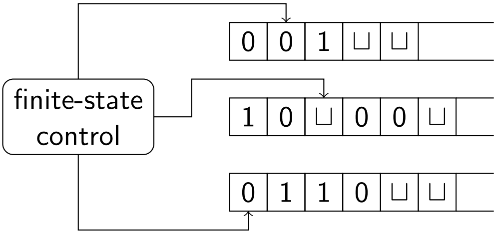
CSE 135: Introduction to Theory of Computation
Variants of Turing Machines and Church-Turing Thesis

Sungjin Im

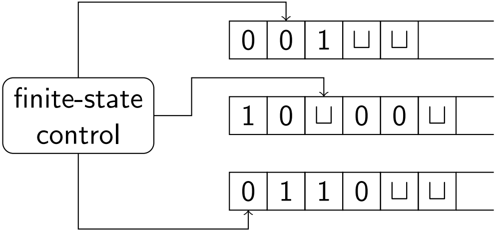
University of California, Merced

04-09-2015

Multi-Tape Turing Machine

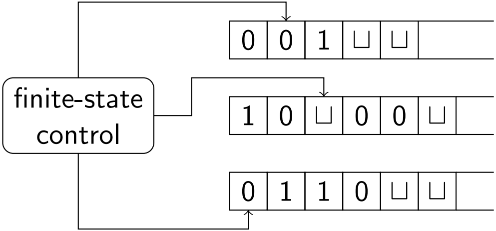


Multi-Tape Turing Machine



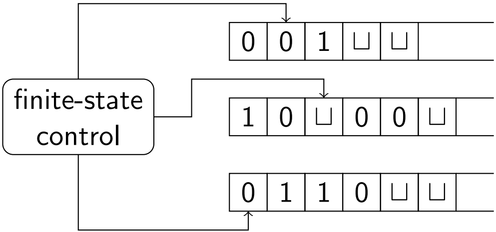
- Input on Tape 1

Multi-Tape Turing Machine



- Input on Tape 1
- Initially all heads scanning cell 1, and tapes 2 to k blank

Multi-Tape Turing Machine



- Input on Tape 1
- Initially all heads scanning cell 1, and tapes 2 to k blank
- In one step: Read symbols under each of the k-heads, and depending on the current control state, write new symbols on the tapes, move the each tape head (possibly in different directions), and change state.

Multi-Tape Turing Machine

Formal Definition

A k-tape Turing Machine is $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ where

Multi-Tape Turing Machine

Formal Definition

A k-tape Turing Machine is $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ where

- Q is a finite set of control states

Multi-Tape Turing Machine

Formal Definition

A k-tape Turing Machine is $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}})$ where

- Q is a finite set of control states
- Σ is a finite set of input symbols

Multi-Tape Turing Machine

Formal Definition

A k-tape Turing Machine is $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}})$ where

- Q is a finite set of control states
- Σ is a finite set of input symbols
- $\Gamma \supseteq \Sigma$ is a finite set of tape symbols. Also, a blank symbol $\sqcup \in \Gamma \setminus \Sigma$

Multi-Tape Turing Machine

Formal Definition

A k-tape Turing Machine is $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}})$ where

- Q is a finite set of control states
- Σ is a finite set of input symbols
- $\Gamma \supseteq \Sigma$ is a finite set of tape symbols. Also, a blank symbol $\sqcup \in \Gamma \setminus \Sigma$
- $q_0 \in Q$ is the initial state

Multi-Tape Turing Machine

Formal Definition

A k-tape Turing Machine is $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ where

- Q is a finite set of control states
- Σ is a finite set of input symbols
- $\Gamma \supseteq \Sigma$ is a finite set of tape symbols. Also, a blank symbol $\sqcup \in \Gamma \setminus \Sigma$
- $q_0 \in Q$ is the initial state
- $q_{\text{acc}} \in Q$ is the accept state

Multi-Tape Turing Machine

Formal Definition

A k-tape Turing Machine is $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}})$ where

- Q is a finite set of control states
- Σ is a finite set of input symbols
- $\Gamma \supseteq \Sigma$ is a finite set of tape symbols. Also, a blank symbol $\sqcup \in \Gamma \setminus \Sigma$
- $q_0 \in Q$ is the initial state
- $q_{\mathsf{acc}} \in Q$ is the accept state
- $q_{\mathsf{rej}} \in Q$ is the reject state, where $q_{\mathsf{rej}} \neq q_{\mathsf{acc}}$

Multi-Tape Turing Machine

Formal Definition

A k-tape Turing Machine is $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}})$ where

