Reference: Introduction to Theoretical Computer Science by Boaz Barak

Formatting:

- **Definitions**
- Nontrivial results / intermediate steps
- *Emphasis*, emphasis

Chapter 1: Math background

- **Layering of DAG**: Let $G = (V, E)$ be a directed graph. A *layering* of G is $f : V \to \mathbb{N}$ such that $\forall u \to v : f(u) < f(v)$
  - A layering is **minimal** of $\forall v \in V$, v has no in-neighbors $\implies f(v) = 0$, else $\exists u \to v : f(u) = f(v) - 1$
- **Topological Sort Theorem:** G acyclic $\iff$ $\exists$minimal layering of G
- Asymptotic Notation: for $f, g : \mathbb{N} \to \mathbb{R}_+$
  - $f = O(g) \iff \exists a, N_0 \in \mathbb{N} : \forall n > N_0, f(n) \leq a \cdot g(n)$`
    - $f = \Theta(G) \iff f = O(g), g = O(f)$
    - $f = \Omega(g) \iff g = O(f)$
  - $f = o(g) \iff \forall \epsilon > 0, \exists N_0 : f(n) < \epsilon \cdot g(n)$
    - $f = \omega(g) \iff g = o(f)$
- For finite set Σ, $\Sigma^* \equiv \bigcup \Sigma^i$

Chapter 2: Representation

- **Cantor's Theorem**: no surjective function from a set S to its power set $\mathscr{P}(S)$
  - Now for each $S' \subseteq S \in \mathscr{P}(S)$, $s \in S' \leftrightarrow g : S \to \{0,1\}$ where $g(s \in S) = 1 \iff s \in S' \in \mathcal{S}$.
  - Suppose there is surjection $h : S \to \{0,1\}^S$, then $\sigma : S \to \{0,1\}$ defined by $\sigma(s) = \neg[h(s)](s)$ cannot be in $\mathrm{Im}(h)$
    - Suppose $\exists s_0 \in S : h(s_0) = \sigma$, then $\sigma(s_0) \neq [h(s_0)](s_0) \implies \sigma \neq h(s_0)$
  - Corollary: set of boolean functions $\{0,1\}^* \to \{0,1\}$ is uncountable
    - $\{0,1\}^* \cong \mathbb{N}$ and $\{f : \{0,1\}^* \to \{0,1\}\} = \{0,1\}^{\{0,1\}^*} \cong \mathscr{P}(\mathbb{N})$
- A **representation scheme** for a set $\mathcal{O}$ is a pair of functions E, D where $E : \mathcal{O} \to \{0,1\}^*$ and $D : \{0,1\}^* \to \mathcal{O}$ such that $D \circ E = I_{\mathcal{O}}$
- For strings $y, y' \in \{0,1\}^*$, y is a **prefix** of y' if $\exists y'' \in \{0,1\} : yy'' = y'$
- $E : \mathcal{O} \to \{0,1\}^*$ is **prefix-free** if $\forall o, o' \in \mathcal{O} : E(o)$ is not prefix of $E(o')$

- Prefix-free encoding $\Longrightarrow$ tuple encoding: $E'((o_1,\ldots,o_k)) = E(o_1)\ldots E(o_k)$
- For every encoding $E:\mathcal{O}\to\{0,1\}^*$ there exists prefix-free encoding E'
  - Let S be a prefix-free encoding of $\mathbb{N}$, then $E'(o) = S(|E(o)|)E(o)$
- Computational tasks are boolean functions up to representation
  - *Computational tasks are mathematical objects*
- *Algorithms are physically realizable specifications which compute functions*
  - Algorithms may be specified irrespective of physical specification (i.e. as mathematical objects whose existence manipulation are physically realizable), but elementary operations (if, for,…,) must be inherently physically realizable

Chapter 3: Defining computation

- A **boolean circuit** with n inputs, m outputs, and s gates is a labeled DAG $G=(V,E)$ with $s+n$ vertices such that:
  - Exactly n input vertices *have no in-neighbors* labeled $x[0]\ldots x[n-1]$.
    - Each input has *at least one out-neighbor*
  - Other s vertices are gates allowing *parallel edges* (e.g.
  - Exactly m output gates with no out-neighbors are labeled $Y[0]\ldots Y[m-1]$
- A s-line **straight-line program** is a list of tuples $L = \left((i_0,i_1,i_2)\in\mathbb{N}^3\right)$ corresponding to the sequence of instructions $x_{i_0} = \mathrm{NAND}(x_{i_1},x_{i_2})$
- Introducing $\mathrm{NAND}(a,b) = \neg(a=b=1) = \mathrm{NOT}(\mathrm{AND}(a,b))$
- Boolean circuits $\Longleftrightarrow$ Straight-line programs
  - Straight-forward conversion
- Two sets of gates A,B are **equivalent** $A\cong B$ if they compute the same set of functions
  - $\cong$ is an *equivalence relation* (reflexive, symmetric, transitive)

