Results from Chapter 7
Theorem 7.1

Equivalence of 1-way infinite and 2-way infinite tape machines

Add D to the state when using the lower track

- Picture:

  Y₈ Y₇ Y₆ Y₅ Y₄ Y₃ Y₂ Y₁ X₁ X₂ X₃ X₄ X₅ X₆ X₇ X₈ X₉ X₁

  q

  original head position

- Simulation:

  X₁ X₂ X₃ X₄ X₅ X₆ X₇ X₈ X₉ X₁₀ X

  $ Y₁ Y₂ Y₃ Y₄ Y₅ Y₆ Y₇ Y₈ Y₉ Y

  q,D

Results from Chapter 7
Theorem 7.2

Equivalence of multi-tape and one tape machines
Simulate $n$ tapes with $2n$ tracks

- The $X$’s represent head positions

<table>
<thead>
<tr>
<th>Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8 Y9 Y10 Y11 Y12 Y13 Y14 Y15 Y16</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 Z11 Z12 Z13 Z14 Z15 Z16 Z</td>
<td>X</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Y1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15 V16</td>
<td>X</td>
</tr>
</tbody>
</table>

Results from Chapter 7

Lemma 7.3

Simulation using a 2-stack machine

Comparing the moves

<table>
<thead>
<tr>
<th>Y8 Y7 Y6 Y5 Y4 Y3 Y2 Y1 X1 X2 X3 X4 X5 X6 X7 X8 X9 X10</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y8 Y7 Y6 Y5 Y4 Y3 Y2 Y1’ X1 X2 X3 X4 X5 X6 X7 X8 X9 X10</td>
<td>q’</td>
</tr>
</tbody>
</table>

- Tape:

- Stacks:

Results from Chapter 7

Lemma 7.4, Theorem 7.9

Simulation using counter machines
Four counter stacks

- With 2 stacks, you can work with the representation of $Z_i_1 Z_i_2 \ldots Z_i_m$ as $j = i_m + k i_{m-1} + k^2 i_{m-2} + \ldots + k^{m-1} i_1$, which is represented on one stack as $B i Z$

One number can represent four numbers

- The coding so far will give a 4-counter machine—a 2-stack machine is simulated by a 4-counter machine
- Another step simulates a 4-counter machine by a 2-counter machine
- Here the counts $i, j, k, m$ on the 4 counter stacks is simulated by the number $2^i 3^j 5^k 7^m$ on one stack (the textbook gives a more detailed outline)

We used it several times in Chapter 8

- If $L \subseteq (0+1)^*$ and $L$ is recursively enumerable, then $L$ is accepted by a 1-tape TM with tape alphabet $\{0, 1, B\}$
- Main feature of the proof:
- Suppose the original tape alphabet has between $2^{k-1} + 1$ and $2^k$ symbols (e.g. $k$ is 4 for alphabets with 9, 10, 11, 12, 13, 14, 15 and 16 symbols)
- Then $k$ bits encode EVERY symbol,
It is just binary

- For example, 8 bits can encode any one of up to 256 different symbols
- To simulate a one-tape machine for $L$, a move of the simulator involves scanning across $k$ bits of data on the tape
- The control state can decode the binary representation, choose the transition and write the appropriate binary encoding on to the tape

prepare the input

- Note that at the beginning, the input is in binary: 0’s and 1’s
- That input has to be encoded using $k$ bits, i.e. the symbols have to be separated out
- The first symbol is given its $k$-bit encoding and all the others are moved $k-1$ positions to the right
- This process is repeated up to the blanks

Results from Chapter 7
Theorem 7.6

Random access machines

- A short section outlines the simulation of RAM machines:
- The tape contains 
  #0*v$_0$#1*v$_1$#10*v$_2$#11*v$_3$#100*v$_4$# ...
- The number between “#” and “*” is the memory address (v$_i$’s are binary)
- N tapes are used to contain the contents of N registers and one tape holds the address of the next instruction
Each cycle

- An execution cycle involves:
  - scan the tape for the address of the next instruction,
  - interpret the operation and its arguments (e.g. 10 bits of \( v_i \) are the op-code and the remaining bits are the address of the operand)
  - increment the location counter
  - execute the instruction

Results from Chapter 7
Lemmata 7.1, 7.2, Theorem 7.7

Generating recursively enumerable sets

Enumerators (generators)