- Q is a finite set of control states
- Σ is a finite set of input symbols
- $\Gamma \supseteq \Sigma$ is a finite set of tape symbols. Also, a blank symbol $\sqcup \in \Gamma \setminus \Sigma$
- $q_0 \in Q$ is the initial state
- $q_{\mathsf{acc}} \in Q$ is the accept state
- $q_{\mathsf{rej}} \in Q$ is the reject state, where $q_{\mathsf{rej}} \neq q_{\mathsf{acc}}$
- $\delta : (Q \setminus \{q_{\mathsf{acc}}, q_{\mathsf{rej}}\}) \times \Gamma^k \to Q \times (\Gamma \times \{\mathsf{L}, \mathsf{R}\})^k$ is the transition function.

Computation, Acceptance and Language

- A configuration of a multi-tape TM must describe the state, contents of all k-tapes, and positions of all k-heads. Thus, $\mathrm{C} \in Q \times (\Gamma^*\{*\}\Gamma\Gamma^*)^k$, where $*$ denotes the head position.

Computation, Acceptance and Language

- A configuration of a multi-tape TM must describe the state, contents of all k-tapes, and positions of all k-heads. Thus, $\mathrm{C} \in Q \times (\Gamma^*\{*\}\Gamma\Gamma^*)^k$, where $*$ denotes the head position.
- Accepting configuration is one where the state is q_{acc}, and starting configuration on input w is $(q_0, *w, *\sqcup, \ldots, *\sqcup)$

Computation, Acceptance and Language

- A configuration of a multi-tape TM must describe the state, contents of all k-tapes, and positions of all k-heads. Thus, $\mathrm{C} \in Q \times (\Gamma^*\{*\}\Gamma\Gamma^*)^k$, where $*$ denotes the head position.
- Accepting configuration is one where the state is q_{acc}, and starting configuration on input w is $(q_0, *w, *\sqcup, \ldots, *\sqcup)$
- Formal definition of a single step is skipped.

Computation, Acceptance and Language

- A configuration of a multi-tape TM must describe the state, contents of all k-tapes, and positions of all k-heads. Thus, $\mathrm{C} \in Q \times (\Gamma^*\{*\}\Gamma\Gamma^*)^k$, where $*$ denotes the head position.
- Accepting configuration is one where the state is q_{acc}, and starting configuration on input w is $(q_0, *w, *\sqcup, \ldots, *\sqcup)$
- Formal definition of a single step is skipped.
- w is accepted by M, if from the starting configuration with w as input, M reaches an accepting configuration.

Computation, Acceptance and Language

- A configuration of a multi-tape TM must describe the state, contents of all k-tapes, and positions of all k-heads. Thus, $\mathrm{C} \in Q \times (\Gamma^*\{*\}\Gamma\Gamma^*)^k$, where $*$ denotes the head position.
- Accepting configuration is one where the state is q_{acc}, and starting configuration on input w is $(q_0, *w, *\sqcup, \ldots, *\sqcup)$
- Formal definition of a single step is skipped.
- w is accepted by M, if from the starting configuration with w as input, M reaches an accepting configuration.
- $L(M) = \{w \mid w \text{ accepted by } M\}$

Expressive Power of multi-tape TM

Expressive Power of multi-tape TM

Theorem

For any k-tape Turing Machine M, there is a single tape TM $single(M)$ such that $L(single(M)) = L(M)$.

Expressive Power of multi-tape TM

Theorem

For any k-tape Turing Machine M, there is a single tape TM $\mathit{single}(M)$ such that $L(\mathit{single}(M)) = L(M)$.

Challenges

- How do we store k-tapes in one?

Expressive Power of multi-tape TM

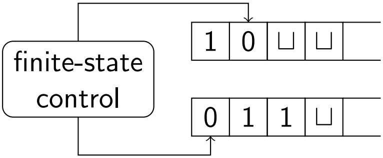
Theorem

For any k-tape Turing Machine M, there is a single tape TM $single(M)$ such that $L(single(M)) = L(M)$.

Challenges

- How do we store k-tapes in one?
- How do we simulate the movement of k independent heads?

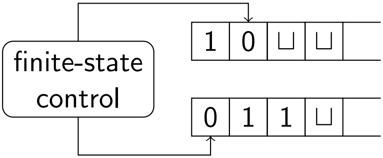
Storing Multiple Tapes



Multi-tape TM M

Store in cell i contents of cell i of all tapes.

