the sentential form of the stage is of the form uv, where $u \in X^*$ and $v \in T^*$. Tracing this derivation backwards, now proceed as follows: Starting from the leftmost input symbol reduce that symbol to a rule for which it is the right-hand side, in this case $id \Rightarrow a$. Then reduce id to F, *etc.* until an E has been produced. All of the previous symbols are *handles* or right-hand sides of rules that allow successful (in the sense that the start symbol will eventually be produced). After E has been obtained, the next input symbol `+' is kept, or better, appended to $E$. Thus the sentential form `E +' is produced. This sentential form is called a *viable prefix* because there is a rule of the form $E \Rightarrow E + T$ (a trivial one). If it recognized that E + is a viable prefix, then, starting with the next input symbol, continue this process from that point onwards until the rest of the right-hand side has been produced, *i.e.* a handle has been found. Then reduce this handle to the left-hand side of the ``correct" rule until the start symbol alone has been produced. This process can be nicely realized using a push-down automaton. Thus, proceeding from left to right on the input string, *shift* or push one or more input symbols onto the stack until a handle is found. The *reduce* or pop that handle from the stack and push the left-hand side of the associated rule onto the stack. On a successful parse, if no reduction is presently forthcoming then the contents of the stack constitute a viable prefix for some rule yet to be determined. Another way of saying the same thing is that the contents of the stack, read from bottom up, are the prefix of a sentential form produced on the way back to the start symbol during a rightmost derivation.

A correct parse of the string $a + b*c$ as a sequence of shift/reduce actions is given in Table 5.3. Notice the decision to handle multiplication before addition is governed by ``looking ahead" one symbol.

Table 5.3: Predictive Parse of the expression $a + b*c$

Stack	**Input**	**Action**
\$	$a + b*c$\$	Shift
id\$	$+ b*c$\$	Reduce
F\$	$+ b*c$\$	Reduce
T\$	$+ b*c$\$	Reduce
E\$	$+ b*c$\$	Reduce

+ E\$	b*c\$	Shift
b + E\$	*c\$	Shift
id + E\$	*c\$	Reduce
F + E\$	*c\$	Reduce
T + E\$	*c\$	Reduce
*T + E\$	c\$	Shift
c*T + E\$	\$	Reduce
id*T + E\$	\$	Reduce
F*T + E\$	\$	Reduce
T + E\$	\$	Reduce
E\$	\$	Accept

Stack	**Input**	**Action**
\$	a + b*c\$	Shift
id\$	+ b*c\$	Reduce
F\$	+ b*c\$	Reduce
T\$	+ b*c\$	Reduce
E\$	+ b*c\$	Reduce
+ E\$	b*c\$	Shift
b + E\$	*c\$	Shift
id + E\$	*c\$	Reduce
F + E\$	*c\$	Reduce
T + E\$	*c\$	Reduce
*T + E\$	c\$	Shift
c*T + E\$	\$	Reduce
id*T + E\$	\$	Reduce
F*T + E\$	\$	Reduce
T + E\$	\$	Reduce
E\$	\$	Accept

UNIT-IV

PROPERTIES OF CONTEXT FREE LANGUAGES

Turing Machines (TM)

- Structure of Turing machines
- Deterministic Turing machines (DTM)
  - Accepting a language
  - Computing a function
- Composite Turing machines
- Multitape Turing machines
- Nondeterministic Turing machines (NTM)
- Universal Turing machines (UTM)

- Determine if an input x is in a Determine if an input x is in a language.
  - That is, answer if the answer of a problem P for the instance x is “yes”.
- Compute a function
  - Given an input x, what is f(x)?
  - language.
  - That is, answer if the answer of a problem P for the instance x is “yes”.
- Compute a function
  - Given an input x, what is f(x)?

How does a TM work?

- At the beginning,
  - A TM is in the *start state* (*initial state*)
  - its tape head points at the first cell
  - The tape contains Δ, following by input string, and the rest of the tape contains Δ.
- For each move, a TM
  - reads the symbol under its tape head

- According to the *transition function* on the symbol read from the tape and its current state, the TM:
  - write a symbol on the tape
  - move its tape head to the left or right one cell or not
  - changes its state to the *next state*

When does a TM stop working?

- A TM stops working,
  - when it gets into the special state called halt state. (halts)
    - The output of the TM is on the tape.
  - when the tape head is on the leftmost cell and is moved to the left. (hangs)
  - when there is no *next state*. (hangs)

How to define deterministic TM (DTM)

- a quintuple $(Q, \Sigma, \Gamma, \delta, s)$, where
  - the set of states Q is finite, not containing halt state h,
  - the input alphabet Σ is a finite set of symbols not including the blank symbol Δ,
  - the tape alphabet Γ is a finite set of symbols containing Σ, but not including the blank symbol Δ,
  - the start state s is in Q, and
  - the transition function δ is a partial function from $Q \times (\cup\Gamma\{\Delta\}) \rightarrow Q \cup \{h\} \times (\cup\Gamma\{\Delta\}) \times \{L, R, S\}$.

Example of a DTM

Definition

- Let $T = (Q, \Sigma, \Gamma, \delta, s)$ be a DTM, and $(q1, \alpha 1\underline{a1}\beta 1)$ and $(q2, \alpha 2\underline{a2}\beta 2)$ be two configurations of T.

We say $(q1, \alpha 1\underline{a1}\beta 1)$ yields $(q2, \alpha 2\underline{a2}\beta 2)$ in one step, denoted by $(q1, \alpha 1\underline{a1}\beta 1) \vdash_T (q2, \alpha 2\underline{a2}\beta 2)$, if

- $\delta(q1, a1) = (q2, a2, S)$, $\alpha 1 = \alpha 2$ and $\beta 1 = \beta 2$,
- $\delta(q1, a1) = (q2, b, R)$, $\alpha 2 = \alpha 1 b$ and $\beta 1 = a2\beta 2$,

– $\delta(q1, a1) = (q2,b,L)$, $\alpha 1 = \alpha 2 a2$ and $\beta 2 = b$

Definition

- Let $T=(Q, \Sigma, \Gamma, \delta, s)$ be a DTM, and $(q1, \alpha 1 \underline{a1} \beta 1)$ and $(q2, \alpha 2 \underline{a2} \beta 2)$ be two configurations of T.

We say $(q1, \alpha 1 \underline{a1} \beta 1)$ yields $(q2, \alpha 1 \underline{a2} \beta 2)$ in zero step or more, denoted by $(q1, \alpha 1 \underline{a1} \beta 1) \vdash^* T (q2, \alpha 1 \underline{a2} \beta 2)$, if

– $q1=q2$, $\alpha 1 = \alpha 2$, $a1 = a2$, and $\beta 1 = \beta 2$, or
– $(q1, \alpha 1 \underline{a1} \beta 1) \vdash T (q, \alpha \underline{a} \beta)$ and $(q, \alpha \underline{a} \beta) \vdash^* T (q2, \alpha 1 \underline{a2} \beta 2)$ for some q in Q, α and β in Γ^*, and a in Γ.

Yield in zero step or more: Example

s,$\underline{\Delta}$0001000)
(p1,@$\underline{0}$001000)
(p2,@$\Delta\underline{0}$01000)
(p2,@Δ001000$\underline{\Delta}$)
(p3,@Δ00100$\underline{0}$)
(p4,@Δ0010$\underline{0}\Delta$)
(p4,@$\underline{\Delta}$00100Δ)
(p1,@$\Delta\underline{0}$0100Δ)
(p2,@$\Delta\Delta\underline{0}100\Delta$) (p4,@$\Delta\Delta$01$\underline{0}$)
(p4,@$\Delta\underline{\Delta}$010)
(p1,@$\Delta\Delta\underline{0}$10)
(p2,@$\Delta\Delta\Delta\underline{1}$0)
(p2,@$\Delta\Delta\Delta$1$\underline{0}$)
(p2,@$\Delta\Delta\Delta$10$\underline{\Delta}$)
(p3,@$\Delta\Delta\Delta$1$\underline{0}$)

(p4,@ΔΔΔ1)
(p4,@ΔΔΔ1)
(p1,@ΔΔΔ1)
(q1,@ΔΔΔΔΔ)
(q1,@)
(q2,ΔΔ)
(h ,Δ1)

(p2,@ΔΔ0100Δ)
(p3,@ΔΔ0100)

TM accepting a language

- Definition

Let $T=(Q, \Sigma, \Gamma, \delta, s)$ be a TM, and $w\Sigma\in*$.
T accepts w if $(s, \varepsilon, \Delta, w)$ |-T* $(h, \varepsilon, \Delta, 1)$.
The language accepted by a TM T, denoted by $L(T)$, is the set of strings accepted by T.