Chapter 4: Syntactic sugar and computing every function

- **Syntactic sugar**: f is computable by the set of functions $S \iff S\cong S\cup\{f\}$
  - $\{\mathrm{NAND}\}\cong\{\mathrm{AND},\mathrm{OR},\mathrm{NOT}\}$
    - $\mathrm{NOT}(a)=\mathrm{NAND}(a,a)$, $\mathrm{AND}(a,b)=\mathrm{NOT}(\mathrm{NAND}(a,b))$
  - $\{\mathrm{if},\mathrm{NAND}\}\cong\{\mathrm{NAND}\}$: $\mathrm{if}(a,b,c) = a\,?\,b:c = (a\wedge b)\vee(\neg a\wedge c)$
- Define $\mathrm{Lookup}_k:\{0,1\}^{2^k+k}\to\{0,1\}$ so $\forall x\in\{0,1\}^{2^k}, i\in\{0,1\}^k, \mathrm{Lookup}(x,i)=x_i$
  - Lemma: Exists $O(2^k)$-sized circuit which computes Lookup_k

- Induction: base step $\text{Lookup}_1(x_0, x_1, i_0) = \text{if}(\neg i_0, x_0, x_1)$ and
  $\text{Lookup}_{k+1} = \text{if}(\neg i_0, \text{Lookup}_k(x_0, \ldots, x_{2^k-1}, i_1, \ldots, i_k), \text{Lookup}_k(x_{2^k}, \ldots, x_{2^{k+1}-1}, i_1, \ldots, i_k))$
- Theorem: **boolean circuits compute every finite function**
  - Proof: $f(y_0, \ldots, y_{n-1}) = \text{Lookup}_n(\ldots [x_i = f(i)] \ldots, y_0, \ldots, y_{n-1})$
    - Corollary: $|\,\text{SIZE}_{n,m}(10mn \cdot 2^n)\,| = 2^{2^n}$. We can do better: $|\,\text{SIZE}_{n,m}(10 \cdot 2^n/n)\,| = 2^{2^n}$
  - Alternative proof: Inductively assume every $f : \{0,1\}^n \to \{0,1\}$ is computable, then $\forall f' : \{0,1\}^{n+1} \to \{0,1\}$. Let $f_1'(x_0, \ldots, x_{n-1}) = f'(1, x_0, .., x_{n-1})$ and $f_0'(x_0, \ldots, x_{n-1}) = f'(0, x_0, \ldots, x_{n-1})$: both are computable, and $f(x_0, \ldots, x_n) = \big(\neg x_0 \wedge f_0'(x_1, \ldots, x_n)\big) \vee \big(x_1 \wedge f_1'(x_1, \ldots, x_n)\big)$
    - Corresponding bound is $O(m2^n)$
- $\forall n, m, s \in \mathbb{N} : \text{SIZE}_{n,m}(s) = \{f : \{0,1\}^n \to \{0,1\}^m \mid f \text{ computable with } \leq s \text{ gates}\}$ is the **size class** of functions with n inputs, m outputs and s gates.
  - $\text{SIZE}_n(s) \equiv \text{SIZE}_{n,1}(S)$, and $\text{SIZE}(s) \equiv \bigcup_{n,m} \text{SIZE}_{n,m}(s)$