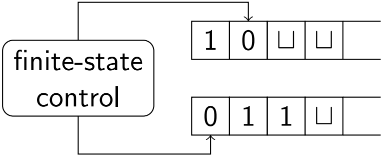
Storing Multiple Tapes



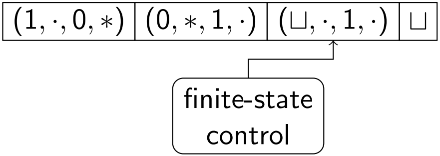
Multi-tape TM *M*

Store in cell i contents of cell i of all tapes. “Mark” head position of tape with $*$.

Storing Multiple Tapes



Multi-tape TM M



1-tape equivalent single(M)

Store in cell i contents of cell i of all tapes. “Mark” head position of tape with $*$.

Simulating One Step

Challenge 1: Head of 1-Tape TM is pointing to one cell. How do we find out all the k symbols that are being read by the k heads, which maybe in different cells?

Simulating One Step

Challenge 1: Head of 1-Tape TM is pointing to one cell. How do we find out all the k symbols that are being read by the k heads, which maybe in different cells?

- Read the tape from left to right, storing the contents of the cells being scanned in the state, as we encounter them.

Simulating One Step

Challenge 1: Head of 1-Tape TM is pointing to one cell. How do we find out all the k symbols that are being read by the k heads, which maybe in different cells?

- Read the tape from left to right, storing the contents of the cells being scanned in the state, as we encounter them.

Challenge 2: After this scan, 1-tape TM knows the next step of k-tape TM. How do we change the contents and move the heads?

Simulating One Step

Challenge 1: Head of 1-Tape TM is pointing to one cell. How do we find out all the k symbols that are being read by the k heads, which maybe in different cells?

- Read the tape from left to right, storing the contents of the cells being scanned in the state, as we encounter them.

Challenge 2: After this scan, 1-tape TM knows the next step of k-tape TM. How do we change the contents and move the heads?

- Once again, scan the tape, change all relevant contents, "move" heads (i.e., move $*$s), and change state.

Overall Algorithm

On input w, the 1-tape TM will work as follows.

1. First the machine will rewrite input w to be in "new" format.

Overall Algorithm

On input w, the 1-tape TM will work as follows.

1. First the machine will rewrite input w to be in “new” format.
2. To simulate one step

Overall Algorithm

On input *w*, the 1-tape TM will work as follows.

1. First the machine will rewrite input *w* to be in "new" format.
2. To simulate one step
    - Read from left-to-right remembering symbols read on each tape, and move all the way back to leftmost position.

Overall Algorithm

On input *w*, the 1-tape TM will work as follows.

1. First the machine will rewrite input *w* to be in "new" format.
2. To simulate one step
    - Read from left-to-right remembering symbols read on each tape, and move all the way back to leftmost position.
    - Read from left-to-right, changing symbols, and moving those heads that need to be moved right.

Overall Algorithm

On input w, the 1-tape TM will work as follows.

1. First the machine will rewrite input w to be in "new" format.
2. To simulate one step
    - Read from left-to-right remembering symbols read on each tape, and move all the way back to leftmost position.
    - Read from left-to-right, changing symbols, and moving those heads that need to be moved right.
    - Scan back from right-to-left moving the heads that need to be moved left.

Nondeterministic Turing Machine

Deterministic TM: At each step, there is one possible next state, symbols to be written and direction to move the head, or the TM may halt.

Nondeterministic Turing Machine

Deterministic TM: At each step, there is one possible next state, symbols to be written and direction to move the head, or the TM may halt.

Nondeterministic TM: At each step, there are finitely many possibilities. So formally, $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}})$, where

Nondeterministic Turing Machine

Deterministic TM: At each step, there is one possible next state, symbols to be written and direction to move the head, or the TM may halt.
Nondeterministic TM: At each step, there are finitely many possibilities. So formally, $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}})$, where

- $Q, \Sigma, \Gamma, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}$ are as before for 1-tape machine

Nondeterministic Turing Machine

Deterministic TM: At each step, there is one possible next state, symbols to be written and direction to move the head, or the TM may halt.
Nondeterministic TM: At each step, there are finitely many possibilities. So formally, $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$, where

- $Q, \Sigma, \Gamma, q_0, q_{\text{acc}}, q_{\text{rej}}$ are as before for 1-tape machine
- $\delta : (Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{\text{L}, \text{R}\})$

Computation, Acceptance and Language

- A configuration of a nondeterministic TM is exactly the same as that of a 1-tape TM.

Computation, Acceptance and Language

- A configuration of a nondeterministic TM is exactly the same as that of a 1-tape TM. So are notions of starting configuration and accepting configuration.