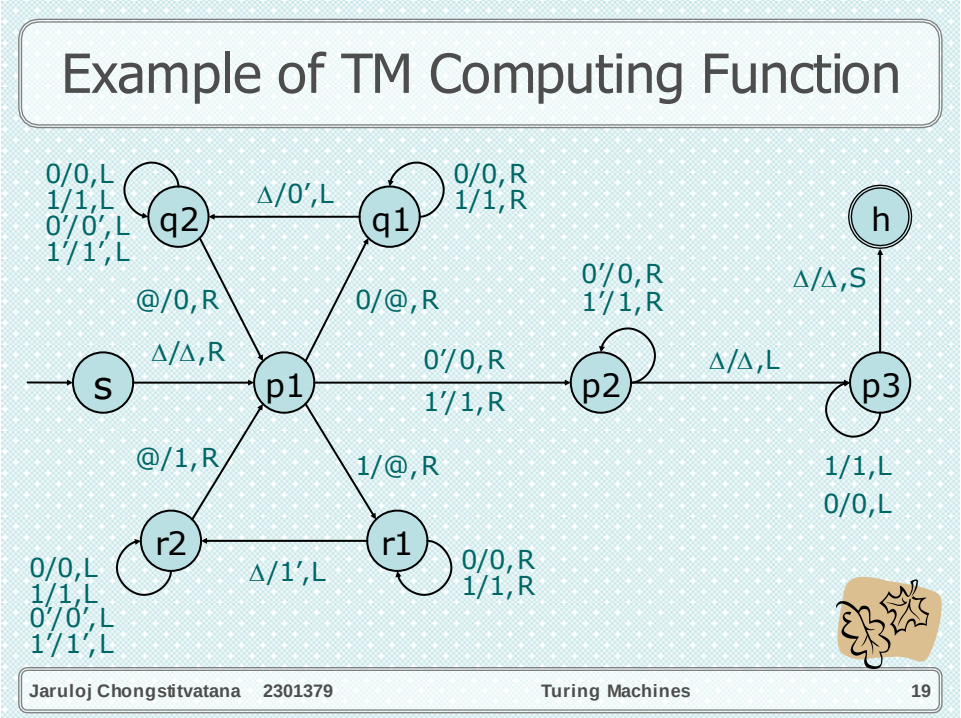
$L(T)=\{0n10n \mid n\geq 0\}$

- T halts on $0n10n$
- T hangs on $0n+110n$ at p3
- T hangs on $0n10n+1$ at q1
- T hangs on $0n\ 12\ 0n$ at q1

TM computing a function

- Definition

Let $T=(Q, \Sigma, \Gamma, \delta, s)$ be a TM, and f be a function from Σ^* to Γ^*. *T* computes *f* if, for any string *w* in Σ^*,



- Let *T1* and *T2* be TM's.
- *T1* → *T2* means executing *T1* until *T1* halts and then executing *T2*.
- *T1* -*a*→ *T2* means executing *T1* until *T1* halts and if the symbol under the tape head when *T1* halts is a then executing *T2*.

Nondeterministic TM

- An NTM starts working and stops working in the same way as a DTM.
- Each move of an NTM can be nondeterministic.

Each Move in an NTM

- reads the symbol under its tape head
-
- According to the *transition relation* on the symbol read from the tape and its current state, the TM choose one move nondeterministically to:
  - write a symbol on the tape
  - move its tape head to the left or right one cell or not
  - changes its state to the *next state*

How to define nondeterministic TM (NTM)

- a quintuple $(Q, \Sigma, \Gamma, \delta, s)$, where
  - the set of states Q is finite, and does not contain halt state h,
  - the input alphabet Σ is a finite set of symbols, not including the blank symbol Δ,
  - the tape alphabet Γ is a finite set of symbols containing Σ, but not including the blank symbol Δ,
  - the start state s is in Q, and
  - the transition fn $\delta: Q \times (\cup \Gamma\{\Delta\}) \rightarrow 2^{Q \cup \{h\}} \times (\cup \Gamma\{\Delta\}) \times \{L,R,S\}$.

Configuration of an NTM

Definition

- Let $T = (Q, \Sigma, \Gamma, \delta, s)$ be an TM.

A configuration of T is an element of $Q \times \Gamma* \times \Gamma \times \Gamma*$

- Can be written as

- (q,l,a,r) or
- $(q,l,\underline{a},r)$ Definition
- Let $T = (Q, \Sigma, \Gamma, \delta, s)$ be an NTM, and $(q1, \alpha 1 \underline{a1} \beta 1)$ and $(q2, \alpha 2 \underline{a2} \beta 2)$ be two configurations of T.
- We say $(q1, \alpha 1 \underline{a1} \beta 1)$ yields $(q2, \alpha 2 \underline{a2} \beta 2)$ in one step, denoted by $(q1, \alpha 1 \underline{a1} \beta 1) \vdash T (q2, \alpha 2 \underline{a2} \beta 2)$, if
- $(q2,a2,S) \in \delta(q1, a1)$, $\alpha 1 = \alpha 2$ and $\beta 1 = \beta 2$,
- $(q2,b,R) \in \delta(q1, a1)$, $\alpha 2 = \alpha 1 b$ and $\beta 1 = a2 \beta 2$,
- $(q2,b,L) \in \delta(q1, a1)$, $\alpha 1 = \alpha 2 a2$ and $\beta 2 = b \beta 1$.
-

NTM accepting a language/computing a function

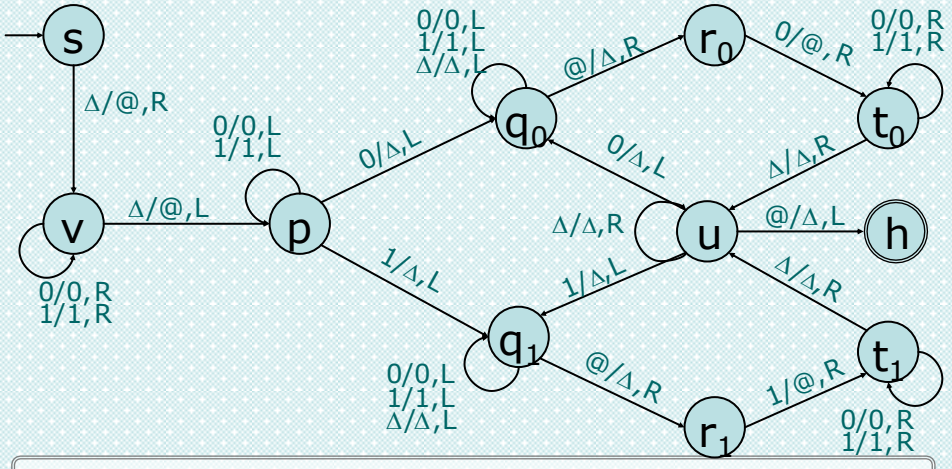
- Definition

Let $T = (Q, \Sigma, \Gamma, \delta, s)$ be an NTM.
Let $w \Sigma \in *$ and f be a function from Σ^* to Γ^*.
T accepts w if $(s, \varepsilon, \Delta, w) \vdash T^* (h, \varepsilon, \Delta, 1)$.
The language accepted by a TM T, denoted by $L(T)$, is the set of strings accepted by T.

T computes f if, for any string w in Σ^*, $(s, \varepsilon, \Delta, w) \vdash T^* (h, \varepsilon, \Delta, f(w))$.