Chapter 5: Code as Data, Data as Code

- *Representation of programs may be used as inputs to other programs*
- $\forall f \in \mathrm{SIZE}(s), \exists P$ computing f such that string representation of P has length $O(s \log s)$
    - Given s-line straight-line program L, there are at most $3s$ variables. Representing each variable takes at most $O(\log s)$ characters, so we need $O(s \log s)$ to represent L
- Program size bound on number of computable functions: $|\mathrm{SIZE}_{n,m}(s)| \leq 2^{O(s \log s)}$
    - Define $\phi : \mathrm{SIZE}_{\mathrm{n,m}}(s) \rightarrow \mathbb{N}^{3s}$ so that given f, $\phi(f)$ is the smallest *size-s* straight-line program which computes f, ϕ is injective, and $|\mathrm{SIZE}_{n,m}(s)| \leq |\mathrm{Im}(\phi)| \leq 2^{O(s \log s)}$
- Theorem: **maximum size of program computing arbitrary $f : \{0,1\}^n \rightarrow \{0,1\}$ is $\Theta(2^n/n)$**
    - Lower bound: $\exists \delta \in \mathbb{R}_+, N_0 \in \mathbb{N} : \forall n \geq N_0, \left|\{0,1\}^{\{0,1\}^n}\right| \geq \mathrm{SIZE}_n\left(\delta 2^n/n\right)$
        - Let $0 < \delta < 1$, then substitute $s \mapsto \delta 2^n/n$ and unrolling the definition of O yields $\exists c > 0, N_0 \in \mathbb{N} : \forall n > N_0 : \mathrm{SIZE}_n\left(\delta 2^n/n\right) \leq 2^{c\delta 2^n/n \cdot \log(\delta 2^n/n))} \leq 2^{c\delta 2^n}$. Choose $\delta < 1/c$ to yield $\left|\mathrm{SIZE}_n\left(\delta 2^n/n\right)\right| \leq 2^{2^n}$
    - Upper bound: we showed that any $f : \{0,1\}^n \rightarrow \{0,1\}$ computable using $O(n2^n)$—the best bound $O(2^n/n)$ suffices to provide the upper bound
- **Size Hierarchy Theorem:** for sufficiently large n and $10n < s < 0.1 \cdot 2^n/n$, $\mathrm{SIZE}_n(s) \subsetneq \mathrm{SIZE}(s + 10n)$
    - Let $f^* : \{0,1\}^n \rightarrow \{0,1\}$ be the function such that $f^* \notin \mathrm{SIZE}_n(0.1 \cdot 2^n/n)$. Consider a class of functions $\{f_i \mid i \in [2^n - 1]\}$ such that $f_i(x) = (x < i)?f^*(x) : 0$
        - f_0 is constant 0, while $f_{2^n-1} = f^*$
        - Now $f_{i+1}(x) = (x = i + 1)?f^*(x) : f_i(x)$ and recursion incurs $O(n)$ overhead
- An **interpreter** $\mathrm{EVAL}_{s,n,m}(px) = P(x)$, where $|p| = O(s \log s), |x| = n$, and p is the string representing program P, is a single *function* which evaluates *arbitrary programs of certain size*
    - Efficient computation of interpreters: computing $\mathrm{EVAL}_{s,n,m} : \{0,1\}^{O(s \log s)+n}$ takes $O(s^2 \log s)$ lines
        - Proof: there are s lines and $O(s)$ variables. Consider a variable *lookup* table; iteratively reference and update the table for each of s lines. Total complexity is $s^2 \log s$
        - *Lookup* is $O(s)$: recall lookup takes linear time w.r.t. size of table
        - *Update* is $O(s \log s)$: s to iterate over the elements and $O(\log s)$ to check index equality

Chapter 6: DFA and regular expressions

- We now consider computation over *unbounded* inputs $\{0,1\}^*$
  - We may compute $f : \{0,1\}^* \to \{0,1\}$ via an *infinite collection* of circuits $\{C_i$ computing $f_i\}$
  - But we want a finitely describable process!
- **Deterministic Finite Automaton** $M = (T, \mathcal{S})$ with C states
  - Transition function $T : [C] \times \{0,1\} \to [C]$
  - $\mathcal{S} \subseteq [C]$ are accepting states
  - Computing $M(x)$: initially $s_0 = 0$ and $\forall i \in \{1,\ldots,|x|\} : s_i \mapsto T(s_i, x_i)$; output $s_{|x|} \in^? \mathcal{S}$
- Lemma: single-pass constant-memory algorithms are DFA-computable
  - Proof: $[C]$ denotes all *possible configurations of memory* (for n-bit memory $C = 2^n - 1$)
- Number of DFA-computable functions are countable
  - Characterizations of DFAs are finite $\Longrightarrow$ set of all DFAs is countable
  - Corollary: exists non-DFA-computable functions
- A **regular expression** e over alphabet Σ is a string over $\Sigma \cup \{(,),|,*,\varnothing,\text{“”}\}$ either:
  - $e \in \Sigma$: single literal in alphabet
  - $e = e'|e''$: logical *or*
  - $e = e'e''$: concatenation of regular expressions
  - $e = e'*$: repetition of expressions
  - $e = \varnothing$: any expression; $e =$ “”: no expression
- The function $\Phi_e : \Sigma^* \to \{0,1\}$ evaluates whether its input **match regular expression** e
- A function f is **regular** if exists regular expression $e : \Phi_e = f$
- Single-pass constant-memory algorithm computes regular functions:
  - Time-complexity: for each reg-ex e and character σ there exists $e[\sigma] : \Phi_e(\sigma s) = \Phi_{e[\sigma]}(s)$
    - Each recursion incurs $O(1)$ overhead in computing $e[\sigma]$
  - Memory-complexity: Memoization of restrictions is $O(|e|)$ and constant w.r.t. n
- Lemma: Regular functions are DFA-computable
  - Corollary: $\exists e' : \Phi_{e'} = \neg \circ \Phi_e$: flip the accepting gates of DFA computing Φ_e
  - **Closure of regular expressions:** $\forall f : \{0,1\}^n \to \{0,1\}, \exists e' : \Phi_{e'} = f \circ ((\Phi_{e1} \ldots \Phi_{en}))$
    - Regular expressions closed under *not* and *or*
    - Corollary: $\text{REGEQ} : \text{REGEQ}(e, e') \equiv (\Phi_e = \Phi_{e'})$ *is computable*
- Lemma: DFA-computable functions are regular
  - Let $A = (T, \mathcal{S})$: $\forall v, w \in [C]$, define $F_{v,w} : \{0,1\}^* \to \{0,1\}$ such that $F_{v,w}(x) = 1 \iff$ DFA starting from v will reach w upon input x