Computation, Acceptance and Language

- A configuration of a nondeterministic TM is exactly the same as that of a 1-tape TM. So are notions of starting configuration and accepting configuration.
- A single step $\vdash$ is defined similarly.
  $X_1X_2\cdots X_{i-1}qX_i\cdots X_n \vdash X_1X_2\cdots pX_{i-1}Y\cdots X_n$, if $(p,Y,L)\in\delta(q,X_i)$

Computation, Acceptance and Language

- A configuration of a nondeterministic TM is exactly the same as that of a 1-tape TM. So are notions of starting configuration and accepting configuration.
- A single step $\vdash$ is defined similarly. $X_1X_2 \cdots X_{i-1}qX_i \cdots X_n \vdash X_1X_2 \cdots pX_{i-1}Y \cdots X_n$, if $(p, Y, L) \in \delta(q, X_i)$; case for right moves is analogous.

Computation, Acceptance and Language

- A configuration of a nondeterministic TM is exactly the same as that of a 1-tape TM. So are notions of starting configuration and accepting configuration.
- A single step $\vdash$ is defined similarly.
  $X_1X_2 \cdots X_{i-1}qX_i \cdots X_n \vdash X_1X_2 \cdots pX_{i-1}Y \cdots X_n$, if $(p, Y, L) \in \delta(q, X_i)$; case for right moves is analogous.
- w is accepted by M, if from the starting configuration with w as input, M reaches an accepting configuration, for some sequence of choices at each step.

Computation, Acceptance and Language

- A configuration of a nondeterministic TM is exactly the same as that of a 1-tape TM. So are notions of starting configuration and accepting configuration.
- A single step $\vdash$ is defined similarly.
  $X_1X_2\cdots X_{i-1}qX_i\cdots X_n \vdash X_1X_2\cdots pX_{i-1}Y\cdots X_n$, if $(p, Y, L) \in \delta(q, X_i)$; case for right moves is analogous.
- w is accepted by M, if from the starting configuration with w as input, M reaches an accepting configuration, for some sequence of choices at each step.
- $L(M) = \{w \mid w \text{ accepted by } M\}$

Expressive Power of Nondeterministic TM

Expressive Power of Nondeterministic TM

Theorem

For any nondeterministic Turing Machine M, there is a (deterministic) TM $det(M)$ such that $L(det(M)) = L(M)$.

Expressive Power of Nondeterministic TM

Theorem

For any nondeterministic Turing Machine M, there is a (deterministic) TM $det(M)$ such that $L(det(M)) = L(M)$.

Proof Idea

$\mathrm{det}(M)$ will simulate M on the input.

Expressive Power of Nondeterministic TM

Theorem

For any nondeterministic Turing Machine M, there is a (deterministic) TM $det(M)$ such that $L(det(M)) = L(M)$.

Proof Idea

det(M) will simulate M on the input.

- Idea 1: det(M) tries to keep track of all possible "configurations" that M could possibly be after each step.

Expressive Power of Nondeterministic TM

Theorem

For any nondeterministic Turing Machine M, there is a (deterministic) TM $det(M)$ such that $L(det(M)) = L(M)$.

Proof Idea

det(M) will simulate M on the input.

- Idea 1: det(M) tries to keep track of all possible "configurations" that M could possibly be after each step. Works for DFA simulation of NFA

Expressive Power of Nondeterministic TM

Theorem

For any nondeterministic Turing Machine M, there is a (deterministic) TM $det(M)$ such that $L(det(M)) = L(M)$.

Proof Idea

det(M) will simulate M on the input.

- Idea 1: det(M) tries to keep track of all possible "configurations" that M could possibly be after each step. Works for DFA simulation of NFA but not convenient here.

Expressive Power of Nondeterministic TM

Theorem

For any nondeterministic Turing Machine M, there is a (deterministic) TM $det(M)$ such that $L(det(M)) = L(M)$.

Proof Idea

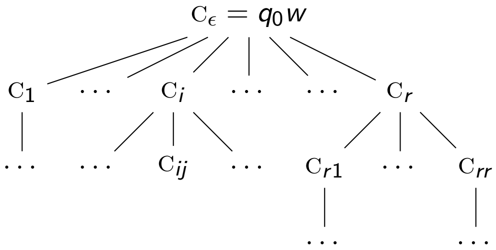
det(M) will simulate M on the input.

- Idea 1: det(M) tries to keep track of all possible "configurations" that M could possibly be after each step. Works for DFA simulation of NFA but not convenient here.
- Idea 2: det(M) will simulate M on each possible sequence of computation steps that M may try in each step.