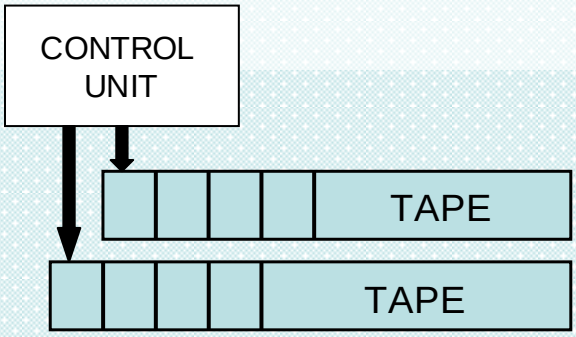
Example of NTM

- Let $L=\{ww \mid w \in \{0,1\}^*\}$



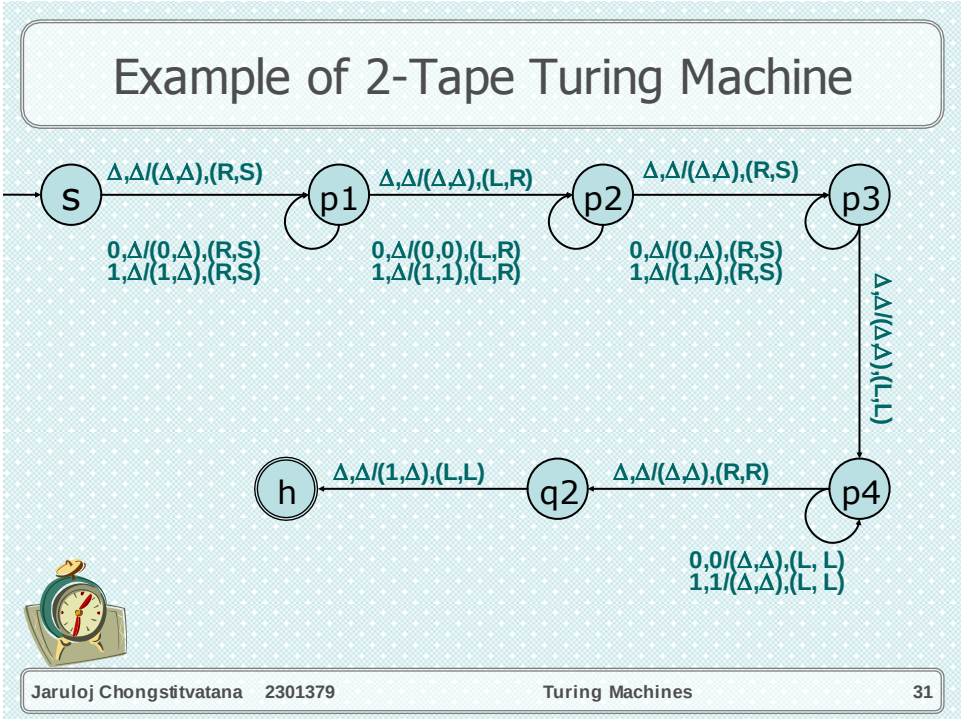
Multitape TM

- TM with more than one tape.
- Each tape has its own tape head.
- Each tape is independent.



2-Tape Turing Machine

- a quintuple $(Q, \Sigma, \Gamma, \delta, s)$, where
  - the set of states Q is finite, and does not contain the halt state h,
  - the input alphabet Σ is a finite set of symbols, not including the blank symbol Δ,
  - the tape alphabet Γ is a finite set of symbols containing Σ, but not including the blank symbol Δ,
  - the start state s is in Q, and
  - the transition function δ is a partial function from $Q \times (\cup\Gamma\{\Delta\})2 \rightarrow Q \cup \{h\} \times (\cup\Gamma\{\Delta\})2 \times \{L, R, S\}2$



Equivalence of 2-tape TM and single-tape TM

Theorem:

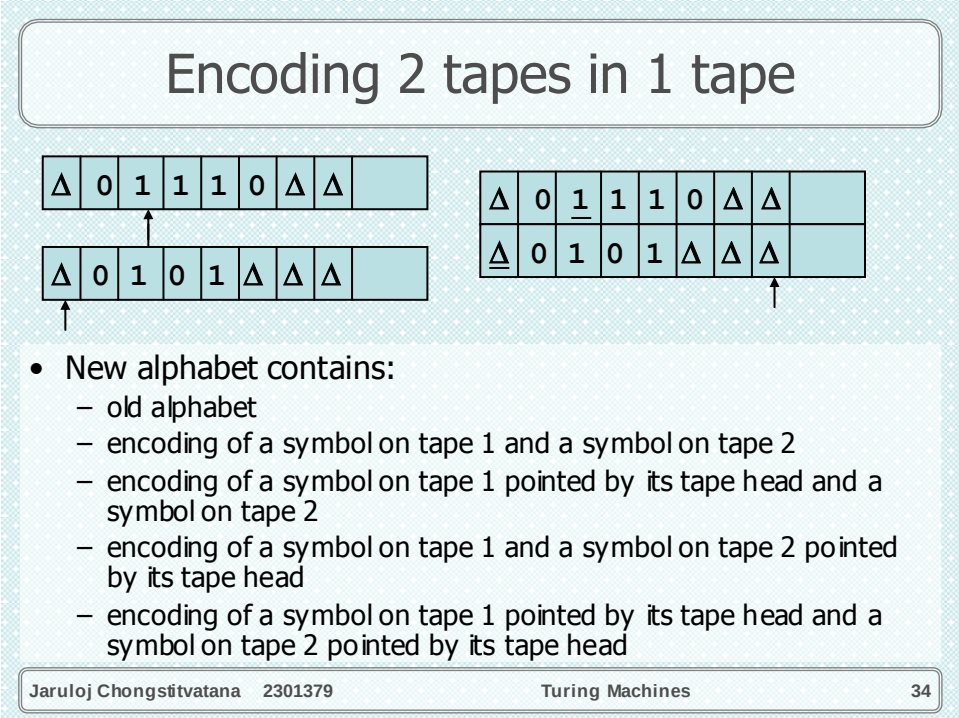
For any 2-tape TM T, there exists a single-tape TM M such that for any string α in Σ^*:

- if T halts on α with β on its tape, then M halts on α with β on its tape, and
- if T does not halt on α, then M does not halt

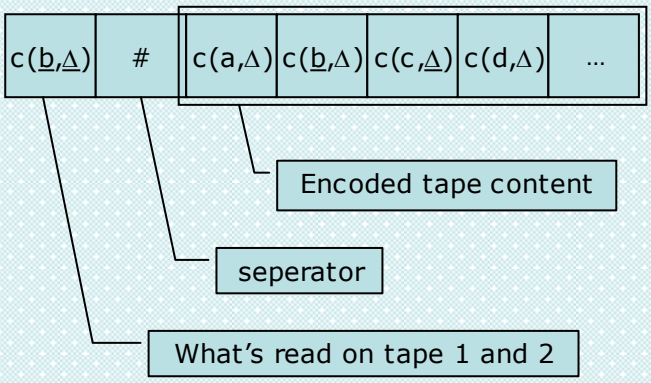
How 1-tape TM simulates 2-tape TM

- Marking the position of each tape head in the content of the tape
- Encode content of 2 tapes on 1 tape
  - When to convert 1-tape symbol into 2-tape symbol

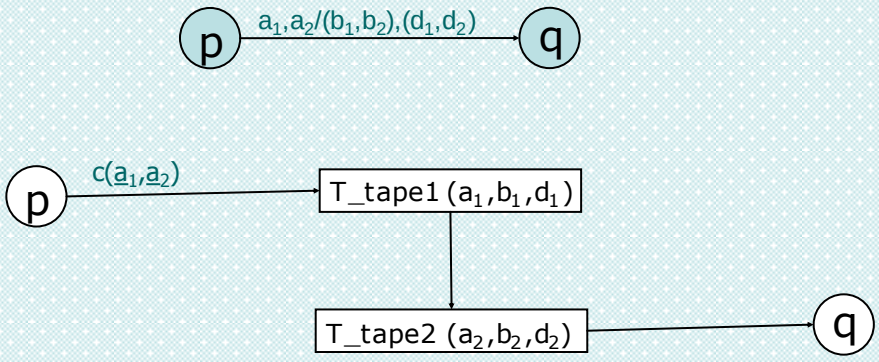
- Construct 1-tape TM simulating a transition in 2-tape TM
- Convert the encoding of 2-tape symbols back to 1-tape symbols

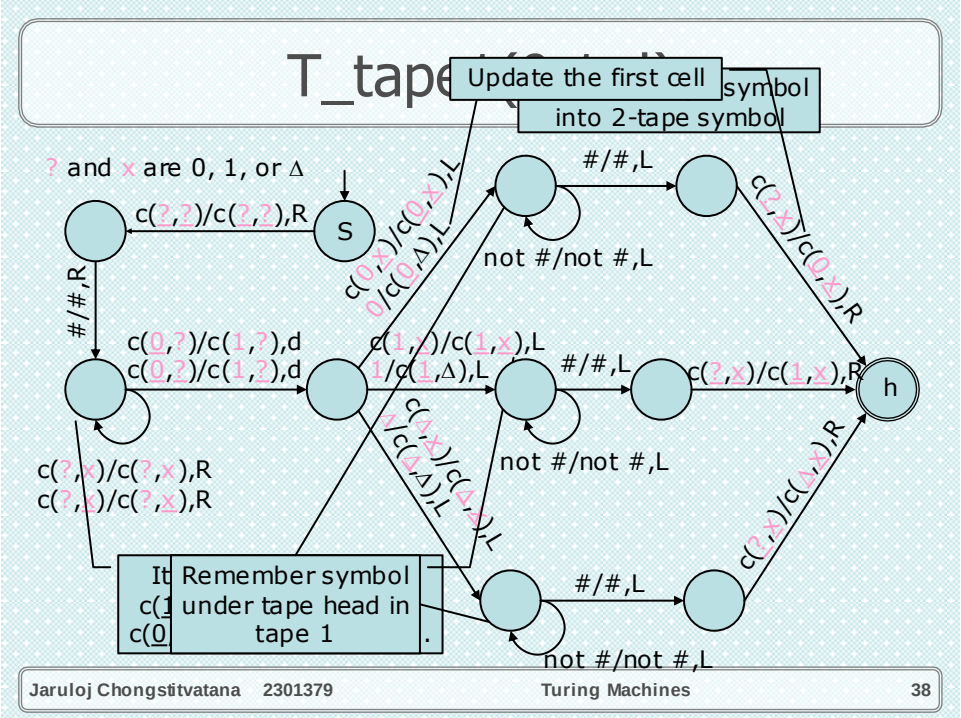


Tape format



Simulating transitions in 2-tape TM in 1-tape TM





Equivalence of 2-tape TM and single-tape TM

Proof:

Let $T = (Q, \Sigma, \Gamma, \delta, s)$ be a 2-tape TM.
We construct a 1-tape TM $M=(K, \Sigma, \Gamma', \delta', s')$ such that

- $\Gamma' = \Gamma \cup \{c(a,b) | a,b$ are in $\cup\Gamma\{\Delta\}\} \cup \{c(\underline{a},b) | a,b$ are in $\cup\Gamma\{\Delta\}\} \cup \{c(a,\underline{b}) | a,b$ are in $\cup\Gamma\{\Delta\}\} \cup \{c(\underline{a},\underline{b}) | a,b$ are in $\cup\Gamma\{\Delta\}\} \cup \{\#\}$

We need to prove that:

- if T halts on α with output β, then M halts on α with output β, and

if T does not halt on α

- If T loops, then M loops.
- If T hangs in a state p, M hangs somewhere from p to the next state.

Equivalence of NTM and DTM

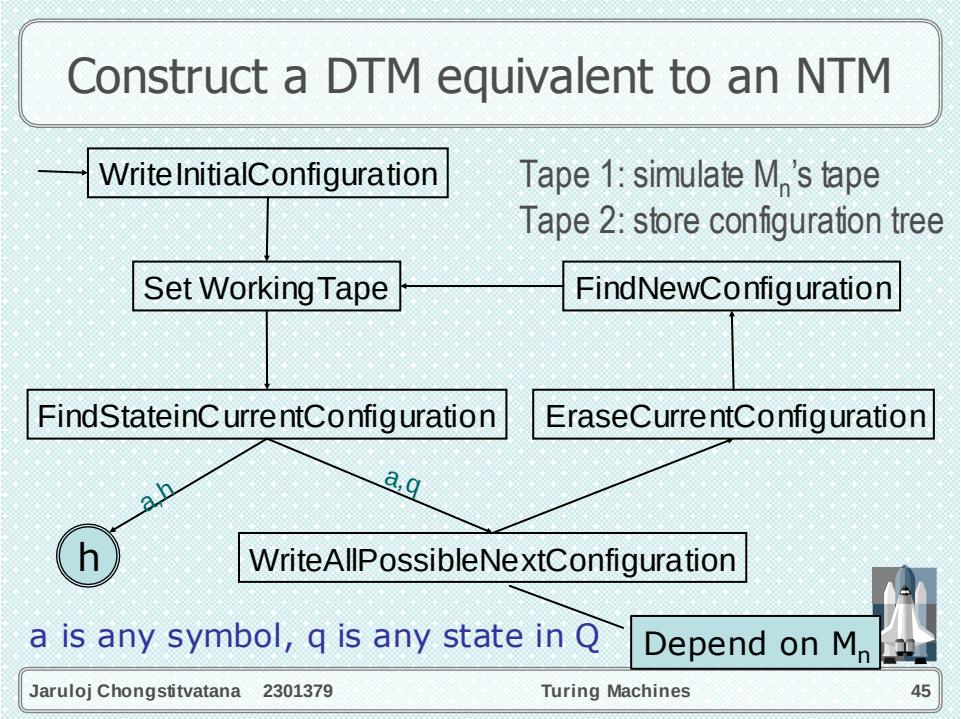
Theorem:

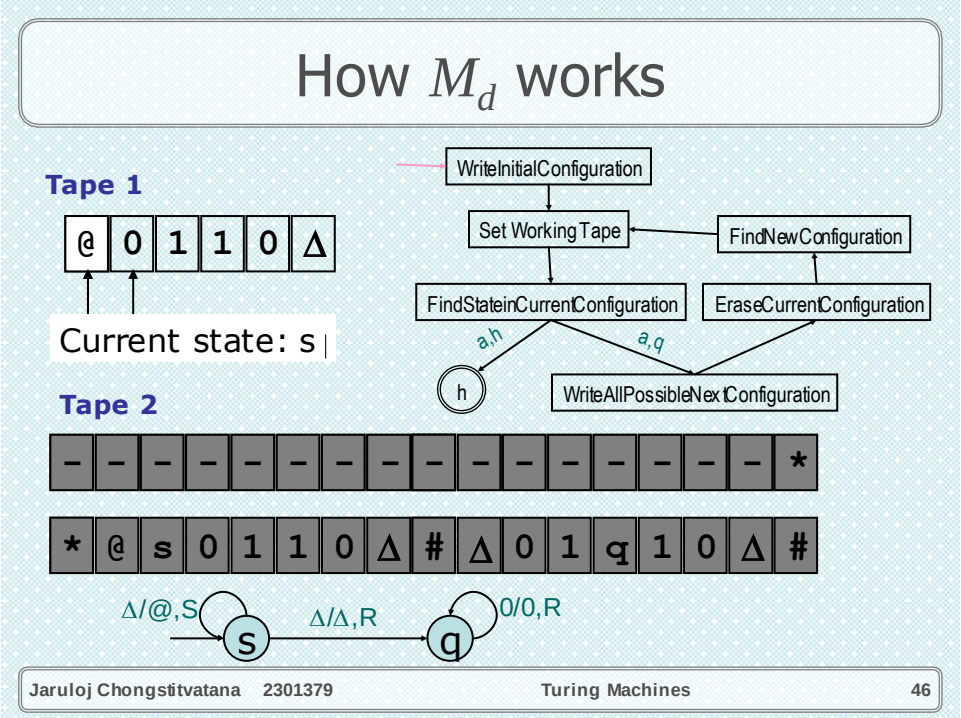
For any NTM *Mn*, there exists a DTM *Md* such that:

- if *Mn* halts on input α with output β, then *Md* halts on input α with output β, and
- if *Mn* does not halt on input α, then *Md* does not halt on input α.

Proof:

Let *Mn* = (Q, Σ, Γ, δ, s) be an NTM.





- Then, there is a positive integer n such that the initial configuration (*s*, $\underline{\Delta}\alpha$) of *Mn* yeilds a halting configuration (*h*, $\underline{\Delta}\beta$) in n steps.
- From the construction of *Md*, the configuration (*h*, $\underline{\Delta}\beta$) must appear on tape 2 at some time.
- Then, *Md* must halt with β on tape 1.

if *Mn* does not halt on input α

- Then, *Mn* cannot reach the halting configuration. That is, (*s*, $\underline{\Delta}\alpha$) never yields a halting configuration (*h*, $\underline{\Delta}\beta$).
- From the construction of *Md*, the configuration (*h*,$\underline{\Delta}\beta$) never appears on tape 2.
- Then, *Md* never halt.

Universal Turing Machine

- Given the description of a DTM *T* and an input string *z*, a universal TM simulates how *T* works on input *z*.
- What's need to be done?