- Define $F^t_{v,w} : \forall t \in [C], F^t_{v,w}(x) = 1 \iff F_{v,w} = 1$ *and intermediate states* $\subseteq [t]$
  - v, w *are not counted as intermediate!*
- $F^0_{v,w}$ is regular: $\varnothing, 0, 1, 0 \mid 1$ depending on $T(v, (x \in \{0,1\})) \stackrel{?}{=} w$
- Inductive step: assume $\forall v', w' \in [C], F^t_{v',w'} = \Phi_{R^t_{v',w'}}$, the newly introduced state is t
  - Then the shortest path $v \to w$ is either: … $\implies R^{t+1}_{v,w} = R^t_{v,w} \mid \left(R^t_{v,t}\left(R^t_{t,t}\right)^* R^t_{t,w}\right)$
    - Contained in $[t]$: path corresponds to $R^t_{v,w}$
    - $v \to t \to w$: path corresponds to $R^t_{v,t}\left(R^t_{t,t}\right)^* R^t_{t,w}$
- Note that $F^C_{v,w} = F_{v,w}, A(x) = \bigcup_{s \in \mathcal{S}} F_{0,s}(x)$. Then $A = \Phi_{\bigcup_{s \in \mathcal{S}} R^C_{0,s}}$

- Theorem: $f : \{0,1\}^* \to \{0,1\}$ **DFA-computable** $\iff$ f **regular**
- **Pumping Lemma:** every regular function must have a length threshold above which a true-valued input string invok es the $*$ operator
  - $\forall f = \Phi_e, \exists p \in \mathbb{N} : \exists x, y, z \in \{0,1\}^* : |y| \geq 1, |xy| \leq p, \forall k \in \mathbb{N}, f(xy^kz) = 1$
  - Using at disproving a function is regular using contradiction.

Chapter 7: Turing machines

- What happens to our description of computation when time is no longer canonical?
- Examine our requirements for an **algorithm**:
  - Finitely enumerable set of elementary operations (elementary / physically realizable?)
  - Potentially *unlimited* working memory for inputs and intermediate results
  - Pointer to modifiable portion of memory
  - Instructions to begin, repeat, and stop
- Alan Turing, 1936. An intuitive mathematical formalism of computation
  - A person writing on scratch papers - up to binary representation of brain and scrap
- A **Turing Machine** with k *states,* alphabet $\Sigma \supseteq \{0,1, \triangleright ,\varnothing\}$, and transition function $\delta_M : [k] \times \Sigma \to [k] \times \Sigma \times \{L, R, S, H\}$, on input x, outputs $M(x)$ via the process:
  - Initialize $T = (\triangleright , x_0, \ldots, x_{n-1}, \varnothing, \varnothing \ldots); i = s = 0$
  - Repeat:
    - $(s', \sigma', D) = \delta_M(s, T[i])$
      - Use T to compute new internal state, memory content, and direction of header
    - $s \mapsto s, T[i] \mapsto \sigma'$
    - $D = \begin{bmatrix} L \\ R \\ S \end{bmatrix} \implies i \mapsto \begin{bmatrix} i-1 \\ i+1 \\ i \end{bmatrix}; D = H \implies HALT$
  - Upon $HALT$, $M(x) = T_{n<\min\{i : T_i \neq \varnothing, T_{i+1} = \varnothing\}}$, else $M(x) = \perp$
  - Remarks: internal state space $[k]$ is finite!
- Given Turing machine over Σ and internal state space $[k]$, a **configuration** of M is a string $\alpha \in \bar{\Sigma}^*$ where $\bar{\Sigma} = \Sigma \times (\{ \cdot \} \cup [k])$ so that *there is exactly one* $i_0 : \alpha_{i_0} = (\sigma \in \Sigma, s \in [k])$ otherwise $i \neq i_0 \iff \alpha_i = (\sigma', \cdot)$
  - " $\cdot$ " are the placeholders. $T[i] = \alpha_{i,0}$ and head is at i_0
- Lemma: Turing machines have finite transition functions
  - Let $\Phi_M : \bar{\Sigma}^* \to \bar{\Sigma}^*$ denote transition between subsequent configurations of a Turing machine, then there exists $\psi_M : \bar{\Sigma}^3 \to \bar{\Sigma}$ satisfying $\forall i, \psi_M(\alpha_{i-1}, \alpha_i, \alpha_{i+1}) = \Phi_M(\alpha)_i$
  - Configuration changes are locally restricted to the immediate vicinity of tape head.
    - Tape change only possible at head position, and head indicator only possible in vicinity
- A function $f : \{0,1\}^* \to \{0,1\}^*$ is **computable** if there exists a Turing machine computing its restriction onto defined domain.
  - Unlike circuits, Turing machines may fail to halt on *some* inputs
  - $\mathbf{R} \equiv$ the **set of all computable functions**
- Introducing **NAND-TM** language