- We have seen how a context-free grammar *generates* a context-free language and
- When we mentioned Chomsky’s hierarchy, we said that unrestricted grammars can generate recursively enumerable languages
- but Turing machines can generate languages also

Running the enumerator

- Enumerators have an output tape, where the TM writes individual strings in the languages and separates them with a “#”
- The language generated can be finite, in which case the TM could stop after printing the last string but it does not have to stop
- Otherwise, the TM runs for ever
Every string in $L$ (& only those) will be an output

- To be an enumerator, it is essential that the TM write every string in the language on to the tape eventually
- Further, ONLY strings from the language can ever appear on the output tape
- Theorem 7.7: The generated languages are exactly the recursively enumerable languages

Proof (Lemma 7.1)

- Given a generator $G$ for $L$, we build a recognizer TM for $L$ as follows:
- Given $w$ on the input tape, run $G$ and compare each output with $w$. If $w$ appears, halt and accept, otherwise keep running $G$
- This shows that if $L$ can be generated by an enumerator it is “recursively enumerable” (recognized by a TM)

Proof (2)

- Suppose, $L$ is recursively enumerable with a TM called $M$
- Add a tape with a counter $k = 1, 2, 3, 4, ...$
- Add two tapes to hold $i$ and $j = (k - i)$
- Add a tape, where we generate strings in canonical order, e.g. for $\Sigma = \{a,b,c\}$, the canonical order is:

Beginning of canonical sequence for $(a+b+c)^*$

- $\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, aac, aba, abb, abc, aca, acb, acc, baa, bab, bac, bba, bbb, bbc, bca, bcb, bcc, caa, cab, cac, cba, cbb, cbc, cca, ccb, ccc, aaaa, aaab, aaac, aaba, aabb, aabc, aaca, aacb, aacc, abaa, abab, abac, abba, abbb, abbc, abca, abcb, abcc, acaa, acab, acab, acac, acba, acbb, acbc, acca, accb, accc, ...
Proof (3)

- For each $i, j$, run $M$ on $w_i$ for exactly $j$ steps. If $M$ halts before the $j$-th step, skip $w_i$ and go on to the next string.
- If $M$ accepts $w_i$ in exactly $j$ steps, write it to the output tape.
- Only the strings of $L$ can appear on the output and every string in $L$ will appear on the output.

Recursive languages

- Lemma 7.2: The recursive languages are exactly the languages that are generated in canonical order.
- Suppose we have a halting TM ($M$) for a language $L$.
- Generate strings in canonical order on a separate tape, as each is generated, run $M$ on the string and output those that are accepted.

Proof (2)

- Suppose $L$ is generated in canonical order.
- If $L$ is finite, build a DFA for $L$ and run it like a TM (finiteness is undecidable).
- If $L$ is infinite, build a halting TM for $L$ as follows:
  - Given an input $w$, start up the enumerator and compare each output with $w$. If $w$ appears, halt and accept. However, if a string longer than $w$ appears, halt and fail.

Results from Chapter 7
Section 7.3

Computable languages and functions.
Functions

- Turing machines can calculate functions where $f(i_1, i_2, \ldots, i_n)$ is an integer
- The calculation encodes $(i_1, i_2, \ldots, i_n)$ as $w = 0^{i_1}10^{i_2}1\ldots10^{i_n}$
- If $f(i_1, i_2, \ldots, i_n)$ is defined and equal to $m$, then the TM has $w$ on the input tape and stops when $0^m$ is on the tape

Partially recursive functions

- If $f$ is undefined for that input, the TM may run for ever or stop but not in the correct form
- Functions that are computed in this way are called partially recursive functions
- Our Turing Machine simulator includes an example of a machine that computes the product of $m$ and $n$

Church-Turing thesis

- Church’s hypothesis (also called the Church-Turing thesis) is that the partially recursive functions are all the “computable” partial functions over the integers
- The partially recursive functions seem to be computable even though the machine may not halt
- Are all computable functions partially recursive functions? -- Who knows?