- How to describe *T* and *z* on tape
  - Use an encoding function
- How to simulate *T*

Encoding function

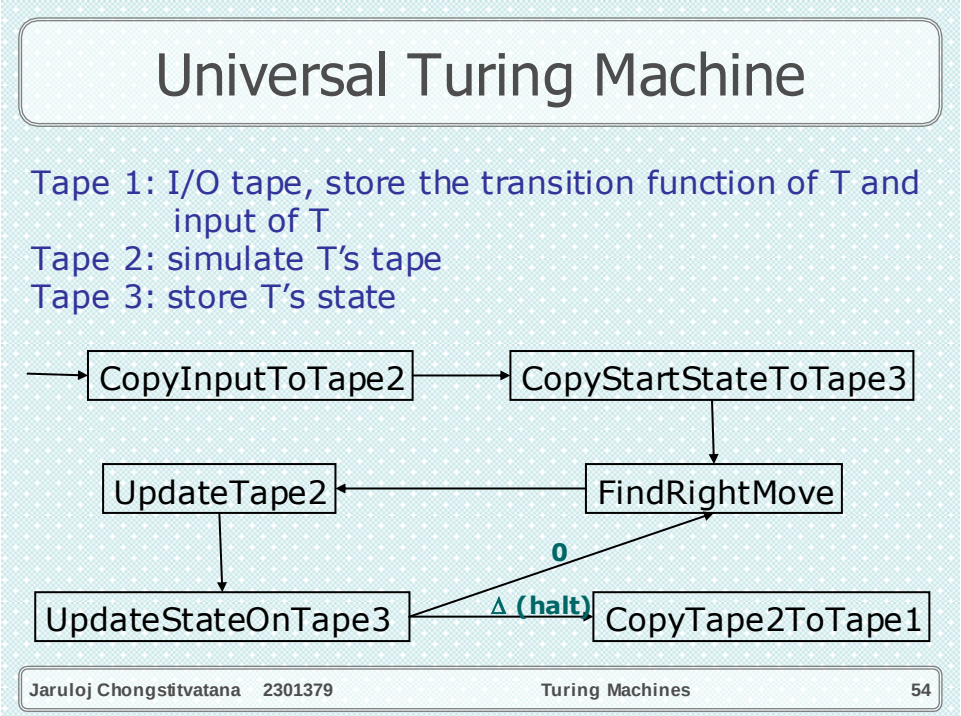
- Let $T=(Q, \Sigma, \delta, s)$ be a TM. The encoding function e(*T*) is defined as follows:
  - e(*T*)=e(*s*)#e(δ),
  - e(δ)=e(*m1*)#e(*m2*)#...#e(*mn*)#, where $\delta = \{m1, m2,..., mn\}$
  - e(*m*)=e(*p*),e(*a*),e(*q*),e(*b*),e(*d*), where *m* = (*p, a, q, b, d*)
  - e(*z*)=1e(*z1*)1e(*z2*)1…1e(*zm*)1, where *z=z1z2…zm* is a string
  - e(Δ)=0, e(*ai*)=0*i*+1, where *ai* is in Σ
  - e(*h*)=0, e(*qi*)=0*i*+1, where *qi* is in *Q*
  - e(S)=0, e(L)=00, e(R)=000

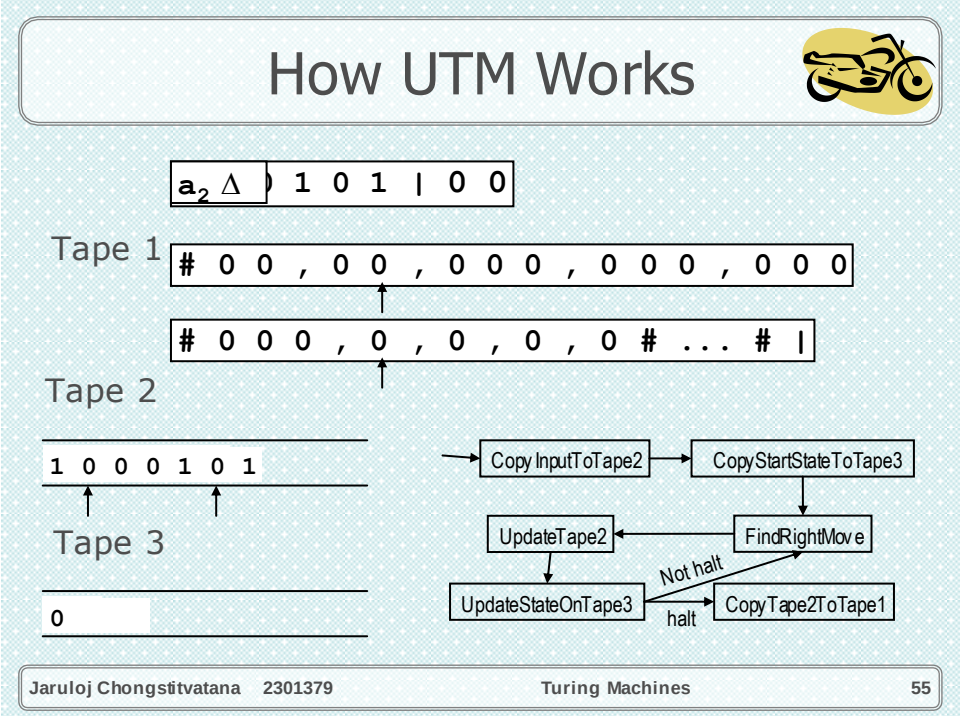
Example of Encoded TM

- e(Δ)=0 , e(a1)=00 , e(a2)=000
- e(h)=0, e(q1)=00, e(q2)=000
- e(S)=0, e(L)=00, e(R)=000
- e(Δa1a1a2Δ) = 1e(Δ)1e(a1)1e(a1)1e(a2)1e(Δ)1
  = 101001001000101
  - e(m1) = (q1),e(a1),e(q2),e(a2),e(R)
    = 00,00,000,000,000
  - e(m2) = e(q2),e(Δ),e(h),e(Δ),e(S)
    = 000,0,0,0,0
  - e(δ) = e(m1)#e(m2)#...#
    = 00,00,000,000,000#000,0,0,0,0#...#
  - e(T) = e(s)#e(δ)
    = 00#00,00,000,000,000#000,0,0,0,0#...#
  - Input = e(Z)|e(T)|

=

101001001000101|00#00,00,000,000,000#000,0,0,0,0#...#|





Church-Turing Thesis

- Turing machines are formal versions of algorithms.
- No computational procedure will be considered an algorithm unless it can be presented as a Turing machine**.**

Checklist

- Construct a DTM, multitape TM, NTM accepting languages or computing function
- Construct composite TM
- Prove properties of languages accepted by specific TM
- Prove the relationship between different types

- Describe the relationship between TM and FA
- Prove the relationship between TM and FA

UNIT -V

Decidability

Decidable/Undecidable problems

Accepting:

Definition

- **Let $T = (Q, \Sigma, \Gamma, \delta, s)$ be a TM.**
- ***T* accepts a string *w* in Σ* if**
  $(s, \underline{\Delta}w) \vdash_T^* (h, \underline{\Delta}1)$.
- ***T* accepts a language $L \subseteq \Sigma^*$ if, for any string *w* in *L*, *T* accepts *w*.**

Characteristic function

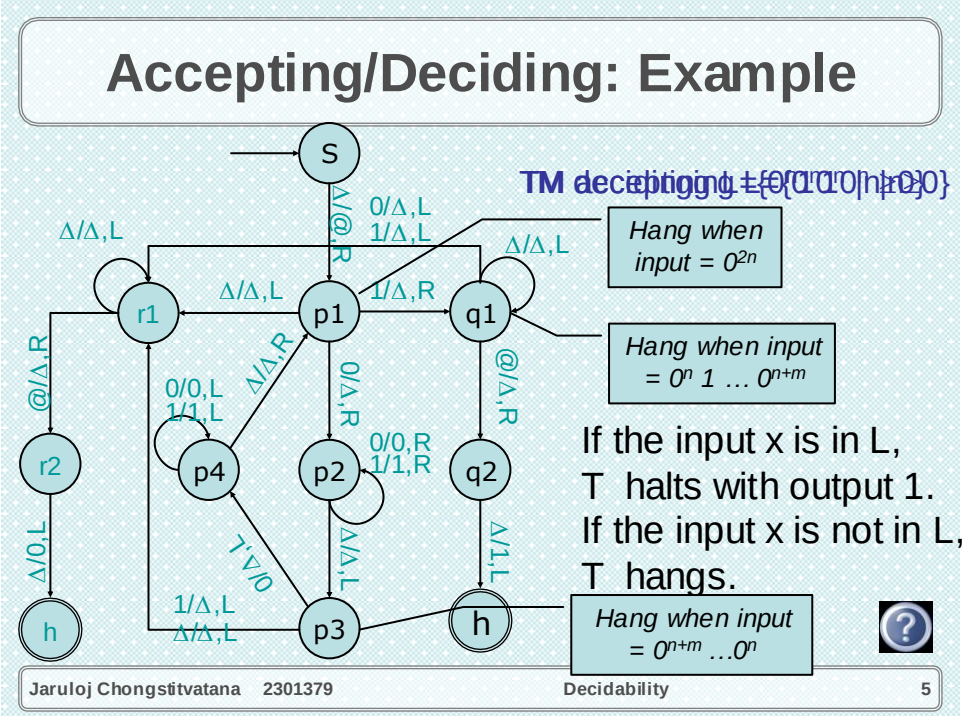
- **For any language $L \subseteq \Sigma^*$, the characteristic function of L is the function $\chi_L(x)$ such that**
  - **$\chi_L(x) = 1$ if $x \in L$**
  - **$\chi_L(x) = 0$ otherwise**
- **Example**

Let $L = \{\omega \in \{0,1\}^* \mid n1(\omega) < n0(\omega) < 2n1(\omega)\}$, where nx(ω) is the number of x's in ω}.