  - Broad-brush idea: NAND-TM $\equiv$ NAND-CIRC + loops + arrays
  - A NAND-TM program P is a sequence of lines $a = \text{NAND}(b, c)$ ending with $\text{MOD\_JUMP}(a, b)$. Variables may be array variables $X[i]$, $Y[i]$, $X_{\text{blank}}[i]$, $Y_{\text{blank}}[i]$, or additional scalar or array variables.
  - Computational process $P(x)$
    - $X[i] = x_i$, $X_{\text{blank}}[i] = (i \geq |x|)$, all other variables and i are 0
    - Execute until $\text{MOD\_JUMP(a, b)} = \{S, R, L, H\}$ and execute corresponding action
  - NAND-TM $\cong$ TM: a theorem by design. Technical details omitted.
- **GOTO** as syntactic sugar:
  - Introduce program counter to indicate which lines should be executed; natural execution increments the program counter; GOTO simply corresponds to changing the counter
  - GOTO unlocks complex loop constructs such as *WHILE, FOR, etc*
- **Uniformity**: computing task across different input lengths with the same instructions.
  - TM's and DFA are uniform, while *collection* of circuits are not
  - **Uniformity implies truly universal interpreters**

Chapter 8: Equivalent models of uniform computation

- **RAM Machine**
  - Memory array M of unbounded size, each cell stores a *word* $\in \{0,1\}^w \cong [2^w]$
  - *Constant* number of **registers** $r_0, \ldots, r_{k-1}$ each containing a single word
  - Operations:
    - Data movement: $r_i := \mathrm{M}_\mathrm{j}, \mathrm{M}_i := r_j$
    - Computations on registers
- **NAND-RAM**
  - Variables are non-negative integer values $\in [0,T-1]$: T is number of executed steps
  - *Integer-indexed access* to integer arrays; basic arithmetic operations; loop constructs
- NAND-RAM $\cong$ NAND-TM
  - Gory details (Pg 290?)
  - Reference via $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}; f(x,y) = (x+y)(x+y+1)/2 + x$
- Let $\mathcal{F}$ be the *set of all partial functions* $\{0,1\}^* \to \{0,1\}^*$. A **computational model** is a map $\mathcal{M} : \{0,1\}^* \to \mathcal{F}$. A program $P \in \{0,1\}^*$ $\mathcal{M}$-computes $F \in \mathcal{F} \iff \mathcal{M}(P) = F$
  - Note that *algorithms are finite strings*, so $\mathcal{M}$ cannot be surjective!
  - $\mathcal{M}$ is **Turing complete** if there exists computable map $\phi : \{0,1\}^* \to \{0,1\}^*$ such that for every TM N as a string computing F, $(M \circ \phi)(N) = F$
    - i.e. computes all Turing-computable functions
  - $\mathcal{M}$ is **Turing equivalent** if it is Turing complete and exists computable map from each P in $\mathcal{M}$ to the Turing machine which computing $\mathcal{M}(P)$
- A one-dimensional **cellular automaton** over $\Sigma \supset \{\varnothing\}$ is described by transition rule $r : \Sigma^3 \to \Sigma$ satisfying $r(\varnothing, \varnothing, \varnothing) = \varnothing$. A *configuration* is a function $A : \mathbb{Z} \to \Sigma$, and computation proceeds via $A \mapsto A' : A'(i) \equiv \Sigma(A(i-1), A(i), A(i+1))$
  - A **configuration is finite** if only finitely many entries are nonzero
- **Theorem: One-dimensional automata are Turing complete**
  - For each M over Σ and state space $[k]$, take automaton over $\Sigma' \supset \{\varnothing = (\varnothing, \cdot)\} \cup \bar{\Sigma}$ with transition function $A = \psi_M$
- **λ-calculus:** Alonzo Church
  - *Anonymous definition of functions* and *functions as first-class objects*
    - $f(x) = x \times x, f(3) \leftrightarrow (\lambda x . x \times x)3$
  - Creating multi-argument functions via **currying**
    - $\lambda x . (\lambda x . x + y)$ corresponds to $g(x,y) = x + y$: taking the first argument y creates a partial function $\lambda x . x + y$