Nondeterministic Computation

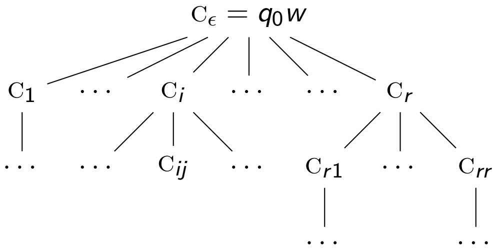
- If $r = \max_{q,X} |\delta(q, X)|$ then the runs of M can be organized as an r-branching tree.

Nondeterministic Computation



- If $r = \max_{q,X} |\delta(q, X)|$ then the runs of M can be organized as an r-branching tree.
- $\mathrm{C}_{i_1 i_2 \cdots i_n}$ is the configuration of M after n-steps, where choice i_1 is taken in step 1, i_2 in step 2, and so on.

Nondeterministic Computation



- If $r = \max_{q,X} |\delta(q, X)|$ then the runs of M can be organized as an r-branching tree.
- $\mathrm{C}_{i_1 i_2 \cdots i_n}$ is the configuration of M after n-steps, where choice i_1 is taken in step 1, i_2 in step 2, and so on.
- Input w is accepted iff $\exists$ accepting configuration in tree.

Proof Idea

The machine $\det(M)$ will search for an accepting configuration in computation tree

Proof Idea

The machine det(M) will search for an accepting configuration in computation tree

- The configuration at any vertex can be obtained by simulating M on the appropriate sequence of nondeterministic choices

Proof Idea

The machine det(M) will search for an accepting configuration in computation tree

- The configuration at any vertex can be obtained by simulating M on the appropriate sequence of nondeterministic choices
- det(M) will perform a BFS on the tree.

Proof Idea

The machine det(M) will search for an accepting configuration in computation tree

- The configuration at any vertex can be obtained by simulating M on the appropriate sequence of nondeterministic choices
- det(M) will perform a BFS on the tree. Why not a DFS?

Proof Idea

The machine det(M) will search for an accepting configuration in computation tree

- The configuration at any vertex can be obtained by simulating M on the appropriate sequence of nondeterministic choices
- det(M) will perform a BFS on the tree. Why not a DFS?

Observe that det(M) may not terminate if w is not accepted.

Proof Details

det(M) will use 3 tapes to simulate M

Proof Details

det(M) will use 3 tapes to simulate M

Proof Details

det(M) will use 3 tapes to simulate M (note, multitape TMs are equivalent to 1-tape TMs)

- Tape 1, called input tape, will always hold input w

Proof Details

det(M) will use 3 tapes to simulate M (note, multitape TMs are equivalent to 1-tape TMs)

- Tape 1, called input tape, will always hold input w
- Tape 2, called simulation tape, will be used as M's tape when simulating M on a sequence of nondeterministic choices

Proof Details

det(M) will use 3 tapes to simulate M (note, multitape TMs are equivalent to 1-tape TMs)

- Tape 1, called input tape, will always hold input w
- Tape 2, called simulation tape, will be used as M's tape when simulating M on a sequence of nondeterministic choices
- Tape 3, called choice tape, will store the current sequence of nondeterministic choices

Execution of det(M)

1. Initially: Input tape contains w, simulation tape and choice tape are blank
2. Copy contents of input tape onto simulation tape
3. Simulate M using simulation tape as its (only) tape
    3.1 At the next step of M, if state is q, simulation tape head reads X, and choice tape head reads i, then simulate the ith possibility in $\delta(q, X)$; if i is not a valid choice, then goto step 4
    3.2 After changing state, simulation tape contents, and head position on simulation tape, move choice tape's head to the right. If Tape 3 is now scanning $\sqcup$, then goto step 4
    3.3 If M accepts then accept and halt, else goto step 3(1) to simulate the next step of M.
4. Write the lexicographically next choice sequence on choice tape, erase everything on simulation tape and goto step 2.

Deterministic Simulation

In a nutshell

- det(M) simulates M over and over again, for different sequences, and for different number of steps.

Deterministic Simulation

In a nutshell

- det(M) simulates M over and over again, for different sequences, and for different number of steps.
- If M accepts w then there is a sequence of choices that will lead to acceptance. det(M) will eventually have this sequence on choice tape, and then its simulation M will accept.

