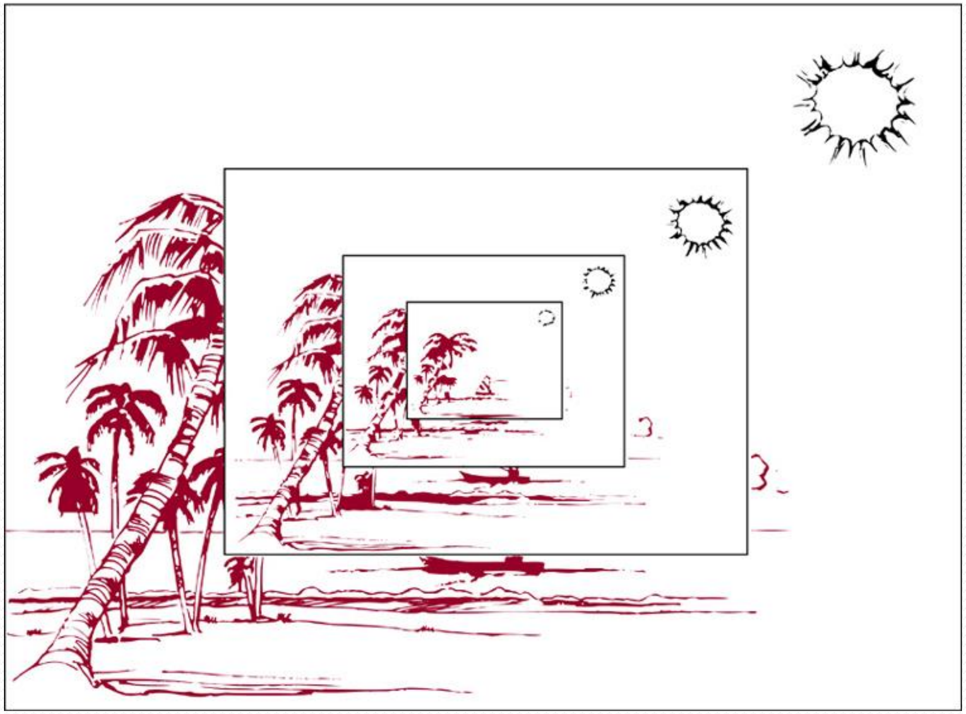
Recursive Definitions and Structural Induction

Section 5.3

Chapter Summary

- Recursively defined Functions.
- Recursively defined sets and Structures
- Structural Induction

5.3 Recursive definitions and structural induction



A recursively defined picture

Recursive definitions

- The sequence of powers of 2 is given by $a_n=2^n$ for $n=0, 1, 2, \ldots$
- Can also be defined by $a_0=1$, and a rule for finding a term of the sequence from the previous one, i.e., $a_{n+1}=2a_n$
- Can use induction to prove results about the sequence

Recursively Defined Functions

- We use two steps to define a function with the set of non-negative integers as its domain:
  - **Basis step**: specify the value for the function at zero
  - **Recursive step**: give a rule for finding its value at an integer from its values at smaller integers
- Such a definition is called a **recursive** or **inductive definition**

Example 1

- Suppose f is defined recursively by
  - $f(0)=3$
  - $f(n+1)=2f(n)+3$

  Find f(1), f(2), f(3), and f(4)
  - $f(1)=2f(0)+3=2*3+3=9$
  - $f(2)=2f(1)+3=2*9+3=21$
  - $f(3)=2f(2)+3=2*21+3=45$
  - $f(4)=2f(3)+3=2*45+3=93$

Example 2

- Give an inductive definition of the factorial function $f(n)=n!$
- Note that $(n+1)!=(n+1)\cdot n!$
- We can define $f(0)=1$ and $f(n+1)=(n+1)f(n)$
- To determine a value, e.g., $f(5)=5!$, we can use the recursive function

$$f(5)=5\cdot f(4)=5\cdot 4\cdot f(3)=5\cdot 4\cdot 3\cdot f(2)=5\cdot 4\cdot 3\cdot 2\cdot f(1)$$
$$=5\cdot 4\cdot 3\cdot 2\cdot 1\cdot f(0)=5\cdot 4\cdot 3\cdot 2\cdot 1\cdot 1=120$$