- A λ **expression** is either a single variable identifier or expression of the following forms:
  - $e = (e\ e')$: Apply e to e'. Application is left-associative: $f\,g\,h = (f\,g)\,h$
  - *Abstraction:* $e = \lambda x\,.\,e'$
- λ expressions may be evaluated by repeatedly applying the following rules:
  - β **reduction:** $(\lambda x\,.\,y)\,z \to y[x \mapsto z]$ (replace all occurrences of x in y by z)
  - α **conversion**: $\lambda x\,.\,z \iff \lambda y\,.\,z[x \mapsto y]$
  - Evaluation protocols:
    - *Call by name (lazy evaluation)*: $\text{eval}\,[(\lambda x\,.\,y)\,z] = \text{eval}(\,y[x \mapsto z]\,)$
    - *Call by value (eager evaluation):* $\text{eval}[(\lambda x\,.\,y)\,z] = \text{eval}(\,y[x \mapsto \text{eval}(z)]\,)$
  - Evaluation may encounter infinite loops such as $(\lambda x\,.\,x\,x)(\lambda x\,.\,x\,x)$
- Syntactic λ sugar: **Church encoding**
  - $1 \equiv \lambda x\,.\,(\lambda y\,.\,x)$, $0 \equiv \lambda x\,.\,(\lambda y\,.\,y)$. Then $\text{if}(\text{cond}, a, b) \equiv \text{cond}\; a\; b$
  - $\text{pair}\, x\ y \equiv \lambda x\,.\,\lambda y\,.\,(\lambda g\,.\,g x y)$ satisfies $(\text{pair}\, x\ y)\, z = z\ x\ y$
    - $\text{head}\, p \equiv p\, 1$ satisfies $\text{head}\,(\text{pair}\, x\ y) = x$. Similarly $\text{tail}\, p = p\, 0$
  - $\text{nil} \equiv \lambda x.1$. Define $\text{ispairnil} \equiv \lambda p\,.\,p\,(\lambda x\,.\,\lambda y.0)$. Where $\text{ispairnil}\, p = (p \stackrel{?}{=} \text{nil})$
  - Encoding numerals: Let $i + 1 = \text{pair}\, 1\, i$, then $i - 1 = \text{tail}\, i$
  - **Y combinator:** Define Y s.t. $Y f$ is a fixed point of $f \iff Y f = f\,(Y f)$
    - $Y \equiv \lambda f.\,(\lambda x\,.\,f\,(x\ x))\,(\lambda x\,.\,f\,(x\ x))$. Then $Y f = f\,((\lambda x\,.\,f\,(x\ x))\,(\lambda x\,.\,f\,(x\ x))) = f\,(Y f)$
    - Define recursive functions f as $F(f, x)$ where self-referencing calls are replaced by f
      - e.g. $g(x) \equiv (x = 0)?1 : xg(x-1) \leftrightarrow F \equiv [\lambda f.\,\lambda x\,.\,(x = 0)?1 : x \cdot f\,(x-1)]$
      - Then $g(2) = Y\,F\,2 = F\,(Y\,F)\,2 = 2 \cdot (Y\,F\,1) = 2 \cdot 1 = 2$
  - Then $\text{reduce} = \lambda g\,.\,\lambda L\,.\,\text{if}\,(\text{isempty}\,(\text{tail}\,L))\,(\text{tail}\,L)\,[g\,(\text{head}\,L)\,(\text{reduce}\,(\text{tail}\,L))]$
    - $\text{map} = \lambda g\,.\,\text{reduce}\,(\lambda x\,.\,\lambda y\,.\,\text{pair}\,(g\,x)\,y)$
    - Y-combinator $\to$ reduce $\to$ map $\to$ filter ...
- λ-expression e computes F if $\forall x \in \{0,1\}^*, e\langle x_0, \ldots, x_{n-1}, \perp\rangle \cong \langle y_0, \ldots, y_{m-1}, \perp\rangle$
  - Inputs and outputs are λ-lists (think oCAML!)
- **λ calculus is Turing-equivalent:** Simulating TM
  - λ-calculus computes NAND thus every finite function, including TM's transition function
  - λ-calculus also supports conditional-dependent recursion.

Chapter 9: Universality and Uncomputability

- Universal Turing Machine Interpreter
  - Exists TM U s.t. on every string M representing a TM and $x \in \{0,1\}^*$, $U(Mx) = M(x)$
  - Representation of TM with k states $\Sigma = \{\sigma_0, \ldots, \sigma_{l-1}\}$, and (finite) δ_M is trivial
  - Trivial proof since we can finitely characterize the operation of TM
  - Remark: every $s \in \{0,1\}^*$ can correspond to a (can-be-trivial) TM!
- Existence of uncomputable functions
  - One way: Cardinality argument
  - Let U be the universal TM, then $F^*(x) = 1 - U(x, x)$ (if no-halt then 0) is uncomputable
- **Halting problem**: $\mathrm{HALT} : \{0,1\}^* \to \{0,1\}^*$ s.t. $\forall M \in \{0,1\}^*, \mathrm{HALT}(M, x)$ computes whether TM M halts on input x. HALT is uncomputable:
  - Assume $\exists M_{\mathrm{HALT}}$, then $M_{F^*}(x) = M_{\mathrm{HALT}}(x, x) ? 1 - U(x, x) : 0$ computes F^* if M_{HALT} computes HALT
  - Assume $\exists M_{\mathrm{HALT}}$. Define $P : P(x) = M_{\mathrm{HALT}}(x, x) ? \mathrm{nohalt} : \mathrm{halt}$
    - $M_{\mathrm{HALT}}(P, P) = 1 \implies M_{\mathrm{HALT}}(P, P) = 0$ and vice versa -> contradiction
- Proof by reduction: f uncomputable $\iff$ (HALT computable $\implies$ f computable)
- Variants of the halting problem
  - $\mathrm{HALT0}$ problem: $\mathrm{HALT0}(M) = \left(M(0) \overset{?}{=} \perp\right)$
  - $\mathrm{HALT0}$ computable $\implies$ HALT computable: given M, x, define $M_x : M_x(0) = U(M, x)$; output $\mathrm{HALT}(M, x) = P(M_x, 0)$
- M, M' **functionally equivalent** $M \cong M' \iff \forall x \in \{0,1\}^*, M(x) = M'(x)$ *including halts*!
- $F : \{0,1\}^* \to \{0,1\}^*$ is **semantic** if $\forall M, M' \in \mathbb{N} : M \cong M' \implies F(M) = F(M')$
  - In other words, F semantic $\iff$ exists extension $\tilde{F} : \{0,1\}^*/\cong \to \{0,1\}^*$
- **Rice's theorem: Nontrivial semantic functions are uncomputable**
  - F nontrivial semantic $\implies \exists M_0 : F(M_0) \neq F(\mathrm{INF})$ where INF is the TM that never halts
    - Without loss of generality let $F(\mathrm{INF}) = 1, F(M_0) = 0$
  - Assuming access to A computing F, then P below computes $\mathrm{HALT0}$:
    - Given $N \in \{0,1\}^*$, construct TM $G(N)$ which given x executes:
      - Compute $N(0)$
      - Return $M_0(x)$
    - Return $F(G(N))$
  - $N(0) = \perp \iff F(G(N)) = F(\mathrm{INF}) = 1$