Deterministic Simulation

In a nutshell

- det(M) simulates M over and over again, for different sequences, and for different number of steps.
- If M accepts w then there is a sequence of choices that will lead to acceptance. det(M) will eventually have this sequence on choice tape, and then its simulation M will accept.
- If M does not accept w then no sequence of choices leads to acceptance. det(M) will therefore never halt!

Random Access Machines

This is an idealized model of modern computers.

Random Access Machines

This is an idealized model of modern computers. Have a finite number of “registers”, an infinite number of available memory locations, and store a sequence of instructions or “program” in memory.

Random Access Machines

This is an idealized model of modern computers. Have a finite number of “registers”, an infinite number of available memory locations, and store a sequence of instructions or “program” in memory.

- Initially, the program instructions are stored in a contiguous block of memory locations starting at location 1. All registers and memory locations, other than those storing the program, are set to 0.

Instruction Set

- `add X, Y`: Add the contents of registers X and Y and store the result in X.
- `loadc X, I`: Place the constant I in register X.
- `load X, M`: Load the contents of memory location M into register X.
- `loadI X, M`: Load the contents of the location "pointed to" by the contents of M into register X.
- `store X, M`: store the contents of register X in memory location M.
- `jmp M`: The next instruction to be executed is in location M.
- `jmz X, M`: If register X is 0, then jump to instruction M.
- `halt`: Halt execution.

Expressive Power of RAMs

Expressive Power of RAMs

Theorem

Anything computed on a RAM can be computed on a Turing machine.

Expressive Power of RAMs

Theorem

Anything computed on a RAM can be computed on a Turing machine.

Robustness of the Class of TM Languages

Various efforts to capture mechanical computation have the same expressive power.

Robustness of the Class of TM Languages

Various efforts to capture mechanical computation have the same expressive power.

- Non-Turing Machine models: random access machines, λ-calculus, type 0 grammars, first-order reasoning, π-calculus, ...

Robustness of the Class of TM Languages

Various efforts to capture mechanical computation have the same expressive power.

- Non-Turing Machine models: random access machines, λ-calculus, type 0 grammars, first-order reasoning, π-calculus, ...
- Enhanced Turing Machine models: TM with 2-way infinite tape, multi-tape TM, nondeterministic TM, probabilistic Turing Machines, quantum Turing Machines ...

Robustness of the Class of TM Languages

Various efforts to capture mechanical computation have the same expressive power.

- Non-Turing Machine models: random access machines, λ-calculus, type 0 grammars, first-order reasoning, π-calculus, ...
- Enhanced Turing Machine models: TM with 2-way infinite tape, multi-tape TM, nondeterministic TM, probabilistic Turing Machines, quantum Turing Machines ...
- Restricted Turing Machine models: queue machines, 2-stack machines, 2-counter machines, ...

Church-Turing Thesis

“Anything solvable via a mechanical procedure can be solved on a Turing Machine.”

Church-Turing Thesis

"Anything solvable via a mechanical procedure can be solved on a Turing Machine."

- Not a mathematical statement that can be proved or disproved!

Church-Turing Thesis

“Anything solvable via a mechanical procedure can be solved on a Turing Machine.”

- Not a mathematical statement that can be proved or disproved!
- Strong evidence based on the fact that many attempts to define computation yield the same expressive power

Consequences

- In the course, we will use an informal pseudo-code to argue that a problem/language can be solved on Turing machines

Consequences

- In the course, we will use an informal pseudo-code to argue that a problem/language can be solved on Turing machines
- To show that something can be solved on Turing machines, you can use any programming language of choice, *unless the problem specifically asks you to design a Turing Machine*

Terminology for Describing Turing Machines

1. Formal description. Spell out in full the TM's states, transition functions, etc. Lowest level of description.

Terminology for Describing Turing Machines

1. Formal description. Spell out in full the TM's states, transition functions, etc. Lowest level of description.
2. Implmentation description. Use English prose to describe the way that the TM moves its head and the way that it stores data on its tape. Intermediate level of description. Note that you can only use "local" knowledge.

Terminology for Describing Turing Machines

1. Formal description. Spell out in full the TM's states, transition functions, etc. Lowest level of description.
2. Implmentation description. Use English prose to describe the way that the TM moves its head and the way that it stores data on its tape. Intermediate level of description. Note that you can only use "local" knowledge.
3. High-level description. Describe an algorithm, ignoring implementation. Highest level of description. Justified by Church-Turing thesis.