Recursive functions

- Recursively defined functions are well defined
- For every positive integer, the value of the function is determined in an unambiguous way
- Given any positive integer, we can use the two parts of the definition to find the value of the function at that integer
- We obtain the same value no matter how we apply two parts of the definition

Example 3

- Give a recursive definition of a^n, where a is a non-zero real number and n is a non-negative integer
- The recursive definition contains two parts :
  - First: $a^0=1$
  - Then the rule for finding $a^{n+1}= a\cdot a^n$ for n= 1,2,3,....
  - These two equations uniquely define a^n for all non-negative integer n

Example 4

- Give a recursive definition of $\sum_{k=0}^{n} a_k$
  - The first part of the recursive definition
  $$\sum_{k=0}^{0} a_k = a_0$$
  - The second part is
  $$\sum_{k=0}^{n+1} a_k = (\sum_{k=0}^{n} a_k) + a_{n+1}$$

Example 5 – Fibonacci numbers

- Fibonacci numbers f_0, f_1, f_2, are defined by the equations, $f_0=0$, $f_1=1$, and $f_n=f_{n-1}+f_{n-2}$ for $n=2, 3, 4, \ldots$
- By definition

$f_2=f_1+f_0=1+0=1$

$f_3=f_2+f_1=1+1=2$

$f_4=f_3+f_2=2+1=3$

$f_5=f_4+f_3=3+2=5$

$f_6=f_5+f_4=5+3=8$

Recursively defined sets and structures

- Consider the subset S of the set of integers defined by
  - Basis step: $3 \in S$
  - Recursive step: if $x \in S$ and $y \in S$, then $x+y \in S$
- The new elements formed by this are $3+3=6$, $3+6=9$, $6+6=12$, …
- We will show that S is the set of all positive multiples of 3 (using structural induction)

String

- Definition 1:
- The set $\sum^*$ of strings over the alphabet $\sum$ can be defined recursively by
  - Basis step: $\lambda \in \sum^*$ (where λ is the empty string containing no symbols)
  - Recursive step: if $w \in \sum^*$ and $x \in \sum$ then $wx \in \sum^*$
- The basis step defines that the empty string belongs to string
- The recursive step states new strings are produced by adding a symbol from $\sum$ to the end of stings in $\sum^*$
- At each application of the recursive step, strings containing one additional symbol are generated

Example 6

- If $\sum=\{0, 1\}$, the strings found to be in $\sum^*$, the set of all bit strings, are λ, specified to be in $\sum^*$ in the basis step
- 0 and 1 found in the 1st recursive step
- 00, 01, 10, and 11 are found in the 2nd recursive step, and so on

Concatenation

- Definition 2: Two strings can be combined via the operation of concatenation
- Let $\sum$ be a set of symbols and $\sum^*$ be the set of strings formed from symbols in $\sum$
- We can define the concatenation for two strings by recursive steps
  - Basis step: if $w \in \sum^*$, then $w \cdot \lambda = w$, where λ is the empty string
  - Recursive step: If $w_1 \in \sum^*$, $w_2 \in \sum^*$ and $x \in \sum$, then $w_1 \cdot (w_2 x) = (w_1 \cdot w_2)x$
  - Oftentimes $w_1 \cdot w_2$ is rewritten as $w_1 w_2$
  - e.g., w_1=abra, and w_2=cadabra, $w_1 w_2$=abracadabra

Length of a string

- Give a recursive definition of l(w), the length of a string w
- The length of a string is defined by
  - $l(\lambda)=0$
  - $l(wx)=l(w)+1$ if $w \in \sum^*$ and $x \in \sum$

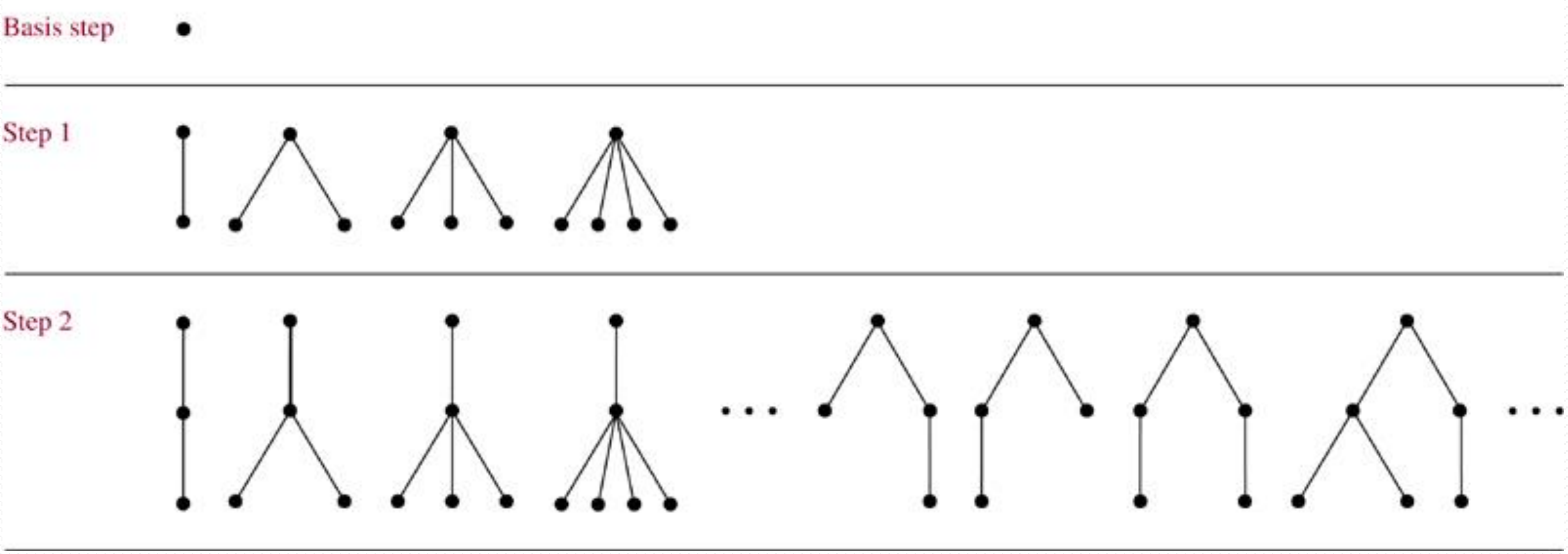
Well-formed formulae

- We can define the set of **well-formed formulae** for compound statement forms involving T, F, proposition variables, and operators from the set {$\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$}
- Basis step: T, F, and s, where s is a propositional variable are well-formed formulae
- Recursive step: If E and F are well-formed formulae, then $\neg E$, $E \wedge F$, $E \vee F$, $E \rightarrow F$, $E \leftrightarrow F$ are well-formed formulae
- From an initial application of the recursive step, we know that $(p \vee q)$, $(p \rightarrow F)$, $(F \rightarrow q)$ and $(q \wedge F)$ are well-formed formulae
- A second application of the recursive step shows that $((p \vee q) \rightarrow (q \wedge F))$, $(q \vee (p \vee q))$, and $((p \rightarrow F) \rightarrow T)$ are well-formed formulae

Rooted trees

- The set of rooted trees, where a rooted tree consists of a set of vertices containing a distinguished vertex called the root, and edges connecting these vertices, can be defined recursively by
  - Basis step: a single vertex r is a rooted tree
  - Recursive step: suppose that T_1, T_2, …, T_n are disjoint rooted trees with roots r_1, r_2, …, r_n, respectively.
  - Then the graph formed by starting with a root r, which is not in any of the rooted trees T_1, T_2, …, T_n, and adding an edge from r to each of the vertices r_1, r_2, …, r_n, is also a rooted tree