Chapter 10: Context-free Grammar

- The more expressive a computational model is, the less semantic questions we can answer
- Given alphabet Σ, a **context-free grammar (CFG)** over Σ is a triple (V, R, s) s.t.
  - V denote variables, disjoint from Σ: $V \cap \Sigma = \emptyset$. Initial variable $s \in V$
  - R are rules: $\forall r \in R, r = (v \in V, z \in (\Sigma \cup V)^*) \leftrightarrow v \Rightarrow z \implies z$ can be derived from v
  - Remarks: $v \in V$ are like "types", $s \in V$ is the type to interpret the input in, and $r \in R$ denotes how types can possibly be composed of each other
- Example: CFG of arithmetic expressions
  - $\Sigma = \{(,), +, -, \times, \div, 0,1,2,3,4,5,6,7,8\}$
  - $V = \{\text{expr}, \text{number}, \text{digit}, \text{operation}\}, s = \text{expr}$
  - Rules: $\text{operation} \Rightarrow + \mid - \mid \times \mid \div, \text{digit} \Rightarrow 0 \mid \ldots \mid 9, \text{number} \Rightarrow \text{digit} \mid \text{digit number}$, $\text{expr} \Rightarrow \text{number} \mid \text{expr operation expr} \mid (\text{expr})$
- Another example: CFG of matching parentheses: $\text{match} \Rightarrow \emptyset \mid \text{match match} \mid (\text{match})$
- Given CFG $G = (V, R, s)$ over Σ and $\alpha, \beta \in (\Sigma \cup V)^*$, β **can be derived in one step** from α, $\alpha \Rightarrow_G \beta \iff \exists (v \Rightarrow z) \in R : \beta = \alpha[v \mapsto z]$. Derivability in general denoted by $\alpha \Rightarrow_G^* \beta$
- $x \in \Sigma^*$ is **matched** by G if x can be derived from s: $s \Rightarrow_G^* x$.
- CFG as computational model: G **computes** $\Phi_G(x) = (s \Rightarrow_G^* x?)$
  - $F : \Sigma^* \to \{0,1\}$ is **context free** if $\exists G : F = \Phi_G$
- Context-free grammar is computable
  - Reduce to **Chomsky normal form** $u \Rightarrow vw$: add auxiliary variables if necessary.
  - Simple matching over possible partitions and rules
- CFG is more expressive than reg-ex
  - Induction over length of e: simple base case.
  - $e = e'e'', e = e' \mid e'', e = (e')^*$ corresponds to simply adding new rules
- CFG for palindrome: $p \Rightarrow \emptyset \mid 0\,p\,0 \mid 1\,p\,1$
  - Non-palindrome: $p \Rightarrow \emptyset \mid 0\,p\,0 \mid 1\,p\,1; d = 0\,p\,1 \mid 1\,p\,0;\ n = d \mid 0\,n \mid 1\,n \mid n\,0 \mid n\,1$
- Context-free pumping lemma
  - A long enough matching string must have repeated variables in its derivation
  - $\forall G + (V, R, s), \exists n_0, n_1 \in \mathbb{N} : \forall x \in \Sigma^*, |x| > n_0, \Phi_G(x) = 1 \implies$ existence of partition $x = abcde : |b| + |c| + |d| \le n_1, |b| + |d| > 1, \forall k \in \mathbb{N}, \Phi_G(ab^kcd^ke) = 1$
  - Assume a long-enough string, then by pigeon-hole principle there must be repeated derivation $v \Rightarrow bvd$ for which $v \Rightarrow c$
  - Example: $\text{EQ} : \{0,1,;\}^* \to \{0,1\}, \text{EQ}(y) = (\exists x : y = x; x)$ cannot be pumped