Revisiting Type 0 grammar

Grammars

Definition

A grammar is $G = (V, \Sigma, R, S)$, where

- V is a finite set of variables/non-terminals
- Σ is a finite set of terminals
- $S \in V$ is the start symbol
- $R \subseteq (\Sigma \cup V)^* \times (\Sigma \cup V)^*$ is a finite set of rules/productions

Grammars

Definition

A grammar is $G = (V, \Sigma, R, S)$, where

- V is a finite set of variables/non-terminals
- Σ is a finite set of terminals
- $S \in V$ is the start symbol
- $R \subseteq (\Sigma \cup V)^* \times (\Sigma \cup V)^*$ is a finite set of rules/productions

We say $\gamma_1 \alpha \gamma_2 \Rightarrow_G \gamma_1 \beta \gamma_2$ iff $(\alpha \to \beta) \in R$.

Grammars

Definition

A grammar is $G = (V, \Sigma, R, S)$, where

- V is a finite set of variables/non-terminals
- Σ is a finite set of terminals
- $S \in V$ is the start symbol
- $R \subseteq (\Sigma \cup V)^* \times (\Sigma \cup V)^*$ is a finite set of rules/productions

We say $\gamma_1 \alpha \gamma_2 \Rightarrow_G \gamma_1 \beta \gamma_2$ iff $(\alpha \to \beta) \in R$. And
$L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow}_G w\}$

Example

Example

Consider the grammar G with $\Sigma = \{a\}$ with

$$\begin{array}{lll} S \rightarrow \$Ca\# \mid a \mid \epsilon & Ca \rightarrow aaC & \$D \rightarrow \$C \\ C\# \rightarrow D\# \mid E & aD \rightarrow Da & aE \rightarrow Ea \\ \$E \rightarrow \epsilon & & \end{array}$$

The following are derivations in this grammar

$$S \Rightarrow \$Ca\# \Rightarrow \$aaC\# \Rightarrow \$aaE \Rightarrow \$aEa \Rightarrow \$Eaa \Rightarrow aa$$

$$\begin{aligned} S & \Rightarrow \$Ca\# \Rightarrow \$aaC\# \Rightarrow \$aaD\# \Rightarrow \$aDa\# \Rightarrow \$Daa\# \Rightarrow \$Caa\# \\ & \Rightarrow \$aaCa\# \Rightarrow \$aaaaC\# \Rightarrow \$aaaaE \Rightarrow \$aaaEa \Rightarrow \$aaEaa \\ & \Rightarrow \$aEaaa \Rightarrow \$Eaaaa \Rightarrow aaaa \end{aligned}$$

Example

Example

Consider the grammar G with $\Sigma = \{a\}$ with

$$\begin{array}{lll} S \rightarrow \$Ca\# \mid a \mid \epsilon & Ca \rightarrow aaC & \$D \rightarrow \$C \\ C\# \rightarrow D\# \mid E & aD \rightarrow Da & aE \rightarrow Ea \\ \$E \rightarrow \epsilon & & \end{array}$$

The following are derivations in this grammar

$$S \Rightarrow \$Ca\# \Rightarrow \$aaC\# \Rightarrow \$aaE \Rightarrow \$aEa \Rightarrow \$Eaa \Rightarrow aa$$

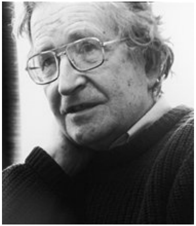
$$\begin{aligned} S &\Rightarrow \$Ca\# \Rightarrow \$aaC\# \Rightarrow \$aaD\# \Rightarrow \$aDa\# \Rightarrow \$Daa\# \Rightarrow \$Caa\# \\ &\Rightarrow \$aaCa\# \Rightarrow \$aaaaC\# \Rightarrow \$aaaaE \Rightarrow \$aaaEa \Rightarrow \$aaEaa \\ &\Rightarrow \$aEaaa \Rightarrow \$Eaaaa \Rightarrow aaaa \end{aligned}$$

$L(G) = \{a^i \mid i \text{ is a power of } 2\}$

Grammars for each task

- What is the expressive power of these grammars?

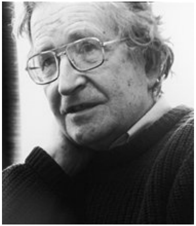
Grammars for each task



Noam Chomsky

- What is the expressive power of these grammars?
- Restricting the types of rules, allows one to describe different aspects of natural languages

Grammars for each task



Noam Chomsky

- What is the expressive power of these grammars?
- Restricting the types of rules, allows one to describe different aspects of natural languages
- These grammars form a hierarchy

Type 0 Grammars

Definition

Type 0 grammars are those where the rules are of the form

$$\alpha \to \beta$$

where $\alpha, \beta \in (\Sigma \cup V)^*$

Example

Consider the grammar G with $\Sigma = \{a\}$ with

$$\begin{array}{lll}
S \to \$Ca\# \mid a \mid \epsilon & Ca \to aaC & \$D \to \$C \\
C\# \to D\# \mid E & aD \to Da & aE \to Ea \\
\$E \to \epsilon & &
\end{array}$$

Expressive Power of Type 0 Grammars

Theorem

L is recursively enumerable (recognizable) iff there is a type 0 grammar G such that $L = L(G)$.

Expressive Power of Type 0 Grammars

Theorem

L is recursively enumerable (recognizable) iff there is a type 0 grammar G such that $L = L(G)$.

Thus, type 0 grammars are as powerful as Turing machines.

Recognizing Type 0 languages

Proposition

If $G = (V, \Sigma, R, S)$ is a type 0 grammar then $L(G)$ is recursively enumerable.

Recognizing Type 0 languages

Proposition

If $G = (V, \Sigma, R, S)$ is a type 0 grammar then $L(G)$ is recursively enumerable.

Proof.

We will show that $L(G)$ is recognized by a 2-tape non-deterministic Turing machine M, with tape 1 storing the input w, and tape 2 used to construct a derivation of w from S. $\cdot\cdot\longrightarrow$

Recognizing Type 0 Grammars

Proof (contd).

Recognizing Type 0 Grammars

Proof (contd).

- At any given time tape 2, stores the current string of the derivation; initial tape contains S.

Recognizing Type 0 Grammars

Proof (contd).

- At any given time tape 2, stores the current string of the derivation; initial tape contains S.
- To simulate the next derivation step, M will (nondeterministically) choose a rule to apply, scan from left to right and choose (nondeterministically) a position to apply the rule, replace the substring matching the LHS of the rule with the RHS to get the string at the next step of derivation.

Recognizing Type 0 Grammars

Proof (contd).

- At any given time tape 2, stores the current string of the derivation; initial tape contains S.
- To simulate the next derivation step, M will (nondeterministically) choose a rule to apply, scan from left to right and choose (nondeterministically) a position to apply the rule, replace the substring matching the LHS of the rule with the RHS to get the string at the next step of derivation.
- If tape 2 contains only terminal symbols, then M will check to see if it matches tape 1. If so, the input is accepted, else it is rejected. □

Describing R.E. Languages

Proposition

If L is recursively enumerable, then there is a type 0 grammar G such that $L = L(G)$.

Describing R.E. Languages

Proposition

If L is recursively enumerable, then there is a type 0 grammar G such that $L = L(G)$.

Proof.

Let M be a Turing machine recognizing L. The grammar G will simulate M "backwards" starting from an accepting configuration.

□

Describing R.E. Languages

Proposition

If L is recursively enumerable, then there is a type 0 grammar G such that $L = L(G)$.

Proof.

Let M be a Turing machine recognizing L. The grammar G will simulate M "backwards" starting from an accepting configuration.

- A string γ in the derivation will encode a configuration of M

□

Describing R.E. Languages

Proposition

If L is recursively enumerable, then there is a type 0 grammar G such that $L = L(G)$.

Proof.

Let M be a Turing machine recognizing L. The grammar G will simulate M "backwards" starting from an accepting configuration.

- A string γ in the derivation will encode a configuration of M
- G has rules such that $\gamma_1 \Rightarrow \gamma_2$ iff $\gamma_2 \vdash_M \gamma_1$

□

Describing R.E. Languages

Proposition

If L is recursively enumerable, then there is a type 0 grammar G such that $L = L(G)$.

Proof.

Let M be a Turing machine recognizing L. The grammar G will simulate M "backwards" starting from an accepting configuration.

- A string γ in the derivation will encode a configuration of M
- G has rules such that $\gamma_1 \Rightarrow \gamma_2$ iff $\gamma_2 \vdash_M \gamma_1$
- The rules of S will generate an accepting configuration of M

□

Describing R.E. Languages

Proposition

If L is recursively enumerable, then there is a type 0 grammar G such that $L = L(G)$.

Proof.

Let M be a Turing machine recognizing L. The grammar G will simulate M "backwards" starting from an accepting configuration.

- A string γ in the derivation will encode a configuration of M
- G has rules such that $\gamma_1 \Rightarrow \gamma_2$ iff $\gamma_2 \vdash_M \gamma_1$
- The rules of S will generate an accepting configuration of M
- Once (some) initial configuration $q_0 w$ is generated, rules in G will erase symbols to produce the terminal w.

□