Rooted trees



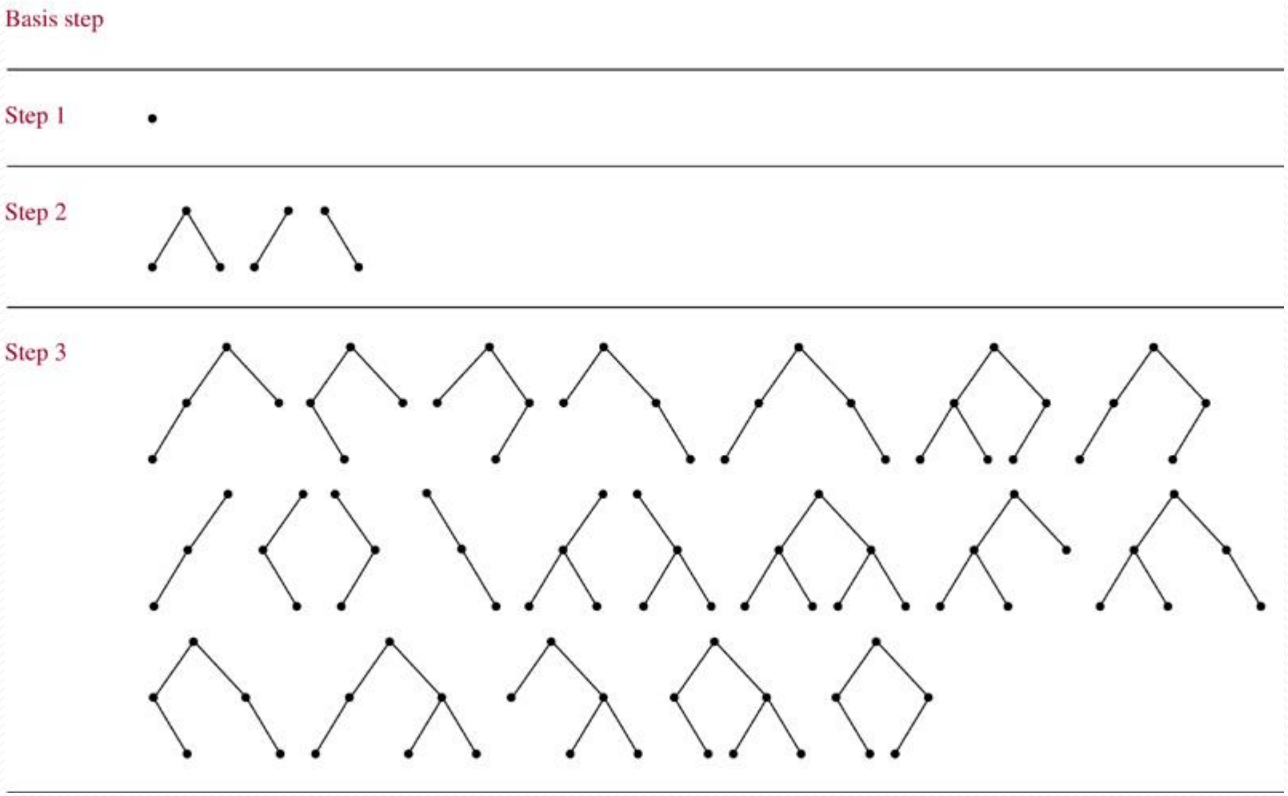
Binary trees

- Binary trees are special type of rooted trees.
- At each vertex, there are at most two branches (one left subtree and one right subtree)
- Extended binary trees: the left subtree or the right subtree can be empty
- Full binary trees: must have left and right subtrees

Extended binary trees

- The set of extended binary trees can be defined by
  - Basis step: the empty set is an extended binary tree
  - Recursive step: If T_1 and T_2 are disjoint extended binary trees, there is an extended binary tree, denoted by $T_1 \cdot T_2$, consisting of a root r together with edges connecting the root to each of the roots of the left subtree T_1 and right subtree T_2, when these trees are non-empty

Extended binary trees



Full binary trees

- The set of full binary trees can be defined recursively
  - Basis step: There is a full binary tree consisting only of a single vertex r
  - Recursive step: If T_1 and T_2 are disjoint full binary trees, there is a full binary tree, denoted by $T_1 \cdot T_2$, consisting of a root r together with edges connecting the root to each of the roots of the left subtree T_1 and right subtree T_2

Full binary tree