Chapter 11: Proofs and Computation, Gödel's Incompleteness Theorem

- **Mathematical statements** are strings $s \in \{0,1\}^*$ whose truth depend on (defined) properties of abstract objects, and not on empirical facts
  - Properties such as "will halt," "is prime," etc.
- Given $\mathcal{T} \subseteq \{0,1\}^*$ be the set of mathematical statements which are considered true. An *algorithm* V constitutes a **proof system for** $\mathcal{T}$ if it is:
  - **Sound**: $\forall x \notin \mathcal{T}, w \in \{0,1\}^*, V(x,w) = 0$. Proofs cannot prove untrue statements
  - **Effective**: $\forall x, w \in \{0,1\}^*$, V halts with an output of 0 or 1
  - V is **complete** if every statement is provable
  - Remarks:
    - Truthfulness of statements are given a priori *independent of the proof system*
    - A proof system is a classifier of true statements: *soundness* requires that it makes no false positive mistakes, *effectiveness* requires it produces output *given a candidate proof*.
    - A true statement $x \in \mathcal{T}$ may be unprovable w.r.t. V if $\forall w \in \{0,1\}^*, V(x,w) = 0$
    - A complete proof system must produce the truth status of any statement in finite time
- ***Completely* provable $\implies$ computable**: given $\mathcal{T} \subseteq \{0,1\}^*$, existence of a *complete* proof system for $\mathcal{T} \implies \Phi_{\mathcal{T}}(x) = (x \overset{?}{\in} \mathcal{T})$ is computable
  - Assume $\forall x \in \{0,1\}^*, \exists \neg x \in \{0,1\}^* : x \in \mathcal{T} \iff \neg x \notin \mathcal{T}$ ($\neg x$ is the logical negation)
    - Our assumption seems sound, but must it hold for every set of true statements?
  - Let V be a complete proof system for $\mathcal{T}$. Given each x, look for proofs of $x, \neg x$—our search process must halt because one of them must be true and V is complete.
- Given any proof system V, we can design a true statement x^* that is not provable by V:
  - x^* is true $\iff$ x^* does not have a proof in V
  - Fixed-point trick to solve the problem of self-reference (arithmetic is better than syntax here)
  - Why would we wish to prescribe arbitrary truth statements *given a proof system*? (Kind of like change the labels of our data after being given a classifier)
- **Quantified Integer Statements (QIS)** are statements with no unbound variables which only uses integers, variables, operators (+, -, =, *, >, <), logical operations AON, and $\exists_{x\in\mathbb{N}}, \forall_{y\in\mathbb{N}}$
  - $\mathrm{QIS} \subseteq \{0,1\}^*$, and we can similarly use syntactic sugar
- **Quantified Mixed Statements (QMS)** is QIS additionally allowing $\forall_{a\in\{0,1\}^*}, \exists_{a\in\{0,1\}^*}, a_i, |a|$
- **Theorem: Φ_{QMS} is uncomputable**
  - Proof idea: Φ_{QMS} computable $\implies$ $\mathrm{HALT0}$ computable
  - A program halts on zero iff a sequence of configurations $H = (\alpha_0, \ldots, \alpha_{T-1})$ s.t. α_0 is a valid starting configuration with input 0, α_{T-1} is halting configuration, and $\alpha_{t+1} = \mathrm{NEXT}(\alpha_t)$

- Concretely, $\text{HALT0}(M) = \Phi_{\text{QMS}}(\exists_{H \in \{0,1\}^*} H$ encodes halting sequence with input zero)
- Define QMS for $\text{NEXT} : \Sigma^* \to \Sigma^*$: it is produced by convolving a local function $\Sigma^3 \to \Sigma$
  - Define QMS over local function, and use universal quantifier to mimic convolution
- Similarly, use universal quantifier to check valid transition over sequence, and finally validate beginning and ending configurations.

- Theorem: Φ_{QIS} computable $\implies$ Φ_{QMS} computable: reducing QMS to QIS
  - Encode every string $x \in \{0,1\}^*$ by $(X, |x|) \in \mathbb{N}^2$ such that $\exists \text{QIS coord} : \text{coord}(X, i) = i < |X| \,?(x_i \stackrel{?}{=} 1) : 0$
    - Then $\forall_{x \in \{0,1\}^*} \mapsto \forall_{X \in \mathbb{N}} \forall_{n \in \mathbb{N}}, |x| \mapsto n, x_i \mapsto \text{coord}(X, i)$
  - **Constructible prime sequence**: exists primes p_i s.t. $\exists \text{QIS PSEQ}(p, i) = (p \stackrel{?}{=} p_i)$
    - Then $X = \Pi_{x_i = 1} p_i, \text{coord}(X, i) = \exists_{p \in \mathbb{N}} \text{PSEQ}(p, i) \wedge (X \,\%\, p = 0)$
  - Corollary: Φ_{QMS} is uncomputable.
  - Corollary: **There is no complete proof system for** $\text{QMS} \subseteq \{0,1\}^*$
    - i.e. we cannot prove every QMS.