- **$\chi_L(\omega) = 1$ if $n1(\omega) < n0(\omega) < 2n1(\omega)$**
- **$\chi_L(\omega) = 0$ otherwise**

Deciding: Definition

- **Let $T = (Q, \Sigma, \Gamma, \delta, s)$ be a TM.**
- ***T* decides a language $L \subseteq \Sigma^*$ if *T* computes the characteristic function of *L*.**
- ***T* decides a language $L \subseteq \Sigma^*$ if**
  - **for any string *w* in *L*, *T* halts on w with output 1,**
  - **for any string *w* in $\bar{L}$, *T* halts on w with output**



Recursively enumerable languages

- A language L is recursively enumerable if there is a Turing machine T accepting L.
- A language L is Turing-acceptable if there is a Turing machine T accepting L.
- Example:

{0n10n|n≥0} is a recursively-enumerable language.

Recursive languages

- A language L is recursive if there is a Turing machine T deciding L.
- A language L is Turing-decidable if there is a Turing machine T deciding L.
- Example:

$\{0^n10^n | n \geq 0\}$ is a recursive language.

Closure Properties of the Class of Recursive Languages

Theorem:

Let L be a recursive language over Σ. Then, $\bar{L}$ is recursive.
Proof:
Let L be a recursive language over Σ.
Then, there exists a TM T computing χ_L.
Construct a tape TM M computing $\chi_{\bar{L}}$. as follows:

$\rightarrow$ T $\rightarrow$ TmoveRight $\underline{0}\rightarrow$ Twrite1
1 Twrite0

Then, $\bar{L}$ is recursive.

Closure Property Under Union

Theorem: Let L1 and L2 be recursive languages over Σ. Then, L1∪L2 is recursive.
Proof:
Let L1 and L2 be recursive languages over Σ.
Then, there exist TM's T1 and T2 computing χ_{L1} and χ_{L2}, respectively.
Construct a 2-tape TM M as follows:

$\rightarrow$ TcopyTape1ToTape2 $\rightarrow$ T1 $\rightarrow$ TmoveRight $\underline{0}\rightarrow$ TcopyTape2ToTape1 $\rightarrow$ T2

Closure Property Under Union

→ TcopyTape1ToTape2 → T1 → TmoveRight 0→ TcopyTape2ToTape1 → T2

If the input w is not in L1 and L2, χL1(w) and χL2(w)=0. Thus, both T1 and T2 must run, and M halts with output 0.
If the input w is in L1, χL1(w)=1. Thus, M halts with output 1.
If the input w is not in L1 but is in L2, χL1(w)=0 and χL2(w)=1. Thus, M halts with output 1.
That is, M computes characteristic function of χL.
Then, L1∪L2 is recursive.

Closure Property Under Intersection

Theorem: Let L1 and L2 be recursive languages over Σ. Then, L1∩L2 is recursive.
Proof:
Let L1 and L2 be recursive languages over Σ.
Then, there exist TM's T1 and T2 computing χL1 and χL2, respectively.

Construct a 2-tape TM M as follows:

→ TcopyTape1ToTape2 → T1 → TmoveRight 1→ TcopyTape2ToTape1 → T2

TcopyTape1ToTape2 → T1 → TmoveRight 1→ TcopyTape2ToTape1 → T2

If the input w is in L1∩L2, χL1(w) and χL2(w)=1. Thus, M halts with output 1.
If the input w is not in L1, χL1(w)=0. Thus, M halts with output 0.
If the input w is in L1 but is not in L2, χL1(w)=1 and χL2(w)=0. Thus, M halts with output 0.
That is, M computes characteristic function of χL1∩L2.
Then, L1∩L2 is recursive.

Closure Properties of the Class of Recursively Enumerable Languages

Theorem: Let L1 and L2 be recursively enumerable languages over Σ. Then, L1∪L2 is also recursively enumerable.
Proof:
Let L1 and L2 be recursively enumerable languages over Σ.
Then, there exist TM's T1 and T2 accepting L1 and L2, respectively.
Construct an NTM M as follows.

Closure Property Under Union

If w is in L1, but not in L2, then T1 in M runs and halts.
If w is in not L1, but in L2, then T2 in M runs and halts.
If w is in both L1 and L2, then either T1 or T2 runs and halts.
For these 3 cases, M halts.
If w is neither in L1 nor in L2, then either T1 or T2 runs but both never halt. Then, M does not halt.
Thus, M accepts L1∪L2. That is, L1∪L2 is recursively enumerable.

Closure Property Under Intersection

Theorem: Let L1 and L2 be recursively enumerable languages over Σ. Then, L1∩L2 is also recursively enumerable.
Proof:
Let L1 and L2 be recursively enumerable languages over Σ.
Then, there exist TM's T1 and T2 accepting L1 and L2, respectively.
Construct an NTM M as follows.
→ TcopyTape1ToTape2 → T1 → TmoveRight 1→ TcopyTape2ToTape1 → T2

Closure Property Under Intersection

If w is in not L1, then T1 in M does not halt. Then, M does not halt.
If w is in L1, but not in L2, then T1 in M halts and T2 can finally start, but does not halt. Then, M does not halt.
If w is in both L1 and L2, then T1 in M halts and T2 can finally start, and finally halt. Then, M halts.
Thus, M accepts L1∩L2. That is, L1∩L2 is recursively enumerable.

Closure Property Under Intersection (II)

Theorem:

Let L1 and L2 be recursively enumerable languages over Σ. Then, L1∩L2 is also recursively enumerable.
Proof:
Let L1 and L2 be recursively enumerable languages over Σ.
Then, there exist DTM's T1 =(Q1, Σ, Γ, δ1, s1) and T2 =(Q2, Σ, Γ, δ2, s2) accepting L1 and L2, respectively.
Construct a 2-tape TM M which simulates T1 and T2 simultaneously. Tape 1 represents T1's tape and Tape 2 represents T2's tape.

Closure Property Under Intersection (II)

Let $M = ((Q1 \cup \{h\}) \times (Q2 \cup \{h\}), \Sigma, \Gamma, \delta, (s1,s2))$ where

- $\delta((p1,p2),a1,a2) = ((q1,q2),b1,b2,d1,d2)$ for $\delta1(p1,a1)=(q1,b1,d1)$ and $\delta2(p2,a2)=(q2,b2,d2)$
- $\delta((h,p2),a1,a2) = ((h,q2),a1,b2,S,d2)$ for all $p2,a1,a2$ and $\delta2(p2,a2)=(q2,b2,d2)$
- $\delta((p1,h),a1,a2) = ((q1,h),b1,a2,d1,S)$ for all $p1,a1,a2$ and $\delta1(p1,a1)=(q1,b1,d1)$
- $\delta((h,h),a1,a2) = (h,a1,a2,S,S)$ for all $a1,a2$

If neither T1 nor T2 halt, M never gets to the state h.
If T1 halts and T2 does not halt, M gets to the state (h,p).
If T2 halts and T1 does not halt, M gets to the state (p,h).
If both T1 and T2 halt, M finally gets to the state h.

Relationship Between the Classes of Recursively Enumerable and Recursive Languages

Theorem: If L is a recursive language, then L is recursively enumerable.
Proof:
Let L be a recursive language over Σ.
Then, there is a TM T deciding L.
Then, T also accepts L.
Thus, L is recursively enumerable.

Relationship between RE and Recursive Languages

Theorem: Let L be a language. If L and $\bar{L}$ are recursively enumerable, then L is recursive.

Proof:
Let L and $\bar{L}$ be recursively-enumerable languages over Σ.
Then, there are a TM T accepting L, and a TM $\bar{T}$ accepting $\bar{L}$.
For any string w in Σ*, w is either in L or in $\bar{L}$.
That is, either T or $\bar{T}$ must halt on w, for any w in Σ*.
We construct an NTM M as follows:
If w is in L, T halts on w and thus, M accepts w.
If w is not in L, $\bar{T}$ halts on w and thus, M rejects w.
Then, M computes the characteristic function of L. Then, L is recursive.

Decision Problems

- **A decision problem is a prob. whose ans. is either yes or no**
- **A yes-instance (or no-instance) of a problem P is the instance of P whose answer is yes (or no, respectively)**
- **A decision problem P can be encoded by fe over Σ as a language {fe(X)| X is a yes-instance of P}.**

Encoding of decision problems

- Is X a prime ?

{1X | X is a prime}

- Does TM T accept string e(T)?

{e(T) | T is a TM accepting string e(T)}

- Does TM T accept string w?

{e(T)e(w) | T is a TM accepting string w} or
{<T,w> | T is a TM accepting string w}

Decidable (or solvable) problems

Definition:

If fe is a reasonable encoding of a decision problem P over Σ, we say P is decidable (or solvable) if the associated language {fe(X)| X is a yes-instance of P} is recursive.

A problem P is undecidable (or unsolvable) if P is not decidable.

Self-Accepting

- **SA (Self-accepting) = {w∈{0,1,#, ,}*| w=e(T) for some TM T and w∈L(T)}**
- **NSA (Non-self-accepting) = {w∈ {0,1,#, ,}*| w=e(T) for some TM T and w∉L(T)}**
- **E (Encoded-TM) = {w∈{0,1,#, ,}*| w=e(T) for some TM T}**

NSA is not recursively enumerable

We prove by contradiction.
Assume NSA is recursively enumerable.
Then, there is TM T0 such that L(T0)=NSA.
Is e(T0) in NSA?

- **If e(T0)∈NSA, then e(T0)∉L(T0) by the definition of NSA But L(T0)=NSA. Thus, contradiction.**
- **If e(T0) ∉NSA, then e(T0) ∈SA and e(T0)∈L(T0) by the definition of SA. But L(T0)=NSA. Thus, contradiction.**

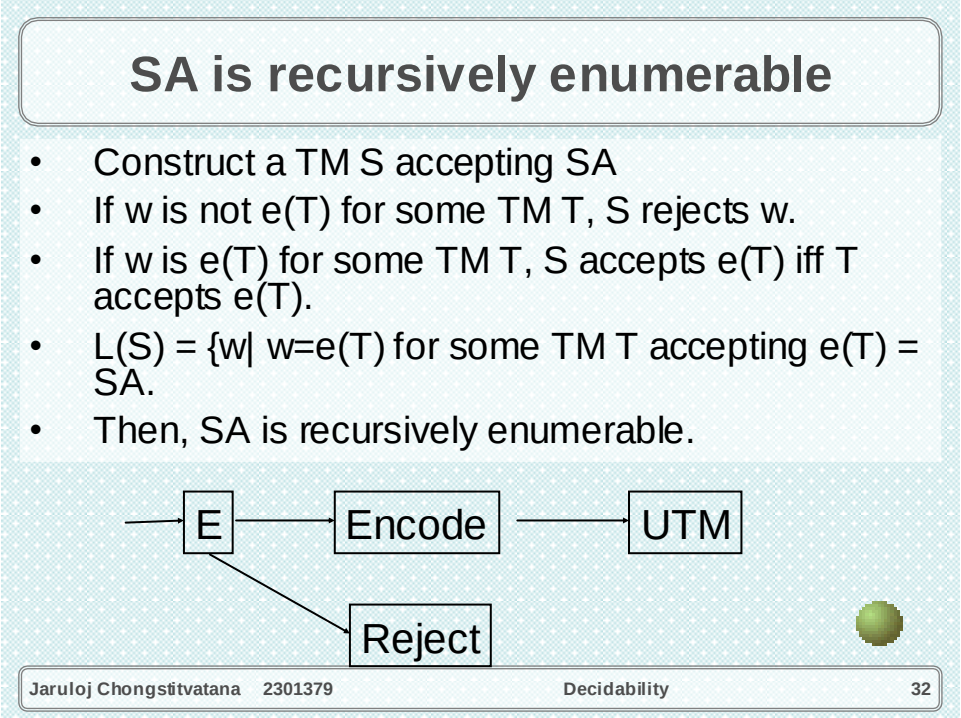
Then, the assumption is false.
That is, NSA is not recursively enumerable.

E is recursive

Theorem: E is recursive.
Proof:

We can construct a regular expression for E according to the definition of the encoding function as follows:
R = S 1 (M #)+
S = 0
M = Q , A , Q , A , D
Q = 0+
A = 0+
D = 0 + 00 + 000

Then, E is regular, and thus recursive.



SA is not recursive

- NSA = E – SA
- NSA is not recursively enumerable (from previous theorem), and thus not recursive.
- But E is recursive.
- From the closure property, if L1 and L2 are recursive, then L1 - L2 is recursive.
- Using its contrapositive, if L1 - L2 is not recursive, then L1 or L2 are not recursive.
- Since NSA is not recursive and E is recursive, SA is not recursive.

Co-R.E.

Definition

- A language L is co-R.E. if its complement $\overline{L}$ is R.E.
- It does not mean L is not R.E.

Examples:

- SA is R.E. $\overline{SA} = \overline{E} \cup NSA$ is not R.E.
  - $\overline{SA}$ is co-R.E., but not R.E.
- NSA is not R.E. $\overline{NSA} = \overline{E} \cup SA$ is R.E.
  - NSA is co-R.E., but not R.E.
- E is recursive, R.E., and co-R.E.

Relationship between R.E., co-R.E. and Recursive Languages

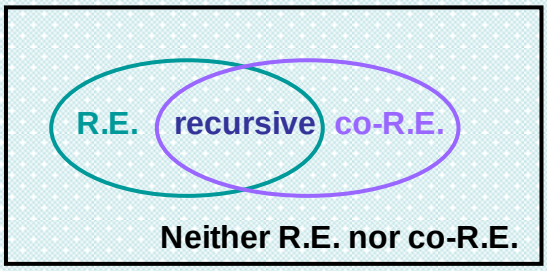
Theorem: Let L be any language. L is R.E. and co-R.E. iff L is recursive.

Proof:

- ($\rightarrow$) Let L be R.E. and co-R.E. Then, $\overline{L}$ is R.E. Thus, L is recursive.
- ($\leftarrow$) Let L be recursive. Then, L is R.E. From the closure under complementation of the class of recursive languages, $\overline{L}$ is also recursive. Then, $\overline{L}$ is also R.E.
- Thus, L is co-R.E.

Observation

- A language L is either
  - recursive
  - R.E., bot not recursive
  - co-R.E., but not recursive
  - Neither R.E. nor co-R.E.



Jaruloj Chongstitvatana 2301379 Decidability 36

Reduction Definition:

Let L1 and L2 be languages over Σ1 and Σ2, respectively. L1 is (many-one) reducible to L2, denoted by L1≤L2, if there is a TM M computing a function f: Σ1*Σ→2* such that w∈L1 ↔ f(w)∈L2.

Definition:

Let P1 and P2 be problems. P1 is (many-one) reducible to P2 if there is a TM M computing a function f: Σ1*Σ→2* such that w is a yes-instance of P1 ↔ f(w) is a yes-instance of P2.

Reduction

Definition:

A function f: Σ1*Σ→2* is a Turing-computable function if there is a Turing machine computing f.

Definition:

Let L1 and L2 be languages over Σ1 and Σ2, respectively. L1 is (many-one) reducible to L2, denoted by L1≤L2, if there is a Turing-computable function f: Σ1*Σ→2* such that w∈L1 ↔ f(w)∈L2.

Meaning of Reduction

P1 is reducible to P2 if ∃ TM M computing a function f: $\Sigma 1^*\Sigma \rightarrow 2^*$ such that w is a yes-instance of P1 $\leftrightarrow$ f(w) is a yes-instance of P2.

- If you can map yes-instances of problem A to yes-instances of problem B, then
  - we can solve A if we can solve B
  - it doesn't mean we can solve B if we can solve A
  - the decidability of B implies the decidability of A

Properties of reduction

Theorem: Let L be a language over Σ. $L \leq L$.

Proof:

Let L be a language over Σ.
Let f be an identity function from $\Sigma^*\Sigma \rightarrow ^*$.
Then, there is a TM computing f.
Because f is an identity function, $w \in L \leftrightarrow f(w)=w \in L$.
By the definition, $L \leq L$.

Properties of reduction

Theorem: Let L1 and L2 be languages over Σ.
If $L1 \leq L2$, then $\overline{L1} \leq L2$.

Proof:

Let L1 and L2 be languages over Σ.
Because $L1 \leq L2$, there is a function f such that $w \in L1 \leftrightarrow f(w) \in L2$, and a TM T computing f.
$w \in L1 \leftrightarrow f(w) \in \overline{L2}$.
By the definition, $\overline{L1} \leq L2$.

Properties of reduction

Theorem: **Let L1, L2 and L3 be languages over Σ.**
If L1≤L2 and L2≤L3, then L1≤L3.

Proof:

Let L1, L2 and L3 be languages over Σ.
There is a function f such that w∈L1 ↔ f(w)∈L2, and a TM T1 computing f because L1≤L2.
There is a function g such that w∈L2 ↔ g(w)∈L3, and a TM T2 computing g because L2≤L3.
w∈L1↔f(w)∈L2↔g(f(w))∈L3, and T1→T2 computes g(f(w)).
By the definition, L1≤L3.

Using reduction to prove decidability

Theorem: **If L2 is recursive, and L1≤L2, then L1 is also recursive.**

Proof:
Let L1 and L2 be languages over Σ, L1≤L2, and L2 be recursive.
Because L2 is recursive, there is a TM T2 computing χL2.
Because L1≤L2, there is a TM T1 computing a function f such that w∈L1 ↔ f(w)∈L2.

Using reduction to prove decidability

Construct a TM T=T1→T2. We show that T computes χL1.

- **If w∈L1, T1 in T computes f(w)∈L2 and T2 in T computes χL2(f(w)), which is 1.**
- **If w∉L1, T1 in T computes f(w) ∉L2 and T2 in T computes χL2(f(w)), which is 0.**

Thus, L1 is also recursive.

Using reduction to prove R.E

Theorem: **If L2 is R.E., and L1≤L2, then L1 is also R.E.**
Proof:
Let L1 and L2 be languages over Σ, L1≤L2, and L2 be R.E.
Because L2 is R.E, there is a TM T2 accepting L2.
Because L1≤L2, there is a TM T1 computing a function f such that w∈L1 ↔ f(w)∈L2.

Using reduction to prove R.E.

Construct a TM T=T1→T2. We show that T accepts L1.

- **If w∈L1, T1 in T computes f(w)∈L2 and T2 in T accepts f(w). Thus, T accepts w.**
- **If w∉L1, T1 in T computes f(w)∉L2 and T2 in T does not accept (f(w)). Thus, T does not accept w.**

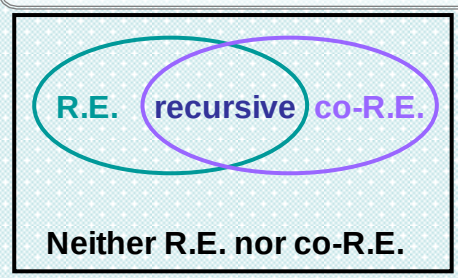
Thus, L1 is also R.E.

Using reduction to prove co-R.E.

Theorem: **If L2 is co-R.E., and L1≤L2, then L1 is also co-R.E.**
Proof:
Let L1 and L2 be languages over Σ, L1≤L2, and L2 be co-R.E.
Because L2 is co-R.E, $\bar{L}2$ is R.E.
Because L1≤L2, $\bar{L}1 \leq \bar{L}2$. Then, $\bar{L}1$ is R.E.
Thus, L1 is co-R.E.

Theorem: **If L2 is co-R.E., and L1≤L2, then L1 is also co-R.E.**
Proof:
Let L1 and L2 be languages over Σ, L1≤L2, and L2 be co-R.E.
Because L2 is co-R.E, $\bar{L}2$ is R.E.
Because L1≤L2, $\bar{L}1 \leq \bar{L}2$. Then, $\bar{L}1$ is R.E.
Thus, L1 is co-R.E.

Another way to prove undecidability



Let L1≤L2.
If L1 is not recursive / R.E. / co-R.E.,
then L2 is not recursive / R.E. / co-R.E.

To prove a language L is not recursive:

1. Guess where L is (not R.E. or not co-R.E.)
2. Choose another non-recursive language R which is of the same type
3. Show R ≤ L.

Jaruloj Chongstitvatana 2301379 Decidability 51

Guess if it's rec., R.E., co-R.E., or neither

Given a TM T,

- does T get to state q on blank tape?
- does T accept ε?
- does T output 1?
- does T accept everything?
- is L(T) finite?

Problem of accepting an empty string

- We will prove that the problem if a TM accepts an empty string is undecidable.
- This problem is corresponding to the following language.
  - Acceptε = {e(M)| M is a TM accepting ε}
- Thus, we will prove that Acceptε is not recursive.

Acceptε is not recursive.

Proof:
(Guess Acceptε is in R.E., but not co-R.E.)

- Show SA ≤ Acceptε

(We want a Turing-computable f n f(<T>)=<M> such that

  - T accepts e(T) → M accepts ε
  - T does not accept e(T) → M does not accept ε

- Let f(T)=M is a TM that first writes e(T) after its input and then runs T.
- M writes e(T) after its input. If its input is ε, T has e(T) as input.

Acceptε is not co–R.E.

Verify that T accepts e(T) ↔ M accepts ε
M writes e(T) and lets T run. If the input of M is ε:

- when T accepts e(T), M accepts ε.
- when T doesn't accept e(T), then M doesn't accept ε.

Acceptε is not co–R.E.

Next, we show that there is a TM TF computing f.
TF works as follows:

- changes the start state of T in e(T) to a new state
- add e(*Write*<*T*>), make its start state the start state of TF, and make the transition from its halt state to T's start state.

Then, SA ≤ Acceptε.
Then,Acceptε is not co-R.E, and is not recursive.

Halting problem

- Problem
  - Given a Turing machine T and string z, does T halt on z?
  - Given a program P and input z, does P halt on z?
- Language
  - Halt = {wΣ∈*| w=e(T)e(z) for a Turing machine T halting on z}.
  - Halt = {<T,z>| T is a Turing machine halting on z}.

Halting problem is undecidable

Proof:
Let Halt = {<T,z>| T is a Turing machine halting on z}.
(Guess Halt is in R.E., but not co-R.E.)

- Show SA ≤ Halt

(We want a Turing-computable f n f(<T1>)=<T2 ,z> such that

  - T1 accepts e(T1) → T2 halts on z
  - T1 does not accept e(T1) → T2 does not halt on z

Then, a possible function is f(<T>) = <T, e(T)> because T accepts e(T) ↔ T halts on e(T).)

Some other undecidable problems

- FINITE

Given a TM T, is L(T) finite?

Guess FINITE is neither R.E. nor co-R.E.

- To assure L(T) is finite, we need to run T on all possible input and count if T accepts a finite number of strings.
- To assure L(T) is infinite, we need to run T on all possible input and count if T accepts an infinite number of strings.

FINITE is not recursive FINITE is not recursive

Let FINITE={<T>| T is a TM such that L(T) is finite.}
Guess FINITE is neither R.E. nor co-R.E.
Choose NSA which is not co-R.E. to show that NSA≤FINITE.
We want to find a Turing-computable function f such that <T>∈NSA ↔ f(<T>)=M∈FINITE
<T>∈NSA→ M accepts ∅, and thus L(M) is finite.
<T>∉NSA→M accepts Σ*, and thus L(M) is infinite.
Then, let M=f(<T>) be a TM that runs T on its input, and accepts everything if T halts.

FINITE is not recursive

Now, we will show that <T>∈NSA ↔ <M>∈FINITE
If <T>∈NSA, then T does not accept <T>. Then, M does not get to start *AccAll*. Thus, M accepts nothing and L(M) is finite.
If <T>∉NSA, then T accepts <T>. Then, M gets pass T, and accept everything. Thus, M accepts

Checklist

- ❑ **Prove a language is recursive, R.E., or co-R.E.**
- ❑ **Prove closure properties of these classes of languages**
- ❑ **Prove properties of reduction**
- ❑ **Prove a language is not recursive, not R.E., or not co-R.E.**

- ❑ prove a problem is decidable
- ❑ Prove a problem is undecidable