Other systems

- Since we do not have a final definition of “computable” we cannot answer the question
- Church’s $\lambda$-calculus, Post systems, general recursive functions (Kleene) all provide a framework for defining computable functions
- and they all define partially recursive functions
Non-deterministic Turing Machines

Simulating a non-deterministic Turing machine using a Deterministic one

Deterministic / non-deterministic Turing machines

- We also want to show that Turing machines differ from PDA’s in that non-determinism does not increase power
- There are non-deterministic Turing machines but they are no more powerful than deterministic ones

non-deterministic TM’s

- We can simulate a non-deterministic machine using a deterministic machine
- First the definition of a non-deterministic TM:

choice of transitions

- In a non-deterministic TM, the transition function is \( \delta : Q \times \Gamma \rightarrow \text{ finite subsets of } Q \times \Gamma \times \{L,R\} \)
  - e.g.
  \[
  \delta(q, X) = \{(p_0, X_0, D_0), (p_1, X_1, D_1), \ldots, (p_{k-1}, X_{k-1}, D_{k-1})\} 
  \]
determining which choice

- One aspect of the deterministic simulation of a non-deterministic machine is to enumerate the possible choices
- Let $r$ be the maximum number of possible transitions for ANY $\delta(q, X)$
- Find $r$ symbols for counting $0,1,2,\ldots,9,A,B,\ldots,K$

key idea

- When the non-deterministic TM $M_{ND}$ runs on an input $w$, it must choose a transition at each step
- Keep a record of the choices: $i_1, i_2, \ldots, i_s$, where each $i$ has a value in 0 through $K$
- If $M_{ND}$ accepts $w$, some sequence of choices will be used

deterministic simulation

- Simulate $M_{ND}$ on input $w$ as follows:
- One tape 1 generate $k = 1, 2, 3, 4, \ldots$
- For each $k$, use tape 2 to generate all sequences of $k$ digits using $0,1,\ldots,K$
- For each sequence $i_1, i_2, \ldots, i_k$ on tape 2, run $M_{ND}$ on $w$, using EXACTLY the choice $i_1$ for step 1, $i_2$ for step 2, etc.

keep generating sequences...

- It can happen that a particular sequence expects transitions that do not actually exist
- In that case, restart $M_{ND}$ over with input $w$, using the next sequence of choices on tape 2
- If all choices are exhausted, increment $k$ on tape 1
...until you accept

- If there is an accepting sequence, it will be generated eventually
- At no time do we try to run \( M_{ND} \) for ever
- The process can halt and fail if we reach a \( k \) for which ALL sequences halt and fail
- The overall process may run for ever without accepting

Computability

Post’s Correspondence Problem
Valid Turing Computations and other results

Instance of PCP

- Take two lists of strings over the same alphabet \( \Sigma \): \( A = w_1, w_2, \ldots, w_k \)
  and \( B = x_1, x_2, \ldots, x_k \) for the same \( k \)
- The two lists \((A, B)\) are called an instance of Post’s Correspondence Problem (PCP)

a solution of an instance of PCP

- The PCP instance has a solution when we can find a SINGLE sequence of indices
  \( i_1, i_2, \ldots, i_m \) \( m \geq 1 \), such that

\[
i_1 w_{i_2} w_{i_3} \ldots w_{i_m} = x_{i_1} x_{i_2} x_{i_3} \ldots x_{i_m}
\]

- Note that the sequence chosen is the same on both sides
Example

- The PCP $A = 1, 10111, 10$ and $B = 111, 10, 0$ has a solution 2,1,1,3:
  
  \[
  10111 \ 1 \ 1 \ 10 = 10 \ 111 \ 111 \ 0
  \]

- The PCP $A = 10, 011, 101$ and $B = 101, 11, 011$ does not have a solution

Proof by contradiction

- If $i_1, i_2, \ldots, i_m, m \geq 1$, is a solution of the PCP: $A = 10, 011, 101$ and $B = 101, 11, 011$
  - $i_1$ must be 1, since 2 and 3 put a 0 opposite a 1, so we have 10 and 101
  - $i_2$ can only be 1 or 3, but 1 is no good since it gives 1010 and 101101, which do not match
  - So $i_2 = 3$ and we have 10101 and 101011, forcing $i_3 = 1$ or 3

Proof continued

- If $i_3$ were 1, then we would have the strings 1010110 and 101011101 and again a 1 clashes with a 0, so we must have $i_3 = 3$ and we have reached 10101101 and 10101101
  - The argument continues with $i_4, i_5, \ldots$ all equal to 3: $10(101)^r$ and $101(011)^r = 10(101)^r11$ -- the two strings will NEVER have the same length
  - There is no solution

PCP is undecidable

- A lengthy proof shows that if PCP were decidable then $L_u$ would be recursive (a special form of PCP is used)
  - PCP can be used to show that it is undecidable whether a CFG is ambiguous: