Part I: Fundamental Concepts

Marco T. Morazán

Seton Hall University

Outline

Introduction to FSM

- Historically, formal languages and automata theory courses have been theoretical pencil-and-paper courses
- Students design algorithms in theory (i.e., without implementing them)

Introduction to FSM

- Historically, formal languages and automata theory courses have been theoretical pencil-and-paper courses
- Students design algorithms in theory (i.e., without implementing them)
- If there is a bug in the algorithm then there is a bug in the proof of a theorem.

Introduction to FSM

- Historically, formal languages and automata theory courses have been theoretical pencil-and-paper courses
- Students design algorithms in theory (i.e., without implementing them)
- If there is a bug in the algorithm then there is a bug in the proof of a theorem.
- Therefore, we focus on implementation.
- Unit testing and runtime bugs give you immediate feedback on your implementation providing the opportunity to make corrections before submitting work for grading.

Introduction to FSM

- Historically, formal languages and automata theory courses have been theoretical pencil-and-paper courses
- Students design algorithms in theory (i.e., without implementing them)
- If there is a bug in the algorithm then there is a bug in the proof of a theorem.
- Therefore, we focus on implementation.
- Unit testing and runtime bugs give you immediate feedback on your implementation providing the opportunity to make corrections before submitting work for grading.
- We use a domain specific language called `FSM` (**F**unctional **S**tate **M**achines)
- `FSM` provides readers of this textbook with the ability to design, program, test, and debug algorithms before writing theorems or submitting for grading
- `FSM` is embedded in `Racket`

Introduction to FSM

- To write FSM programs the first line in every program must be:

```
#lang fsm
```

- This line informs DrRacket that the programming language used is FSM
- Provides the testing facilities from rackunit

Introduction to FSM

- To write FSM programs the first line in every program must be:

```
#lang fsm
```

- This line informs DrRacket that the programming language used is FSM
- Provides the testing facilities from rackunit
- You may use anything in Racket

Introduction to FSM

- To write FSM programs the first line in every program must be:

```
#lang fsm
```

- This line informs DrRacket that the programming language used is FSM
- Provides the testing facilities from rackunit
- You may use anything in Racket
- A program is never considered complete without unit tests
- For the purposes of this textbook, the required syntax is:

```
(check-equal? <expression> <expression>)
```

Introduction to FSM

- To write FSM programs the first line in every program must be:

```
#lang fsm
```

- This line informs DrRacket that the programming language used is FSM
- Provides the testing facilities from rackunit
- You may use anything in Racket
- A program is never considered complete without unit tests
- For the purposes of this textbook, the required syntax is:

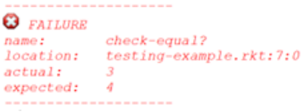
```
(check-equal? <expression> <expression>)
```

- Consider, for example, running the following program:

```
#lang fsm

(check-equal? (= 6 6) #t)
(check-equal? (* (+ 2 3) (/ 20 2)) 50)
(check-equal? (string-length "FSM") 4)
```

The result in the interactions window is:



Introduction to FSM

- The Design Recipe
  1. Outline the representation of values.
  2. Outline the computation.
  3. Write the function's signature and purpose.
  4. Write the function's header.
  5. Write unit tests.
  6. Write the function's body.
  7. Run the tests and, if necessary, redesign.

Introduction to FSM

- The Design Recipe
  1. Outline the representation of values.
  2. Outline the computation.
  3. Write the function's signature and purpose.
  4. Write the function's header.
  5. Write unit tests.
  6. Write the function's body.
  7. Run the tests and, if necessary, redesign.
- Scaling a binary tree of numbers

Introduction to FSM

- The Design Recipe
  1. Outline the representation of values.
  2. Outline the computation.
  3. Write the function's signature and purpose.
  4. Write the function's header.
  5. Write unit tests.
  6. Write the function's body.
  7. Run the tests and, if necessary, redesign.
- Scaling a binary tree of numbers
- ```
  ;; A binary tree of numbers, (btof number), is either:
  ;;  1. '()  2. number  3. (list number (btof number) (btof number))
  ```
  ```

Introduction to FSM

- The Design Recipe
  1. Outline the representation of values.
  2. Outline the computation.
  3. Write the function's signature and purpose.
  4. Write the function's header.
  5. Write unit tests.
  6. Write the function's body.
  7. Run the tests and, if necessary, redesign.
- Scaling a binary tree of numbers
- ```
  ;; A binary tree of numbers, (btof number), is either:
  ;;  1. '()  2. number  3. (list number (btof number) (btof number))
  ```
- ```
  ;;; (btof number) ... → ...
  ;;; Purpose: ...
  ;(define (f-on-bt a-bt ...)
  ;  (cond [(empty? a-bt)  ...]
  ;        [(number? a-bt) ...(f-on-number a-bt)...]
  ;        [else ...(f-on-number (first a-bt))...
  ;              ...(f-on-bt (second a-bt))...
  ;              ...(f-on-bt (third a-bt))...]))
  ;;; Tests
  ;(check-equals? (f-on-bt empty ...) ...)
  ;(check-equals? (f-on-bt number ...) ...)
  ;(check-equals? (f-on-bt (list number ... ...) ...) ...)
  ```

Introduction to FSM

- Reason about each subtype independently

Introduction to FSM

- Reason about each subtype independently
- If the given binary tree is empty then there is nothing to scale and the resulting tree is empty

Introduction to FSM

- Reason about each subtype independently
- If the given binary tree is empty then there is nothing to scale and the resulting tree is empty
- If the given binary tree is a leaf then multiply it by the given scalar

Introduction to FSM

- Reason about each subtype independently
- If the given binary tree is empty then there is nothing to scale and the resulting tree is empty
- If the given binary tree is a leaf then multiply it by the given scalar
- If the given binary tree is an interior then make a list with the result of calling * with the root value and the given scalar, making a recursive call with the left subtree and the scalar, and making a recursive call with the right subtree and the scalar

Introduction to FSM

- ```
  #lang fsm

  ;; (btof number) number → (btof number)
  ;; Purpose: Scale the given (btof number) by the
  ;;          given scalar
  (define (scale-bt a-bt k)
  ```
  ```

Introduction to FSM

- ```
  #lang fsm

  ;; (btof number) number → (btof number)
  ;; Purpose: Scale the given (btof number) by the
  ;;          given scalar
  (define (scale-bt a-bt k)
  ```

- ```
  ;; Tests
  ;; empty bt tests
  (check-equal? (scale-bt '() 10) '())
  ;; leaf bt tests
  (check-equal? (scale-bt -50 2) -100)
  (check-equal? (scale-bt 40  8) 320)
  ;; interior node bt tests
  (check-equal? (scale-bt (list 10 '() (list -8 -4 '())) -2)
                (list -20 '() (list 16 8 '())))
  (check-equal? (scale-bt (list 0
                                (list 1 2 3)
                                (list 4
                                      (list 5 '() '())
                                      (list 6 7 8)))
  ```

Introduction to FSM

- ```
  #lang fsm

  ;; (btof number) number → (btof number)
  ;; Purpose: Scale the given (btof number) by the
  ;;          given scalar
  (define (scale-bt a-bt k)
  ```
- ```
  (cond [(empty? a-bt)  a-bt]
        [(number? a-bt) (* k a-bt)]
        [else (list (* k (first a-bt))
                    (scale-bt (second a-bt) k)
                    (scale-bt (third a-bt) k))]))
  ```
- ```
  ;; Tests
  ;; empty bt tests
  (check-equal? (scale-bt '() 10) '())
  ;; leaf bt tests
  (check-equal? (scale-bt -50 2) -100)
  (check-equal? (scale-bt 40  8) 320)
  ;; interior node bt tests
  (check-equal? (scale-bt (list 10 '() (list -8 -4 '())) -2)
                (list -20 '() (list 16 8 '())))
  (check-equal? (scale-bt (list 0
                                (list 1 2 3)
                                (list 4
                                      (list 5 '() '())
                                      (list 6 7 8)))
  ```
  ```

Introduction to FSM

HOMEWORK

- Problems: 1–10

Introduction to FSM

- Two constants that are useful to know before writing FSM programs:

BLANK	Denotes a blank space in an input tape.
EMP	Denotes the empty word (i.e., a word of length 0)

Introduction to FSM

- Two constants that are useful to know before writing FSM programs:

`BLANK`	Denotes a blank space in an input tape.
`EMP`	Denotes the empty word (i.e., a word of length 0)

- The following are some important FSM data definitions:

`alphabet` A list of lowercase symbols of length 1 not including `EMP`.

`word` A nonempty `(listof symbol)` from an alphabet.

`nts` A set of nonterminal symbols. Each nonterminal symbol is denoted by an uppercase English letter: `[A..Z]`.

`state machine` A state machine is either:

- Deterministic Finite Automaton (`dfa`)
- Nondeterministic Finite Automaton (`ndfa`)
- Pushdown Automaton (`pda`)
- Turing Machine (`tm`)
- Turing Machine Language Recognizer (`tm-language-recognizer`)
- Multitape Turing Machine (`mttm`)
- `mttm` Language Recognizer

`grammar` A grammar is either:

- A Regular Grammar (`rg`)
- A Context-Free Grammar (`cfg`)
- A Context-Sensitive Grammar (`csg`)

Essential Background

- You are embarking on a journey to explore some of the most fundamental questions in Computer Science and, perhaps surprisingly to you, in nature

Essential Background

- You are embarking on a journey to explore some of the most fundamental questions in Computer Science and, perhaps surprisingly to you, in nature
- The most obvious one is: what is an algorithm?
- Go ahead, tell me!

Essential Background

- You are embarking on a journey to explore some of the most fundamental questions in Computer Science and, perhaps surprisingly to you, in nature
- The most obvious one is: what is an algorithm?
- Go ahead, tell me!
- We shall explore models of computation that allow us to formally define what "algorithm" means

Essential Background

- You are embarking on a journey to explore some of the most fundamental questions in Computer Science and, perhaps surprisingly to you, in nature
- The most obvious one is: what is an algorithm?
- Go ahead, tell me!
- We shall explore models of computation that allow us to formally define what "algorithm" means
- What can be computed?
- Is there an algorithm to solve every problem that can be posed?

Essential Background

- You are embarking on a journey to explore some of the most fundamental questions in Computer Science and, perhaps surprisingly to you, in nature
- The most obvious one is: what is an algorithm?
- Go ahead, tell me!
- We shall explore models of computation that allow us to formally define what "algorithm" means
- What can be computed?
- Is there an algorithm to solve every problem that can be posed?
- When is an algorithm practical?

Essential Background

- Automata theory is concerned with the mathematical properties of computation models
- It helps us understand what can and cannot be computed with a given model
- You may think of this as programming using an `API`
- Given an `API` there are problems that may be solved with it and there are problems that cannot be solved wit it.

Essential Background

- For instance, consider the following mathematical functions:

```
g(x) = g(x+1)

f(x, y) = 42
```

- The value of `f(0, 50)` is clearly 42
- Can you implement these functions as a program?

Essential Background

- For instance, consider the following mathematical functions:

```
g(x) = g(x+1)

f(x, y) = 42
```

- The value of `f(0, 50)` is clearly 42
- Can you implement these functions as a program?
- In Java, you may attempt to write methods that looks like this:

```
int g(int x)
{ return(g(x++)); }

int f(int x, int y)
{ return(42); }

void main(String[] args)
{
  System.out.println(f(10, 15));
  System.out.println(f(10, g(8)));
}
```

- What happens when you run the program?

Essential Background

- For instance, consider the following mathematical functions:

```
g(x) = g(x+1)

f(x, y) = 42
```

- The value of `f(0, 50)` is clearly 42
- Can you implement these functions as a program?
- In Java, you may attempt to write methods that looks like this:

```
int g(int x)
{ return(g(x++)); }

int f(int x, int y)
{ return(42); }

void main(String[] args)
{
  System.out.println(f(10, 15));
  System.out.println(f(10, g(8)));
}
```

- What happens when you run the program?
- 42 is printed for the first call to `f`
- The second call to `f` goes into an infinite recursion

Essential Background

- In Haskell, the functions may be implemented as follows:

```
g:: Int -> Int
g x = g x+1

f:: Int -> Int -> Int
f x y = 42

main :: IO ()
main =  do
print(f 10 15)
print(f 10 (g 8))
```

- What happens when you run the program?

Essential Background

- In Haskell, the functions may be implemented as follows:

```
g:: Int -> Int
g x = g x+1

f:: Int -> Int -> Int
f x y = 42

main :: IO ()
main =  do
print(f 10 15)
print(f 10 (g 8))
```

- What happens when you run the program?
- Both calls to f print 42

Essential Background

- In Haskell, the functions may be implemented as follows:

```
g:: Int -> Int
g x = g x+1

f:: Int -> Int -> Int
f x y = 42

main :: IO ()
main =  do
print(f 10 15)
print(f 10 (g 8))
```

- What happens when you run the program?
- Both calls to f print 42
- The difference is the model of computation Java and Haskell use

Essential Background

- Our study of computation models shall focus on the recognition and the generation of language elements
- A *language* is a set of *words* over a given alphabet
- A word is an ordered collection of alphabet elements, read left to right, that may contain repetitions

Essential Background

- Our study of computation models shall focus on the recognition and the generation of language elements
- A *language* is a set of *words* over a given alphabet
- A word is an ordered collection of alphabet elements, read left to right, that may contain repetitions
- The number of words in a language may be finite or infinite
- For example, the following defines the language containing all words of length less than or equal to 2 over the alphabet (a b):

  LT4 = $\{\epsilon$ a b aa ab ba bb$\}$

Essential Background

- Our study of computation models shall focus on the recognition and the generation of language elements
- A *language* is a set of *words* over a given alphabet
- A word is an ordered collection of alphabet elements, read left to right, that may contain repetitions
- The number of words in a language may be finite or infinite
- For example, the following defines the language containing all words of length less than or equal to 2 over the alphabet `(a b)`:

  `LT4 =` $\{\epsilon$ `a b aa ab ba bb`$\}$
- Given that the language is finite it suffices to list its elements
- We denote the empty word (the word with zero alphabet elements) as ϵ
- This word has length zero and `bb` has length 2.

Essential Background

- The following defines the infinite language for all strings that end with an a over `(a b)`:

  `ENDA = {w | w ends with an a}`

- In this definition a `set former` is used
- This is necessary because it is impossible to list all the elements of the language

Essential Background

- The following defines the infinite language for all strings that end with an a over (a b):

  `ENDA = {w | w ends with an a}`

- In this definition a `set former` is used
- This is necessary because it is impossible to list all the elements of the language
- The computation needed to determine if a word, `w`, is a member of a language `L` is performed by a *finite-state machine* (aka a finite-state automaton)
- These theoretical machines have lead to efficient algorithms to determine if a pattern is found in a strand of DNA or in a block of text

Essential Background

- The following defines the infinite language for all strings that end with an a over (a b):

  `ENDA = {w | w ends with an a}`

- In this definition a `set former` is used
- This is necessary because it is impossible to list all the elements of the language
- The computation needed to determine if a word, `w`, is a member of a language `L` is performed by a *finite-state machine* (aka a finite-state automaton)
- These theoretical machines have lead to efficient algorithms to determine if a pattern is found in a strand of DNA or in a block of text
- The computations needed to generate a word `w` that is a member of a language `L` are done using a grammar
- The study of grammars have led to the development of parsers, interpreters, and compilers for programming languages and to natural language processing
- You should dispel any misconception that the study of automata theory and formal languages has no practical applications relevant to the life of a problem solver and programmer.

Essential Mathematical Background

Sets

- A set is a collection of items without repetitions
- Used to represent alphabets and languages

Essential Mathematical Background

Sets

- A set is a collection of items without repetitions
- Used to represent alphabets and languages
- An alphabet, Σ, may be defined to be a set of alphabetic characters and a language may be defined as a set of words formed by the characters in Σ:

$$\Sigma = \{\texttt{a b c}\}$$

$$\texttt{L} = \{\texttt{w} \mid \texttt{w} \in \Sigma^* \wedge \texttt{w has an even length}\}$$

- The above definitions state that Σ is the alphabet containing `a`, `b`, and `c`
- `L` is the set of words formed using zero or more elements of Σ that have an even length

Essential Mathematical Background

Sets

- A set is a collection of items without repetitions
- Used to represent alphabets and languages
- An alphabet, Σ, may be defined to be a set of alphabetic characters and a language may be defined as a set of words formed by the characters in Σ:

  Σ = $\{$a b c$\}$

  L = $\{$w | w $\in$ Σ^* $\wedge$ w has an even length$\}$
- The above definitions state that Σ is the alphabet containing a, b, and c
- L is the set of words formed using zero or more elements of Σ that have an even length
- The *, called Kleene star, stands for zero or more elements of the set
- A $^+$ stands for one or more elements of the set

Essential Mathematical Background

Sets

- Finite sets are specified by listing their elements inside curly braces
- Infinite sets are represented using a set former specifying the conditions that must hold:

  {w | P(w)}

Essential Mathematical Background

Sets

- HOMEWORK: 1–3

Essential Mathematical Background

Sets

- If every element of A is a member of B then A is a *subset* of B. We denote this as:

$$\mathtt{A} \subseteq \mathtt{B}$$

Essential Mathematical Background

Sets

- If every element of `A` is a member of `B` then `A` is a *subset* of `B`. We denote this as:

  $A \subseteq B$

- If every element of `A` is a member of `B` and not all elements of `B` are in `A` then `A` is a `proper subset` of `B`. We denote this as:

  $A \subset B$

Essential Mathematical Background

Sets

- If every element of `A` is a member of `B` then `A` is a *subset* of `B`. We denote this as:

  $$\mathtt{A} \subseteq \mathtt{B}$$

- If every element of `A` is a member of `B` and not all elements of `B` are in `A` then `A` is a `proper subset` of `B`. We denote this as:

  $$\mathtt{A} \subset \mathtt{B}$$

- We say that two sets are equal, `A = B`, if and only if $\mathtt{A} \subseteq \mathtt{B}$ and $\mathtt{B} \subseteq \mathtt{A}$

Essential Mathematical Background

Sets

- Sets may be combined to create new sets using `set operations`
- Six basic set operations: union, intersection, difference, complement, cross product, and power set
- Assume that Σ `= (a b)`.

Essential Mathematical Background

Sets

- Sets may be combined to create new sets using `set operations`
- Six basic set operations: union, intersection, difference, complement, cross product, and power set
- Assume that Σ = `(a b)`.
- The union of two sets:

$$\mathtt{A} \cup \mathtt{B} = \{\mathtt{x} \mid \mathtt{x} \in \mathtt{A} \vee \mathtt{x} \in \mathtt{B}\}$$

Essential Mathematical Background

Sets

- Sets may be combined to create new sets using `set operations`
- Six basic set operations: union, intersection, difference, complement, cross product, and power set
- Assume that Σ = `(a b)`.
- The union of two sets:

$$A \cup B = \{x \mid x \in A \vee x \in B\}$$

- The intersection of two sets:

$$A \cap B = \{x \mid x \in A \wedge x \in B\}$$

- We say that two sets whose intersection is empty are *mutually exclusive*

Essential Mathematical Background

Sets

- Sets may be combined to create new sets using `set operations`
- Six basic set operations: union, intersection, difference, complement, cross product, and power set
- Assume that Σ = `(a b)`.
- The union of two sets:

  $$A \cup B = \{x \mid x \in A \vee x \in B\}$$

- The intersection of two sets:

  $$A \cap B = \{x \mid x \in A \wedge x \in B\}$$

- We say that two sets whose intersection is empty are *mutually exclusive*
- The difference of two sets:

  $$A - B = \{w \mid w \in A \wedge w \notin B\}$$

Essential Mathematical Background

Sets

- Sets may be combined to create new sets using `set operations`
- Six basic set operations: union, intersection, difference, complement, cross product, and power set
- Assume that Σ = `(a b)`.
- The union of two sets:

  $$\texttt{A} \cup \texttt{B} = \{\texttt{x} \mid \texttt{x} \in \texttt{A} \vee \texttt{x} \in \texttt{B}\}$$

- The intersection of two sets:

  $$\texttt{A} \cap \texttt{B} = \{\texttt{x} \mid \texttt{x} \in \texttt{A} \wedge \texttt{x} \in \texttt{B}\}$$

- We say that two sets whose intersection is empty are *mutually exclusive*
- The difference of two sets:

  $$\texttt{A} - \texttt{B} = \{\texttt{w} \mid \texttt{w} \in \texttt{A} \wedge \texttt{w} \notin \texttt{B}\}$$

- The complement of a set:

  $$\bar{\texttt{A}} = \Sigma^* - \texttt{A}$$

Essential Mathematical Background

Sets

- Sets may be combined to create new sets using `set operations`
- Six basic set operations: union, intersection, difference, complement, cross product, and power set
- Assume that Σ = `(a b)`.
- The union of two sets:

$$A \cup B = \{x \mid x \in A \vee x \in B\}$$

- The intersection of two sets:

$$A \cap B = \{x \mid x \in A \wedge x \in B\}$$

- We say that two sets whose intersection is empty are *mutually exclusive*
- The difference of two sets:

$$A - B = \{w \mid w \in A \wedge w \notin B\}$$

- The complement of a set:

$$\bar{A} = \Sigma^* - A$$

- The cross product of two languages:

$$A \times B = \{(a\ b) \mid a \in a \wedge b \in B\}$$

Essential Mathematical Background

Sets

- The power set of a set $\mathtt{A}$, $2^{\mathtt{A}}$, is the complete set of subsets of $\mathtt{A}$

Essential Mathematical Background

Sets

- The power set of a set A, $2^{\mathtt{A}}$, is the complete set of subsets of A
- ```
  EMPTY = {}

  SET1 = {r e a}
  ```
  We can observe that the power set of each is:
  $2^{\mathtt{EMPTY}}$ = {{}}

  $2^{\mathtt{SET1}}$ = {{r e a} {r e} {r a} {r} {e a} {e} {a} {}}
  ```

Essential Mathematical Background

Sets

- The power set of a set A, 2^{A}, is the complete set of subsets of A
-
  ```
  EMPTY = {}

  SET1 = {r e a}
  ```
  We can observe that the power set of each is:

  $2^{\mathtt{EMPTY}}$ = {{}}

  $2^{\mathtt{SET1}}$ = {{r e a} {r e} {r a} {r} {e a} {e} {a} {}}
- An arbitrary element of 2^{A} either contains or does not contain a
- Suggests an algorithm to compute 2^{A}

Essential Mathematical Background

Sets

- ```
  #lang fsm
  #| Data Definitions
      A list of X, lox, is either:
        1. '()
        2. (cons X lox)
      A set of X, setx, is a (listof X)      |#
  ;; Sample setx
  (define EMPTY-SET '())
  (define SET1 '(r e a))
  ```
  ```

Essential Mathematical Background

Sets

- ```
  #lang fsm
  #| Data Definitions
      A list of X, lox, is either:
        1. '()
        2. (cons X lox)
      A set of X, setx, is a (listof X)     |#
  ;; Sample setx
  (define EMPTY-SET '())
  (define SET1 '(r e a))
  ```
- ```
  ;; setx → (listof setx)
  ;; Purpose: Return the power set of the given set
  (define (powerSet A)
  ```

Essential Mathematical Background

Sets

- ```
  #lang fsm
  #| Data Definitions
      A list of X, lox, is either:
        1. '()
        2. (cons X lox)
      A set of X, setx, is a (listof X)      |#
  ;; Sample setx
  (define EMPTY-SET '())
  (define SET1 '(r e a))
  ```
- ```
  ;; setx → (listof setx)
  ;; Purpose: Return the power set of the given set
  (define (powerSet A)
  ```
- ```
  (check-equal? (powerSet EMPTY-SET) '(()))
  (check-equal? (powerSet SET1)
                '((r e a) (r e) (r a) (r) (e a) (e) (a) ()))
  ```
  ```

Essential Mathematical Background

Sets

- ```
  #lang fsm
  #| Data Definitions
       A list of X, lox, is either:
         1. '()
         2. (cons X lox)
       A set of X, setx, is a (listof X)      |#
  ;; Sample setx
  (define EMPTY-SET '())
  (define SET1 '(r e a))
  ```
- ```
  ;; setx → (listof setx)
  ;; Purpose: Return the power set of the given set
  (define (powerSet A)
  ```
- ```
    (cond [(null? A) (list '())]
  ```
- ```
  (check-equal? (powerSet EMPTY-SET) '(()))
  (check-equal? (powerSet SET1)
                '((r e a) (r e) (r a) (r) (e a) (e) (a) ()))
  ```

Essential Mathematical Background

Sets

- ```
  #lang fsm
  #| Data Definitions
      A list of X, lox, is either:
        1. '()
        2. (cons X lox)
      A set of X, setx, is a (listof X)      |#
  ;; Sample setx
  (define EMPTY-SET '())
  (define SET1 '(r e a))
  ```
- ```
  ;; setx → (listof setx)
  ;; Purpose: Return the power set of the given set
  (define (powerSet A)
  ```
- ```
    (cond [(null? A) (list '())]
  ```
- ```
          [else
           (let ((rest (powerSet (cdr A))))
             (append
              (map (lambda (x) (cons (car A) x)) rest)
              rest))]))
  ```
- ```
  (check-equal? (powerSet EMPTY-SET) '(()))
  (check-equal? (powerSet SET1)
                '((r e a) (r e) (r a) (r) (e a) (e) (a) ()))
  ```
  ```

Essential Mathematical Background

Set Laws

- $A \cup A = A$

  $A \cap A = A$

Essential Mathematical Background

Set Laws

- $$\texttt{A} \cup \texttt{A} = \texttt{A}$$

  $$\texttt{A} \cap \texttt{A} = \texttt{A}$$

- Let us prove the first.

Theorem

$A \cup A = A$

Essential Mathematical Background

Set Laws

- ```
  A ∪ A = A

  A ∩ A = A
  ```
- Let us prove the first.

Theorem

$A \cup A = A$

Proof.

We shall prove that:

(a) A $\cup$ A $\subseteq$ A
(b) A $\subseteq$ A $\cup$ A.

(a) Assume x is an arbitrary element in A $\cup$ A. This means that x $\in$ A or x $\in$ A. Hence, x $\in$ A. Therefore, we may conclude that A $\cup$ A $\subseteq$ A.
(b) Assume that x is an arbitrary element in A. This means that x $\in$ A $\cup$ A. Therefore, we may conclude that A $\subseteq$ A $\cup$ A.
Therefore, the following implication holds:
A $\cup$ A $\subseteq$ A $\wedge$ A $\subseteq$ A $\cup$ A $\Rightarrow$ A = A $\cup$ A. $\square$

-
  ```

Essential Mathematical Background

Set Laws

- $$A \cup B = B \cup A$$

  $$A \cap B = B \cap A$$

Essential Mathematical Background

Set Laws

- $$A \cup B = B \cup A$$

  $$A \cap B = B \cap A$$

- Let us prove the second.

Theorem

$A \cap B = B \cap A$

Essential Mathematical Background

Set Laws

- ```
  A ∪ B = B ∪ A

  A ∩ B = B ∩ A
  ```
- Let us prove the second.

Theorem

$A \cap B = B \cap A$

Proof.

We shall prove that:

(a) A $\cap$ B $\subseteq$ B $\cap$ A

(b) B $\cap$ A $\subseteq$ A $\cap$ B.

(a) Assume x is an arbitrary element in A $\cap$ B. This means that x $\in$ A and x $\in$ B. Hence, x $\in$ B $\cap$ A. Thus, we may conclude that A $\cap$ B $\subseteq$ B $\cap$ A.

(b) Assume that x is an arbitrary element in B $\cap$ A. This means that x $\in$ B and x $\in$ A. This means that x $\in$ B and x $\in$ A. Hence, x $\in$ A $\cap$ B. Thus, we may conclude that B $\cap$ A $\subseteq$ A $\cap$ B.

Therefore, the following implication holds:

$A \cap B \subseteq B \cap A \wedge B \cap A \subseteq A \cap B \Rightarrow A \cap B = B \cap A.$ $\square$

-
  ```

Essential Mathematical Background

Set Laws

- $(A \cup B) \cup C = A \cup (B \cup C)$

  $(A \cap B) \cap C = A \cap (B \cap C)$

Essential Mathematical Background

Set Laws

- $$(A \cup B) \cup C = A \cup (B \cup C)$$
  $$(A \cap B) \cap C = A \cap (B \cap C)$$
- Let us prove the first.

Theorem

$(A \cup B) \cup C = A \cup (B \cup C)$

Essential Mathematical Background

Set Laws

- $$(A \cup B) \cup C = A \cup (B \cup C)$$

  $$(A \cap B) \cap C = A \cap (B \cap C)$$
- Let us prove the first.

Theorem

$(A \cup B) \cup C = A \cup (B \cup C)$

Proof.

We shall prove that:

(a) $(A \cup B) \cup C \subseteq A \cup (B \cup C)$
(b) $A \cup (B \cup C) \subseteq (A \cup B) \cup C$.

(a) Assume x is an arbitrary element of $(A \cup B) \cup C$. This means that:

- $$\begin{aligned} & x \in (A \cup B) \vee x \in C \\ \Rightarrow\ & x \in A \vee x \in B \vee x \in C \\ \Rightarrow\ & x \in A \cup (B \cup C) \end{aligned}$$

Thus, we may conclude that $(A \cup B) \cup C \subseteq A \cup (B \cup C)$.

(b) Assume x is an arbitrary element of $A \cup (B \cup C)$. This means that:

$$\begin{aligned} & x \in A \vee x \in (B \cup C) \\ \Rightarrow\ & x \in A \vee x \in B \vee x \in C \\ \Rightarrow\ & x \in (A \cup B) \cup C \end{aligned}$$

Thus, we may conclude that $A \cup (B \cup C) \subseteq (A \cup B) \cup C$.

Essential Mathematical Background

Set Laws

- The absorption laws are:

$$(A \cup B) \cap A = A$$

$$(A \cap B) \cup A = A$$

Essential Mathematical Background

Set Laws

- The absorption laws are:

  ```
  (A ∪ B) ∩ A = A

  (A ∩ B) ∪ A = A
  ```

- Let us prove the first.

Theorem

$(A \cup B) \cap A = A$

Essential Mathematical Background

Set Laws

- The absorption laws are:

$$(A \cup B) \cap A = A$$

$$(A \cap B) \cup A = A$$

- Let us prove the first.

Theorem

$(A \cup B) \cap A = A$

Proof.

We shall prove that:
(a) $(A \cup B) \cap A \subseteq A$
(b) $A \subseteq (A \cup B) \cap A$.

(a) Assume x is an arbitrary element of $(A \cup B) \cap A$. This means that:

$$x \in (A \cup B) \wedge x \in A$$

Therefore, we may conclude that $(A \cup B) \cap A \subseteq A$.

(b) Assume x is an arbitrary element of A. This means that:

$$x \in (A \cup B) \Rightarrow x \in (A \cup B) \cap A$$

Therefore, we may conclude that $A \subseteq (A \cup B) \cap A$.

Thus, the following implication holds:

$$(A \cup B) \cap A \subseteq A \wedge A \subseteq (A \cup B) \cap A$$

Essential Mathematical Background

Set Laws

- DeMorgan's laws are:
  `A - (B ∪ C) = (A - B) ∩ (A - C)`   `A - (B ∩ C) = (A - B) ∪ (A - C)`

Essential Mathematical Background

Set Laws

- DeMorgan's laws are:
  $A - (B \cup C) = (A - B) \cap (A - C) \quad A - (B \cap C) = (A - B) \cup (A - C)$

Theorem

$A - (B \cup C) = (A - B) \cap (A - C)$

Proof.

(a) $A - (B \cup C) \subseteq (A - B) \cap (A - C)$
(b) $(A - B) \cap (A - C) \subseteq A - (B \cup C)$

(a) Assume x is an arbitrary element of $A - (B \cup C)$. This means that:

$$x \in A \wedge x \notin B \wedge x \notin C \Rightarrow x \in (A - B) \wedge x \in (A - C)$$

Therefore, we may conclude that $A - (B \cup C) \subseteq (A - B) \cap (A - C)$.

(b) Assume x is an arbitrary element of $(A - B) \cap (A - C)$. This means that:

$$\begin{aligned} x \in (A - B) \wedge x \in (A - C) &\Rightarrow x \in A \wedge x \notin B \wedge x \notin C \\ &\Rightarrow x \notin (B \cup C) \\ &\Rightarrow x \in A - (B \cup C) \end{aligned}$$

Therefore, we may conclude that $(A - B) \cap (A - C) \subseteq A - (B \cup C)$.
Thus, the following implication holds:

$$A - (B \cup C) \subseteq (A - B) \cap (A - C) \wedge (A - B) \cap (A - C) \subseteq A - (B \cup C)$$
$$\Rightarrow$$
$$A - (B \cup C) = (A - B) \cap (A - C)$$

Essential Mathematical Background

Sets

- HOMEWORK: 4–9

Essential Mathematical Background

Relations and Functions

- Relations associate an element of the domain (the input set) with an element of the range (the output set)

Essential Mathematical Background

Relations and Functions

- Relations associate an element of the domain (the input set) with an element of the range (the output set)
- $\geq$ has as its domain pairs (or tuples) of real numbers and has as its range the Booleans (i.e., true and false)

Essential Mathematical Background

Relations and Functions

- Relations associate an element of the domain (the input set) with an element of the range (the output set)
- $\geq$ has as its domain pairs (or tuples) of real numbers and has as its range the Booleans (i.e., true and false)
- Relations define a language (or if you like a set)
- The members of such a language are the elements that are related. The language defined by $\geq$ may be specified as follows:

  $$\texttt{GEQ} = \{(\texttt{x y}) \mid \texttt{x}, \texttt{y} \in \mathbb{R} \land \texttt{x} \geq \texttt{y}\}$$

Relations and Functions

Sets

- A special type of relation is called a `function`. A function is a binary relation that associates a member from its domain with a unique member from its range

Relations and Functions

Sets

- A special type of relation is called a `function`. A function is a binary relation that associates a member from its domain with a unique member from its range
- For instance, consider the following sets:

$$\mathtt{NAMES} = \{\mathtt{w} \mid \mathtt{w\ is\ a\ first\ name}\}$$

$$\mathtt{PASSP} = \{\mathtt{p} \mid \mathtt{p\ is\ a\ person\ with\ a\ single\ passport}\}$$

- The relation that maps a person with a single passport to a first name is specified as follows:

$$\mathtt{R}_1 = \{(\mathtt{p}\ \mathtt{n}) \mid \mathtt{p} \in \mathtt{PASSP} \land \mathtt{n} \in \mathtt{NAMES}\}$$

- $\mathtt{R}_1$ is a function because every single passport holder has a first name

Relations and Functions

Sets

- A special type of relation is called a `function`. A function is a binary relation that associates a member from its domain with a unique member from its range
- For instance, consider the following sets:

$$\texttt{NAMES} = \{\texttt{w} \mid \texttt{w is a first name}\}$$

$$\texttt{PASSP} = \{\texttt{p} \mid \texttt{p is a person with a single passport}\}$$

- The relation that maps a person with a single passport to a first name is specified as follows:

$$\texttt{R}_1 = \{(\texttt{p n}) \mid \texttt{p} \in \texttt{PASSP} \wedge \texttt{n} \in \texttt{NAMES}\}$$

- $\texttt{R}_1$ is a function because every single passport holder has a first name
- On the other hand, consider this relation:

$$\texttt{R}_2 = \{(\texttt{n p}) \mid \texttt{p} \in \texttt{PASSP} \wedge \texttt{n} \in \texttt{NAMES}\}$$

  $\texttt{R}_2$ is not a function because a more than one person with a single passport may have the same first name

Relations and Functions

Sets

- We say that a function, `f`, is *one-to-one* if for any distinct `a` and `b`, `f(a)` $\neq$ `f(b)`

Relations and Functions

Sets

- We say that a function, `f`, is *one-to-one* if for any distinct `a` and `b`, `f(a)` $\neq$ `f(b)`
- For instance, consider the following sets:

  STATES = {s | s is a state in the USA}

  STCAPS = {c | c is a state capital in the USA}
- The following is a one-to-one function:

  $\texttt{R}_3$ = {(c s) | c $\in$ STCAPS $\wedge$ s $\in$ STATES}

  This is a one-to-one functions because every state capital is the capital of a different state

Relations and Functions

Sets

- We say that a function, f, is *one-to-one* if for any distinct a and b, f(a) $\neq$ f(b)
- For instance, consider the following sets:

  STATES = {s | s is a state in the USA}

  STCAPS = {c | c is a state capital in the USA}
- The following is a one-to-one function:

  R_3 = {(c s) | c $\in$ STCAPS $\wedge$ s $\in$ STATES}

  This is a one-to-one functions because every state capital is the capital of a different state
- We say that a function, f, is *onto* if each element in the range is mapped to by at least one element in the domain
- R_3 is onto because every state has a state capital

Relations and Functions

Sets

- We say that a function, f, is *one-to-one* if for any distinct a and b, $\texttt{f(a)} \neq \texttt{f(b)}$
- For instance, consider the following sets:

$$\texttt{STATES = } \{\texttt{s | s is a state in the USA}\}$$

$$\texttt{STCAPS = } \{\texttt{c | c is a state capital in the USA}\}$$

- The following is a one-to-one function:

$$\texttt{R}_3 \texttt{ = } \{\texttt{(c s) | c} \in \texttt{STCAPS} \wedge \texttt{s} \in \texttt{STATES}\}$$

  This is a one-to-one functions because every state capital is the capital of a different state
- We say that a function, f, is *onto* if each element in the range is mapped to by at least one element in the domain
- $\texttt{R}_3$ is onto because every state has a state capital
- A function is a bijection if it is both one-to-one and onto.
- $\texttt{R}_3$ is a bijection because it is both one-to-one and onto.

Countable and Uncountable

Sets

- The study of formal languages involves infinite sets
- We must be careful about how we reason about infinite sets

Countable and Uncountable

Sets

- The study of formal languages involves infinite sets
- We must be careful about how we reason about infinite sets
- At an intuitive level an infinite set is different from a finite set because all the members of the infinite set cannot be listed in a finite amount of time
- If you understand this observation then it is easy to see that the size or *cardinality* of a finite set is less than the cardinality of an infinite set

Countable and Uncountable

Sets

- The study of formal languages involves infinite sets
- We must be careful about how we reason about infinite sets
- At an intuitive level an infinite set is different from a finite set because all the members of the infinite set cannot be listed in a finite amount of time
- If you understand this observation then it is easy to see that the size or *cardinality* of a finite set is less than the cardinality of an infinite set
- Two sets, A and B, have the same cardinality, |A| and |B|, if there is a bijection between A and B
- The existence of such a bijection informs us that the sets are *equinumerous*

Countable and Uncountable Sets

- The study of formal languages involves infinite sets
- We must be careful about how we reason about infinite sets
- At an intuitive level an infinite set is different from a finite set because all the members of the infinite set cannot be listed in a finite amount of time
- If you understand this observation then it is easy to see that the size or *cardinality* of a finite set is less than the cardinality of an infinite set
- Two sets, A and B, have the same cardinality, |A| and |B|, if there is a bijection between A and B
- The existence of such a bijection informs us that the sets are *equinumerous*
- It is easy to believe that the cardinality of two infinite sets is the same
- All infinite sets are not equinumerous
- There exists infinite sets, |C| and |D|, such that a bijection between them does not exist

Countable and Uncountable Sets

- The study of formal languages involves infinite sets
- We must be careful about how we reason about infinite sets
- At an intuitive level an infinite set is different from a finite set because all the members of the infinite set cannot be listed in a finite amount of time
- If you understand this observation then it is easy to see that the size or *cardinality* of a finite set is less than the cardinality of an infinite set
- Two sets, `A` and `B`, have the same cardinality, `|A|` and `|B|`, if there is a bijection between `A` and `B`
- The existence of such a bijection informs us that the sets are *equinumerous*
- It is easy to believe that the cardinality of two infinite sets is the same
- All infinite sets are not equinumerous
- There exists infinite sets, `|C|` and `|D|`, such that a bijection between them does not exist
- Consider the set of real numbers, $\mathbb{R}$, and, $\mathbb{Z}$, the set of integers
- Both sets are infinite but they are not equinumerous
- How can we prove this?

Countable and Uncountable Sets

- The study of formal languages involves infinite sets
- We must be careful about how we reason about infinite sets
- At an intuitive level an infinite set is different from a finite set because all the members of the infinite set cannot be listed in a finite amount of time
- If you understand this observation then it is easy to see that the size or *cardinality* of a finite set is less than the cardinality of an infinite set
- Two sets, `A` and `B`, have the same cardinality, `|A|` and `|B|`, if there is a bijection between `A` and `B`
- The existence of such a bijection informs us that the sets are *equinumerous*
- It is easy to believe that the cardinality of two infinite sets is the same
- All infinite sets are not equinumerous
- There exists infinite sets, `|C|` and `|D|`, such that a bijection between them does not exist
- Consider the set of real numbers, $\mathbb{R}$, and, $\mathbb{Z}$, the set of integers
- Both sets are infinite but they are not equinumerous
- How can we prove this?
- It is tempting to say that $\mathbb{Z} \subset \mathbb{R}$ and, therefore, there are more real numbers than integers
- Thus, $\mathbb{Z}$ and $\mathbb{R}$ do not have the same cardinality and are not equinumerous

Countable and Uncountable Sets

- The study of formal languages involves infinite sets
- We must be careful about how we reason about infinite sets
- At an intuitive level an infinite set is different from a finite set because all the members of the infinite set cannot be listed in a finite amount of time
- If you understand this observation then it is easy to see that the size or *cardinality* of a finite set is less than the cardinality of an infinite set
- Two sets, `A` and `B`, have the same cardinality, `|A|` and `|B|`, if there is a bijection between `A` and `B`
- The existence of such a bijection informs us that the sets are *equinumerous*
- It is easy to believe that the cardinality of two infinite sets is the same
- All infinite sets are not equinumerous
- There exists infinite sets, `|C|` and `|D|`, such that a bijection between them does not exist
- Consider the set of real numbers, $\mathbb{R}$, and, $\mathbb{Z}$, the set of integers
- Both sets are infinite but they are not equinumerous
- How can we prove this?
- It is tempting to say that $\mathbb{Z} \subset \mathbb{R}$ and, therefore, there are more real numbers than integers
- Thus, $\mathbb{Z}$ and $\mathbb{R}$ do not have the same cardinality and are not equinumerous
- This is fallacious reasoning
- It says nothing about whether or not there exist a bijection between $\mathbb{Z}$ and $\mathbb{R}$
- We shall hold off on this argument until we learn about diagonalization proofs in the next chapter

Countable and Uncountable

Sets

- A set, A, is finite if it is equinumerous with the first n elements of $\mathbb{N}$
- We say that n is the cardinality of A

Countable and Uncountable

Sets

- A set, `A`, is finite if it is equinumerous with the first `n` elements of $\mathbb{N}$
- We say that `n` is the cardinality of `A`
- For example, consider:

  `A = {f o r {m a l}}`

- There is a bijection, `g`, between `A` and `{0 1 2 3}`:

  ```
  g(0) = f      g(1) = o
  g(2) = r      g(3) = {m a l}
  ```

  The `|A|` is 4 and `A` is, therefore, finite.

Countable and Uncountable

Sets

- A set, `A`, is finite if it is equinumerous with the first `n` elements of $\mathbb{N}$
- We say that `n` is the cardinality of `A`
- For example, consider:

```
A = {f o r {m a l}}
```

- There is a bijection, g, between `A` and `{0 1 2 3}`:

```
g(0) = f      g(1) = o
g(2) = r      g(3) = {m a l}
```

  The `|A|` is 4 and `A` is, therefore, finite.
- A set is infinite if it is not finite
- The sets $\mathbb{R}$, $\mathbb{N}$, and $\mathbb{Z}$ are infinite
- There is no bijection between the elements of these sets and the first `n` elements of $\mathbb{N}$

Countable and Uncountable

Sets

- A set, A, is finite if it is equinumerous with the first n elements of $\mathbb{N}$
- We say that n is the cardinality of A
- For example, consider:

  `A = {f o r {m a l}}`
- There is a bijection, g, between A and {0 1 2 3}:

  ```
  g(0) = f      g(1) = o
  g(2) = r      g(3) = {m a l}
  ```

  The |A| is 4 and A is, therefore, finite.
- A set is infinite if it is not finite
- The sets $\mathbb{R}$, $\mathbb{N}$, and $\mathbb{Z}$ are infinite
- There is no bijection between the elements of these sets and the first n elements of $\mathbb{N}$
- Are any of these sets equinumerous?

Countable and Uncountable Sets

- A set, `A`, is finite if it is equinumerous with the first `n` elements of $\mathbb{N}$
- We say that `n` is the cardinality of `A`
- For example, consider:

  `A = {f o r {m a l}}`

- There is a bijection, g, between `A` and `{0 1 2 3}`:

  ```
  g(0) = f      g(1) = o
  g(2) = r      g(3) = {m a l}
  ```

  The `|A|` is 4 and `A` is, therefore, finite.

- A set is infinite if it is not finite
- The sets $\mathbb{R}$, $\mathbb{N}$, and $\mathbb{Z}$ are infinite
- There is no bijection between the elements of these sets and the first `n` elements of $\mathbb{N}$
- Are any of these sets equinumerous?
- $\mathbb{N}$ and $\mathbb{Z}$ are equinumerous. The following is a bijection between these two sets:

$$f(n) = \begin{cases} \frac{n}{2} & \texttt{if n is even} \\ -\frac{n+1}{2} & \texttt{if n is odd} \end{cases}$$

Countable and Uncountable Sets

- HOMEWORK: 10–11

Countable and Uncountable

Sets

- A set is *countably infinite* if it is equinumerous with $\mathbb{N}$
- Intuitively, this means that a program may be written to print out the members of the set
- The program, of course, will run for ever but if you wait long enough any arbitrary element of the set will eventually be printed.

Countable and Uncountable

Sets

- A set is *countably infinite* if it is equinumerous with $\mathbb{N}$
- Intuitively, this means that a program may be written to print out the members of the set
- The program, of course, will run for ever but if you wait long enough any arbitrary element of the set will eventually be printed.
- $\mathbb{N}$ is countably infinite

```
#lang fsm

;;  → (void)
;; Purpose: Print the natural numbers
(define (print-natnums)
  ;; natnum → (void)
  ;; Purpose: Print the natural numbers starting with
  ;;          the given natural number
  (define (printer n)
    (if (= n +inf.0)
        (void)
        (begin
          (displayln n)
          (printer (add1 n)))))
  (printer 0))
```

Countable and Uncountable

Sets

- A set is *countable* if it is finite or it is countably infinite.
- A set is *uncountable* if it is not countable
- Intuitively, a set is uncountable if a program that eventually prints any arbitrary element cannot be written

Countable and Uncountable

Sets

- A set is *countable* if it is finite or it is countably infinite.
- A set is *uncountable* if it is not countable
- Intuitively, a set is uncountable if a program that eventually prints any arbitrary element cannot be written
- To demonstrate that a set is countable it suffices to write a program to print its elements guaranteeing that eventually any arbitrary element is eventually printed
- Sometimes this requires careful design and creativity:

$$\begin{aligned}\texttt{EVENNATS} &= \{2\texttt{n} \mid \texttt{n} \in \mathbb{N}\}\\ \texttt{MULTSOF3} &= \{3\texttt{n} \mid \texttt{n} \in \mathbb{N}\}\\ \texttt{A}^* &= \{\texttt{w} \mid \texttt{w} \in \texttt{a}^*\}\\ \texttt{BIGSET} &= \texttt{EVENNATS} \cup \texttt{MULTSOF3} \cup \texttt{A}^*\end{aligned}$$

- Is BIGSET countable?

Countable and Uncountable

Sets

- A set is *countable* if it is finite or it is countably infinite.
- A set is *uncountable* if it is not countable
- Intuitively, a set is uncountable if a program that eventually prints any arbitrary element cannot be written
- To demonstrate that a set is countable it suffices to write a program to print its elements guaranteeing that eventually any arbitrary element is eventually printed
- Sometimes this requires careful design and creativity:

$$\begin{aligned} \texttt{EVENNATS} &= \{2\texttt{n} \mid \texttt{n} \in \mathbb{N}\} \\ \texttt{MULTSOF3} &= \{3\texttt{n} \mid \texttt{n} \in \mathbb{N}\} \\ \texttt{A}^* &= \{\texttt{w} \mid \texttt{w} \in \texttt{a}^*\} \\ \texttt{BIGSET} &= \texttt{EVENNATS} \cup \texttt{MULTSOF3} \cup \texttt{A}^* \end{aligned}$$

- Is BIGSET countable?
- We can observe that EVENNATS, MULTSOF3, and $\texttt{A}^*$ are countable (you can write the programs!)

Countable and Uncountable

Sets

- A set is *countable* if it is finite or it is countably infinite.
- A set is *uncountable* if it is not countable
- Intuitively, a set is uncountable if a program that eventually prints any arbitrary element cannot be written
- To demonstrate that a set is countable it suffices to write a program to print its elements guaranteeing that eventually any arbitrary element is eventually printed
- Sometimes this requires careful design and creativity:

$$\begin{aligned}\texttt{EVENNATS} &= \{2\texttt{n} \mid \texttt{n} \in \mathbb{N}\}\\ \texttt{MULTSOF3} &= \{3\texttt{n} \mid \texttt{n} \in \mathbb{N}\}\\ \texttt{A}^* &= \{\texttt{w} \mid \texttt{w} \in \texttt{a}^*\}\\ \texttt{BIGSET} &= \texttt{EVENNATS} \cup \texttt{MULTSOF3} \cup \texttt{A}^*\end{aligned}$$

- Is BIGSET countable?
- We can observe that EVENNATS, MULTSOF3, and $\texttt{A}^*$ are countable (you can write the programs!)
- We need a program to prints its elements:

```
;;  → (void)   Purpose: Print the elements of BIGSET
(define (print-bigset)
  (begin
    (print-even-natnums)
    (print-mults3)
    (print-a*)))
```

- Unfortunately, this design does not work

Countable and Uncountable Sets

- A dovetailing strategy yields a different design
- Smoothly fit together printing members of all three sets

Countable and Uncountable Sets

- A dovetailing strategy yields a different design
- Smoothly fit together printing members of all three sets
- Recall that all three sets are countable: each set is equinumerous with $\mathbb{N}$
- Suggests the program can print an element of each set as it traverses the natural numbers:

```
;;  → (void)
;; Purpose: Print the elements of BIGSET
(define (print-bigset)
  ;; natnum → (void)
  ;; Purpose: Print the elements of EVENNATS, MULTS3, and
  ;;          A* starting with the elements indexed by
  ;;          the given natural number
  (define (printer n)
    (if (= n +inf.0)
        (void)
        (begin
          (displayln (* 2 n))
          (displayln (* 3 n))
          (displayln
            (list->string (build-list n (λ (i) #\a))))
          (printer (add1 n)))))
  (printer 0))
```

Countable and Uncountable

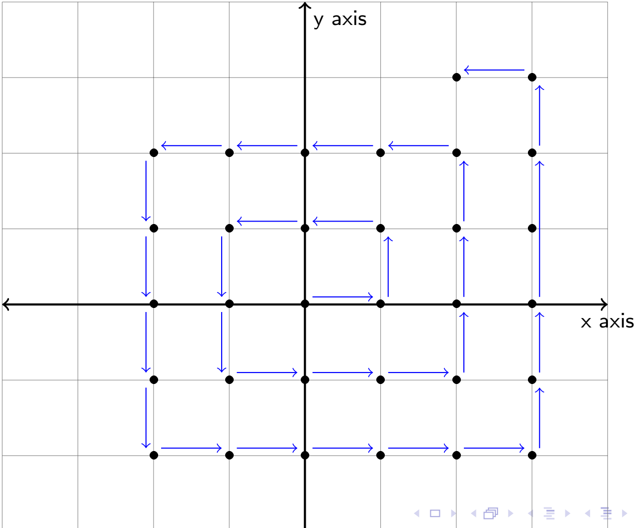
Sets

- Dovetailing is a powerful design technique
- Consider the integer points, `(x, y)`, in the Cartesian plane
- Is this set countable?

Countable and Uncountable

Sets

- Dovetailing is a powerful design technique
- Consider the integer points, `(x, y)`, in the Cartesian plane
- Is this set countable?



Countable and Uncountable

- HOMEWORK: 12, 14
- QUIZ: 17 (due in a week)

Types of Proofs

- A proof is an argument that establishes that a hypothesis is true
- It uses assumptions to reach a conclusion
- Developing a proof is challenging!

Types of Proofs

- A proof is an argument that establishes that a hypothesis is true
- It uses assumptions to reach a conclusion
- Developing a proof is challenging!
- Some fundamental proof techniques that can help guide the process: formal logic proofs, mathematical induction proofs, pigeonhole principle proofs, proofs by contradiction, and diagonalization proofs

Types of Proofs

- A proof is an argument that establishes that a hypothesis is true
- It uses assumptions to reach a conclusion
- Developing a proof is challenging!
- Some fundamental proof techniques that can help guide the process: formal logic proofs, mathematical induction proofs, pigeonhole principle proofs, proofs by contradiction, and diagonalization proofs
- Precisely state the assumptions and state the conclusion (the statement you want to prove)
- Failing to do so is tantamount to writing a function without knowing its inputs and its purpose

Types of Proofs

Formal Logic Proofs

- We are mostly interested in proving conjunctions, disjunctions, implications and equivalences

Types of Proofs

Formal Logic Proofs

- We are mostly interested in proving conjunctions, disjunctions, implications and equivalences
- A conjunction that is true means that all input statements are true
    -

A	B	A $\wedge$ B
False	False	False
False	True	False
True	False	False
True	True	True

- This means that if we wish to prove a conjunction true we must demonstrate that the input statements are true

Types of Proofs

Formal Logic Proofs

- We are mostly interested in proving conjunctions, disjunctions, implications and equivalences
- A conjunction that is true means that all input statements are true
  -

A	B	A ∧ B
False	False	False
False	True	False
True	False	False
True	True	True

- This means that if we wish to prove a conjunction true we must demonstrate that the input statements are true
- Prove:

$$28 + 2 + 1 = 31 \wedge 2 * 3 + 4 = 10$$

Types of Proofs
Formal Logic Proofs

- We are mostly interested in proving conjunctions, disjunctions, implications and equivalences
- A conjunction that is true means that all input statements are true
A	B	A $\wedge$ B
False	False	False
False	True	False
True	False	False
True	True	True
- This means that if we wish to prove a conjunction true we must demonstrate that the input statements are true
- Prove:

$$28 + 2 + 1 = 31 \wedge 2 * 3 + 4 = 10$$

- $$\begin{aligned} 30 + 1 = 31 &\wedge 6 + 4 = 10 \\ 31 = 31 &\wedge 10 = 10 \end{aligned}$$

Types of Proofs

Formal Logic Proofs

- A disjunction that is true means that any of its input statements are true

Types of Proofs

Formal Logic Proofs

- A disjunction that is true means that any of its input statements are true
- The following is the truth table for $\vee$ with two input statements:

A	B	A $\vee$ B
False	False	False
False	True	True
True	False	True
True	True	True

- If we wish to prove a disjunction true we must demonstrate that at least one of its input statements is true

Types of Proofs
Formal Logic Proofs

- A disjunction that is true means that any of its input statements are true
- The following is the truth table for $\vee$ with two input statements:

A	B	A $\vee$ B
False	False	False
False	True	True
True	False	True
True	True	True

- If we wish to prove a disjunction true we must demonstrate that at least one of its input statements is true
- $7 + 1 + 3 > 20 \quad \vee \quad -2 * 2 + 4 \geq 0 \quad \vee \quad x \geq x + x + x$

Types of Proofs

Formal Logic Proofs

- A disjunction that is true means that any of its input statements are true
- The following is the truth table for $\vee$ with two input statements:

A	B	A $\vee$ B
False	False	False
False	True	True
True	False	True
True	True	True

- If we wish to prove a disjunction true we must demonstrate that at least one of its input statements is true
- $7 + 1 + 3 > 20 \quad \vee \quad -2 * 2 + 4 \geq 0 \quad \vee \quad x \geq x + x + x$
- $$\begin{array}{rcl} 7 + 1 + 3 > 20 & \vee & -2 * 2 + 4 \geq 0 \quad \vee \quad x \geq x + x + x \\ 8 + 3 > 20 & \vee & -4 + 4 \geq 0 \quad \vee \quad x \geq x + 2x \\ 11 > 20 & \vee & 0 \geq 0 \quad \vee \quad x \geq 3x \end{array}$$

  Clearly, $0 \geq 0$. Therefore, we may conclude that the disjunction holds.

Types of Proofs

Formal Logic Proofs

- We must know how to prove an implication: `A` $\Rightarrow$ `B`
- `A` is the antecedent and `B` is the consequent
- It states that if `A` is true then `B` is also true:

A	B	A $\Rightarrow$ B
False	False	True
False	True	True
True	False	False
True	True	True

Types of Proofs

Formal Logic Proofs

- We must know how to prove an implication: `A` $\Rightarrow$ `B`
- `A` is the antecedent and `B` is the consequent
- It states that if `A` is true then `B` is also true:

A	B	A $\Rightarrow$ B
False	False	True
False	True	True
True	False	False
True	True	True

- When the antecedent is true then the consequent is also true.
- Assume the antecedent is true and demonstrate that the consequent is true

Types of Proofs

Formal Logic Proofs

- We must know how to prove an implication: `A` $\Rightarrow$ `B`
- `A` is the antecedent and `B` is the consequent
- It states that if `A` is true then `B` is also true:

A	B	A $\Rightarrow$ B
False	False	True
False	True	True
True	False	False
True	True	True

- When the antecedent is true then the consequent is also true.
- Assume the antecedent is true and demonstrate that the consequent is true
- $3x - 9 \geq 0 \Rightarrow x \geq 3$

Types of Proofs

Formal Logic Proofs

- We must know how to prove an implication: $A \Rightarrow B$
- A is the antecedent and B is the consequent
- It states that if A is true then B is also true:

A	B	A ⇒ B
False	False	True
False	True	True
True	False	False
True	True	True

- When the antecedent is true then the consequent is also true.
- Assume the antecedent is true and demonstrate that the consequent is true
- $3x - 9 \geq 0 \Rightarrow x \geq 3$
- Assume that $3x - 9 \geq 0$ is true
- We must show that $x \geq 3$ is true

Types of Proofs

Formal Logic Proofs

- We must know how to prove an implication: $A \Rightarrow B$
- A is the antecedent and B is the consequent
- It states that if A is true then B is also true:

A	B	A ⇒ B
False	False	True
False	True	True
True	False	False
True	True	True

- When the antecedent is true then the consequent is also true.
- Assume the antecedent is true and demonstrate that the consequent is true
- $3x - 9 \geq 0 \Rightarrow x \geq 3$
- Assume that $3x - 9 \geq 0$ is true
- We must show that $x \geq 3$ is true
- $$\begin{aligned} 3x - 9 \geq 0 &\Rightarrow x \geq 3 \\ 3x \geq 9 &\Rightarrow x \geq 3 \\ x \geq 3 &\Rightarrow x \geq 3 \end{aligned}$$

Types of Proofs

Formal Logic Proofs

- Logical equivalence: $\mathtt{A} \Leftrightarrow \mathtt{B}$:

  $\mathtt{A} \Rightarrow \mathtt{B} \wedge \mathtt{B} \Rightarrow \mathtt{A}$

Types of Proofs

Formal Logic Proofs

- Logical equivalence: $\texttt{A} \Leftrightarrow \texttt{B}$:

  $\texttt{A} \Rightarrow \texttt{B} \wedge \texttt{B} \Rightarrow \texttt{A}$

A	B	A ⇔ B
False	False	True
False	True	False
True	False	False
True	True	True

- Holds when both input statements have the same value
- Proving logical equivalence requires proving two implications

Types of Proofs

Formal Logic Proofs

- Logical equivalence: $A \Leftrightarrow B$:
  $A \Rightarrow B \wedge B \Rightarrow A$
-

A	B	A ⇔ B
False	False	True
False	True	False
True	False	False
True	True	True

- Holds when both input statements have the same value
- Proving logical equivalence requires proving two implications
- x is a square $\Leftrightarrow \sqrt{x} \in \mathbb{N}$

Types of Proofs

Formal Logic Proofs

- Logical equivalence: $\texttt{A} \Leftrightarrow \texttt{B}$:
  $\texttt{A} \Rightarrow \texttt{B} \wedge \texttt{B} \Rightarrow \texttt{A}$
-

A	B	A ⇔ B
False	False	True
False	True	False
True	False	False
True	True	True

- Holds when both input statements have the same value
- Proving logical equivalence requires proving two implications
- $\texttt{x is a square} \Leftrightarrow \sqrt{x} \in \mathbb{N}$
- We must prove that:
  1. $\texttt{x is a square} \Rightarrow \sqrt{x} \in \mathbb{N}$
  2. $\sqrt{x} \in \mathbb{N} \Rightarrow \texttt{x is a square}$

Types of Proofs

Formal Logic Proofs

- Logical equivalence: $\texttt{A} \Leftrightarrow \texttt{B}$:

  $\texttt{A} \Rightarrow \texttt{B} \wedge \texttt{B} \Rightarrow \texttt{A}$

-

A	B	A ⇔ B
False	False	True
False	True	False
True	False	False
True	True	True

- Holds when both input statements have the same value
- Proving logical equivalence requires proving two implications
- $\texttt{x is a square} \Leftrightarrow \sqrt{x} \in \mathbb{N}$
- We must prove that:
  1. $\texttt{x is a square} \Rightarrow \sqrt{x} \in \mathbb{N}$
  2. $\sqrt{x} \in \mathbb{N} \Rightarrow \texttt{x is a square}$
- Assume x is a square:

$$\begin{aligned}\texttt{x is a square} &\Rightarrow \texttt{x = k * k, k} \in \mathbb{N}\\ &\Rightarrow \sqrt{x} = \texttt{k}\\ &\Rightarrow \sqrt{x} \in \mathbb{N}\end{aligned}$$

Types of Proofs

Formal Logic Proofs

- Logical equivalence: $\texttt{A} \Leftrightarrow \texttt{B}$:

  $\texttt{A} \Rightarrow \texttt{B} \wedge \texttt{B} \Rightarrow \texttt{A}$

  -

A	B	A ⇔ B
False	False	True
False	True	False
True	False	False
True	True	True

- Holds when both input statements have the same value
- Proving logical equivalence requires proving two implications
- $\texttt{x is a square} \Leftrightarrow \sqrt{x} \in \mathbb{N}$
- We must prove that:
  1. $\texttt{x is a square} \Rightarrow \sqrt{x} \in \mathbb{N}$
  2. $\sqrt{x} \in \mathbb{N} \Rightarrow \texttt{x is a square}$
- Assume x is a square:

$$\begin{aligned} \texttt{x is a square} &\Rightarrow \texttt{x = k * k, k} \in \mathbb{N} \\ &\Rightarrow \sqrt{x} = \texttt{k} \\ &\Rightarrow \sqrt{x} \in \mathbb{N} \end{aligned}$$

- Assume $\sqrt{x} \in \mathbb{N}$:

$$\begin{aligned} \sqrt{x} \in \mathbb{N} &\Rightarrow \texttt{x = k * k} \\ &\Rightarrow \texttt{x = k}^2 \end{aligned}$$

Types of Proofs

Formal Logic Proofs

- Universal quantification states that for all members of a set some predicate holds

Types of Proofs

Formal Logic Proofs

- Universal quantification states that for all members of a set some predicate holds
- $\texttt{M4 = \{n | n} \in \mathbb{N} \wedge \texttt{n is a multiple of 4\}}$

Types of Proofs

Formal Logic Proofs

- Universal quantification states that for all members of a set some predicate holds
- $\texttt{M4 = } \{\texttt{n | n} \in \mathbb{N} \wedge \texttt{n is a multiple of 4}\}$
- We can make statements about all the members of M4:

  $\forall\ \texttt{n} \in \texttt{M4 n = 2k, where k} \in \mathbb{N}$

Types of Proofs

Formal Logic Proofs

- Universal quantification states that for all members of a set some predicate holds
- $\texttt{M4 = } \{\texttt{n | n} \in \mathbb{N} \wedge \texttt{n is a multiple of 4}\}$
- We can make statements about all the members of M4:

  $\forall\ \texttt{n} \in \texttt{M4 n = 2k, where k} \in \mathbb{N}$
- To prove a statement using universal quantification we must argue that the statement is true for an arbitrary member of the set:

Types of Proofs

Formal Logic Proofs

- Universal quantification states that for all members of a set some predicate holds
- $\texttt{M4 = \{n | n} \in \mathbb{N} \wedge \texttt{n is a multiple of 4\}}$
- We can make statements about all the members of M4:

  $\forall$ n $\in$ M4 n = 2k, where k $\in \mathbb{N}$
- To prove a statement using universal quantification we must argue that the statement is true for an arbitrary member of the set:
- Let x be an arbitrary member of M4.

  x $\in$ M4 $\Rightarrow$ x = 4h

  $\Rightarrow$ x = 2*2*h

  $\Rightarrow$ x = 2k, where k = 2h $\square$

Types of Proofs

Formal Logic Proofs

- Existential quantification states that there exists a member of the set for which a predicate holds

Types of Proofs

Formal Logic Proofs

- Existential quantification states that there exists a member of the set for which a predicate holds
- ∃ `n` ∈ `M4 n is a multiple of 5`

Types of Proofs

Formal Logic Proofs

- Existential quantification states that there exists a member of the set for which a predicate holds
- $\exists$ `n` $\in$ `M4 n is a multiple of 5`
- To prove a statement using existential quantification we must demonstrate that a specific member of the set satisfies the predicate

Types of Proofs

Formal Logic Proofs

- Existential quantification states that there exists a member of the set for which a predicate holds
- $\exists$ n $\in$ M4 n is a multiple of 5
- To prove a statement using existential quantification we must demonstrate that a specific member of the set satisfies the predicate
- Let x = 20.

  x = 20 $\Rightarrow$ x = 4 * 5
  $\Rightarrow$ x $\in$ M4 $\wedge$ x is a multiple of 5 $\square$

Types of Proofs

Formal Logic Proofs

- HOMEWORK: 1–10

Types of Proofs

Mathematical Induction

- We define the set of natural numbers, $\mathbb{N}$, as follows:

```
A natural number is either:
  1. 0
  2. n + 1, where n ∈ ℕ
```

Types of Proofs

Mathematical Induction

- We define the set of natural numbers, $\mathbb{N}$, as follows:

  ```
  A natural number is either:
    1. 0
    2. n + 1, where n ∈ ℕ
  ```

- We say that 0 is a base instance of $\mathbb{N}$: a value known to be in the set
- A set may have more than one base value

Types of Proofs

Mathematical Induction

- We define the set of natural numbers, $\mathbb{N}$, as follows:

```
A natural number is either:
  1. 0
  2. n + 1, where n ∈ ℕ
```

- We say that 0 is a base instance of $\mathbb{N}$: a value known to be in the set
- A set may have more than one base value
- Values that are not base values (let us call them inductive values) must be constructed from other elements in the set

Types of Proofs

Mathematical Induction

- We define the set of natural numbers, $\mathbb{N}$, as follows:

```
A natural number is either:
  1. 0
  2. n + 1, where n ∈ ℕ
```

- We say that 0 is a base instance of $\mathbb{N}$: a value known to be in the set
- A set may have more than one base value
- Values that are not base values (let us call them inductive values) must be constructed from other elements in the set
- 5 is a natural number:

$$\begin{aligned} 0 \in \mathbb{N} &\Rightarrow 1 \in \mathbb{N} \\ &\Rightarrow 2 \in \mathbb{N} \\ &\Rightarrow 3 \in \mathbb{N} \\ &\Rightarrow 4 \in \mathbb{N} \\ &\Rightarrow 5 \in \mathbb{N} \end{aligned}$$

Types of Proofs

Mathematical Induction

- What is required to prove that a predicate holds $\forall\ \mathbb{N}$?

Types of Proofs

Mathematical Induction

- What is required to prove that a predicate holds $\forall\ \mathbb{N}$?
- The definition suggests that such a proof must have two parts
- The first part establishes that the predicate holds for the base value (i.e., 0)

Types of Proofs

Mathematical Induction

- What is required to prove that a predicate holds $\forall\ \mathbb{N}$?
- The definition suggests that such a proof must have two parts
- The first part establishes that the predicate holds for the base value (i.e., 0)
- Second part establishes the predicate holds for the inductive instances

Types of Proofs

Mathematical Induction

- What is required to prove that a predicate holds $\forall\ \mathbb{N}$?
- The definition suggests that such a proof must have two parts
- The first part establishes that the predicate holds for the base value (i.e., 0)
- Second part establishes the predicate holds for the inductive instances
- The second part, however, requires special care:

```
Prove P(1) holds
Prove P(2) holds
Prove P(3) holds
         ⋮
```

Types of Proofs

Mathematical Induction

- What is required to prove that a predicate holds $\forall\ \mathbb{N}$?
- The definition suggests that such a proof must have two parts
- The first part establishes that the predicate holds for the base value (i.e., 0)
- Second part establishes the predicate holds for the inductive instances
- The second part, however, requires special care:

```
Prove P(1) holds
Prove P(2) holds
Prove P(3) holds
      ⋮
```

- We must rely on the implication suggested by the data definition:

$$\mathtt{n} \in \mathbb{N} \Rightarrow \mathtt{n + 1} \in \mathbb{N}$$

Types of Proofs

Mathematical Induction

- What is required to prove that a predicate holds $\forall\ \mathbb{N}$?
- The definition suggests that such a proof must have two parts
- The first part establishes that the predicate holds for the base value (i.e., 0)
- Second part establishes the predicate holds for the inductive instances
- The second part, however, requires special care:

```
Prove P(1) holds
Prove P(2) holds
Prove P(3) holds
        ⋮
```

- We must rely on the implication suggested by the data definition:

  $\mathtt{n} \in \mathbb{N} \Rightarrow \mathtt{n + 1} \in \mathbb{N}$

- Suggests that we must show: `P(n) holds` $\Rightarrow$ `P(n+1) holds`

Types of Proofs

Mathematical Induction

- What is required to prove that a predicate holds $\forall \mathbb{N}$?
- The definition suggests that such a proof must have two parts
- The first part establishes that the predicate holds for the base value (i.e., 0)
- Second part establishes the predicate holds for the inductive instances
- The second part, however, requires special care:

```
Prove P(1) holds
Prove P(2) holds
Prove P(3) holds
        ⋮
```

- We must rely on the implication suggested by the data definition:

  $\mathtt{n} \in \mathbb{N} \Rightarrow \mathtt{n + 1} \in \mathbb{N}$

- Suggests that we must show: `P(n) holds` $\Rightarrow$ `P(n+1) holds`
- The good news is that we know how to prove an implication
- `P(n)` is called the inductive hypothesis.

Types of Proofs

Mathematical Induction

- What is required to prove that a predicate holds $\forall \mathbb{N}$?
- The definition suggests that such a proof must have two parts
- The first part establishes that the predicate holds for the base value (i.e., 0)
- Second part establishes the predicate holds for the inductive instances
- The second part, however, requires special care:

```
Prove P(1) holds
Prove P(2) holds
Prove P(3) holds
         ⋮
```

- We must rely on the implication suggested by the data definition:

  $\mathtt{n} \in \mathbb{N} \Rightarrow \mathtt{n + 1} \in \mathbb{N}$

- Suggests that we must show: `P(n) holds` $\Rightarrow$ `P(n+1) holds`
- The good news is that we know how to prove an implication
- `P(n)` is called the inductive hypothesis.
- This is the principal of *mathematical induction*:
  1. Prove the base case
  2. The inductive step
     1. State, `P(k)`, the inductive hypothesis
     2. State, `P(k+1)`, what must be proven
     3. Assume `P(k)` is true and prove `P(k+1)`
- The inductive hypothesis is valid for all values in `[0..k]`

Types of Proofs

Mathematical Induction

- ```
  #lang fsm

  ;; natnum → natnum
  ;; Purpose: Compute the square of the given natnum
  (define (square n)
    (if (= n 0)
        0
        (+ (sub1 (* 2 n)) (square (sub1 n)))))

  ;; Tests
  (check-equal? (square 0)   0)
  (check-equal? (square 5)   25)
  (check-equal? (square 100) 10000)
  ```
  ```

Types of Proofs

Mathematical Induction

- ```
  #lang fsm

  ;; natnum → natnum
  ;; Purpose: Compute the square of the given natnum
  (define (square n)
    (if (= n 0)
        0
        (+ (sub1 (* 2 n)) (square (sub1 n)))))

  ;; Tests
  (check-equal? (square 0)   0)
  (check-equal? (square 5)   25)
  (check-equal? (square 100) 10000)
  ```
- Can you prove that the function is correct?
  ```

Types of Proofs

Mathematical Induction

- ```
  ;; natnum → natnum
  ;; Purpose: Compute the square of the given natnum
  (define (square n)
    (if (= n 0)
        0
        (+ (sub1 (* 2 n)) (square (sub1 n)))))
  ```
  ```

Types of Proofs

Mathematical Induction

-
```
;; natnum → natnum
;; Purpose: Compute the square of the given natnum
(define (square n)
  (if (= n 0)
      0
      (+ (sub1 (* 2 n)) (square (sub1 n)))))
```

Theorem

(square n) returns n^2

Proof.

Base Case: $n = 0$

If $n = 0$ then `(square n)` $=$ `(square 0)` returns $0 = 0^2 = n^2$

Inductive Step:

Assume: `(square k)` returns k^2, for $n = k \geq 0$

Show that: `(square (add1 k))` returns `(add1 k)`^{2}

$k \geq 0 \Rightarrow$ `(add1 k)` > 0

$\Rightarrow$ `(square k+1)` returns `(+ (sub1 (* 2 (add1 k))) (square k))`

$\Rightarrow$ `(square k+1)` returns `(+ (sub1 (+ (* 2 k) 2)) k`^{2}`)`

$\Rightarrow$ `(square k+1)` returns `(+ (+ (* 2 k) 1)) k`^{2}`)`

$\Rightarrow$ `(square k+1)` returns `(add1 k)`^{2} $\square$ $\square$

-

Types of Proofs

Mathematical Induction

- HOMEWORK: 11–13

Types of Proofs

Pigeonhole Principle

- Imagine that in a colony of pigeons you have 50 pigeons and 45 pigeonholes
- It is impossible to place each pigeon in its own pigeonhole

Types of Proofs

Pigeonhole Principle

- Imagine that in a colony of pigeons you have 50 pigeons and 45 pigeonholes
- It is impossible to place each pigeon in its own pigeonhole
- A one-to-one function from the set of pigeons to the set of pigeonholes does not exists
- This observation leads to *the pigeonhole principle*.

Types of Proofs

Pigeonhole Principle

Theorem

A and B are finite sets $\wedge$ $|A| > |B| \Rightarrow \nexists$ a one-to-one function from A to B.

- We shall prove the theorem by induction on $\mathtt{n} = |\mathtt{B}|$.

Types of Proofs

Pigeonhole Principle

Theorem

A and B are finite sets $\wedge$ $|A| > |B| \Rightarrow \nexists$ *a one-to-one function from A to B.*

- We shall prove the theorem by induction on $n = |B|$.
- Base Case: $n = 0$ (i.e., $B = \emptyset$)
  There is no function from A to B, because nothing can be mapped to elements of B. No function from A to B $\Rightarrow$ no one-to-one function from A to B.

Types of Proofs

Pigeonhole Principle

Theorem

A and B are finite sets $\wedge$ $|A| > |B| \Rightarrow \nexists$ a one-to-one function from A to B.

- We shall prove the theorem by induction on $\mathtt{n} = |\mathtt{B}|$.
- Base Case: $n = 0$ (i.e., $\mathtt{B} = \emptyset$)
  There is no function from $\mathtt{A}$ to $\mathtt{B}$, because nothing can be mapped to elements of $\mathtt{B}$. No function from $\mathtt{A}$ to $\mathtt{B} \Rightarrow$ no one-to-one function from $\mathtt{A}$ to $\mathtt{B}$.
- Inductive Step
  Assume: $\mathtt{f:A} \rightarrow \mathtt{B}$ is not a one-to-one function such that $|\mathtt{A}|>|\mathtt{B}|$, $|\mathtt{B}|\leq\mathtt{n}$, and $\mathtt{n} \geq 0$.
  We must show: $\mathtt{f:}\ \ \mathtt{A} \rightarrow \mathtt{B}$ is not a one-to-one function such that $|\mathtt{A}| > |\mathtt{B}| = \mathtt{n+1}$.

Types of Proofs

Pigeonhole Principle

Theorem

A and B are finite sets $\wedge$ *|A|* $>$ *|B|* $\Rightarrow \nexists$ *a one-to-one function from A to B.*

- We shall prove the theorem by induction on $n = |B|$.
- Base Case: $n = 0$ (i.e., $B = \emptyset$)
  There is no function from A to B, because nothing can be mapped to elements of B. No function from A to B $\Rightarrow$ no one-to-one function from A to B.
- Inductive Step
  Assume: $f:A \rightarrow B$ is not a one-to-one function such that $|A|>|B|$, $|B| \leq n$, and $n \geq 0$.
  We must show: $f: A \rightarrow B$ is not a one-to-one function such that $|A| > |B| = n+1$.
- Observe that A has at least 2 elements because $n+1 > 0$. Pick two distinct arbitrary elements, a and b, from A. If $f(a) = f(b)$ then f is not one-to-one (because two distinct elements of A map to the same element in B).

Types of Proofs

Pigeonhole Principle

Theorem

A and B are finite sets $\wedge$ $|A| > |B| \Rightarrow \nexists$ a one-to-one function from A to B.

- We shall prove the theorem by induction on $\mathtt{n} = |\mathtt{B}|$.
- Base Case: $\mathtt{n} = 0$ (i.e., $\mathtt{B} = \emptyset$)
  There is no function from A to B, because nothing can be mapped to elements of B. No function from A to B $\Rightarrow$ no one-to-one function from A to B.
- Inductive Step
  Assume: $\mathtt{f:A} \rightarrow \mathtt{B}$ is not a one-to-one function such that $|\mathtt{A}|>|\mathtt{B}|$, $|\mathtt{B}|\leq\mathtt{n}$, and $\mathtt{n} \geq 0$.
  We must show: $\mathtt{f:}\ \mathtt{A} \rightarrow \mathtt{B}$ is not a one-to-one function such that $|\mathtt{A}| > |\mathtt{B}| = \mathtt{n+1}$.
- Observe that A has at least 2 elements because $\mathtt{n+1} > 0$. Pick two distinct arbitrary elements, a and b, from A. If $\mathtt{f(a)} = \mathtt{f(b)}$ then f is not one-to-one (because two distinct elements of A map to the same element in B).
- If $\mathtt{f(a)} \neq \mathtt{f(b)}$ suppose that a is the only element mapped to f(a). Consider the sets $\mathtt{A'} = \mathtt{A} - \{\mathtt{a}\}$ and $\mathtt{B'} = \mathtt{B} - \{\mathtt{f(a)}\}$ and a function f' such that $\forall\ \mathtt{x} \in \mathtt{A'}\ \mathtt{f'(x)} = \mathtt{f(x)}$. The inductive hypothesis applies because $|\mathtt{B'}| = \mathtt{n}$ and $|\mathtt{A'}| > |\mathtt{B'}|$. This means that there are two distinct elements in A' that are mapped by f' to the same element of B'. Given that f agrees with f' on all elements, it follows that f is not one-to-one.

Types of Proofs

Pigeonhole Principle

Theorem

Let G be a graph with n nodes. Any path with n edges has a repeated node.

-

Types of Proofs

Pigeonhole Principle

Theorem

Let G be a graph with n nodes. Any path with n edges has a repeated node.

-

Proof.

Every edge connects two nodes (not necessarily distinct nodes). This means that `n` edges connect `n+1` nodes. By the pigeonhole principle there is no one-to-one function from the nodes in the path (the pigeons) to the nodes in the graph (the pigeonholes). Therefore, we may conclude that there is at least one node repeated in the path. $\square$

-

Types of Proofs

Pigeonhole Principle

- HOMEWORK: 14–16

Types of Proofs

Proofs by Contradiction

- A proof by contradiction proves that a statement holds by showing that assuming that the statement is false is absurd
- Also known as *reductio ad absurdum*

Types of Proofs

Proofs by Contradiction

- A proof by contradiction proves that a statement holds by showing that assuming that the statement is false is absurd
- Also known as *reductio ad absurdum*
- It is based on the observation that a statement cannot be both true and false

Types of Proofs

Proofs by Contradiction

- A proof by contradiction proves that a statement holds by showing that assuming that the statement is false is absurd
- Also known as *reductio ad absurdum*
- It is based on the observation that a statement cannot be both true and false
- A proof by contradiction establishes that the negation of a statement leads to such a contradiction
- Assuming `S` is false leads to concluding that a statement `A`, that we know to be false, is true
- This means that our assumption must be wrong and, therefore, `S` must be true.

Types of Proofs

Proofs by Contradiction

Theorem

$\sqrt{2}$ *is an irrational number*

Types of Proofs

Proofs by Contradiction

Theorem

$\sqrt{2}$ is an irrational number

- Assume $\sqrt{2}$ is a rational number

Types of Proofs

Proofs by Contradiction

Theorem

$\sqrt{2}$ is an irrational number

- Assume $\sqrt{2}$ is a rational number
- If $\sqrt{2}$ is rational then it can be expressed as a fraction, $\frac{a}{b}$, in lowest terms for `a,b` $\in \mathbb{Z}$

Types of Proofs

Proofs by Contradiction

Theorem

$\sqrt{2}$ is an irrational number

- Assume $\sqrt{2}$ is a rational number
- If $\sqrt{2}$ is rational then it can be expressed as a fraction, $\frac{a}{b}$, in lowest terms for a,b $\in \mathbb{Z}$
- Observe that at least one of a and b must be odd
- Consider: $\frac{a}{b} = \sqrt{2}$

  $\frac{a^2}{b^2} = 2$

  $a^2 = 2b^2$
- This means that a^2 is even

Types of Proofs

Proofs by Contradiction

Theorem

$\sqrt{2}$ is an irrational number

- Assume $\sqrt{2}$ is a rational number
- If $\sqrt{2}$ is rational then it can be expressed as a fraction, $\frac{a}{b}$, in lowest terms for `a,b` $\in \mathbb{Z}$
- Observe that at least one of `a` and `b` must be odd
- Consider: $\frac{a}{b} = \sqrt{2}$

  $\frac{a^2}{b^2} = 2$

  $a^2 = 2b^2$
- This means that a^2 is even
- Observe that this also means that `a` is even because `a * a` must be divisible by 2 (if `a` were odd then `a * a` would not be divisible by 2)

Types of Proofs

Proofs by Contradiction

Theorem

$\sqrt{2}$ *is an irrational number*

- Assume $\sqrt{2}$ is a rational number
- If $\sqrt{2}$ is rational then it can be expressed as a fraction, $\frac{a}{b}$, in lowest terms for `a,b` $\in \mathbb{Z}$
- Observe that at least one of `a` and `b` must be odd
- Consider: $\frac{a}{b} = \sqrt{2}$
  $\frac{a^2}{b^2} = 2$
  $a^2 = 2b^2$
- This means that a^2 is even
- Observe that this also means that `a` is even because `a * a` must be divisible by 2 (if `a` were odd then `a * a` would not be divisible by 2)
- Given that `a` is even and $\frac{a}{b}$ is in lowest terms, `b` must be odd (otherwise, `a` and `b` have 2 as a common factor)

Types of Proofs

Proofs by Contradiction

Theorem

$\sqrt{2}$ is an irrational number

- Assume $\sqrt{2}$ is a rational number
- If $\sqrt{2}$ is rational then it can be expressed as a fraction, $\frac{a}{b}$, in lowest terms for `a,b` $\in \mathbb{Z}$
- Observe that at least one of `a` and `b` must be odd
- Consider: $\frac{a}{b} = \sqrt{2}$
  $\frac{a^2}{b^2} = 2$
  $a^2 = 2b^2$
- This means that a^2 is even
- Observe that this also means that `a` is even because `a * a` must be divisible by 2 (if `a` were odd then `a * a` would not be divisible by 2)
- Given that `a` is even and $\frac{a}{b}$ is in lowest terms, `b` must be odd (otherwise, `a` and `b` have 2 as a common factor)
- Observe that a^2 is a multiple of 4:
  $a^2 = a * a = 2j * 2j = 4j^2$

Types of Proofs

Proofs by Contradiction

Theorem

$\sqrt{2}$ is an irrational number

- Assume $\sqrt{2}$ is a rational number
- If $\sqrt{2}$ is rational then it can be expressed as a fraction, $\frac{a}{b}$, in lowest terms for `a,b` $\in \mathbb{Z}$
- Observe that at least one of `a` and `b` must be odd
- Consider: $\frac{a}{b} = \sqrt{2}$
  $\frac{a^2}{b^2} = 2$
  $a^2 = 2b^2$
- This means that a^2 is even
- Observe that this also means that `a` is even because `a * a` must be divisible by 2 (if `a` were odd then `a * a` would not be divisible by 2)
- Given that `a` is even and $\frac{a}{b}$ is in lowest terms, `b` must be odd (otherwise, `a` and `b` have 2 as a common factor)
- Observe that a^2 is a multiple of 4:
  $a^2 = a * a = 2j * 2j = 4j^2$
- This means that $2b^2$ is a multiple of 4. We observe that `b` must be even given that if it were odd `b` could equal 3 which means $2b^2$ is not a multiple of 4 (i.e., $2(3)^2 = 18$ which is not a multiple of 4)

Types of Proofs

Proofs by Contradiction

Theorem

$\sqrt{2}$ is an irrational number

- Assume $\sqrt{2}$ is a rational number
- If $\sqrt{2}$ is rational then it can be expressed as a fraction, $\frac{a}{b}$, in lowest terms for `a,b` $\in \mathbb{Z}$
- Observe that at least one of `a` and `b` must be odd
- Consider: $\frac{a}{b} = \sqrt{2}$
  $\frac{a^2}{b^2} = 2$
  $a^2 = 2b^2$
- This means that a^2 is even
- Observe that this also means that `a` is even because `a * a` must be divisible by 2 (if `a` were odd then `a * a` would not be divisible by 2)
- Given that `a` is even and $\frac{a}{b}$ is in lowest terms, `b` must be odd (otherwise, `a` and `b` have 2 as a common factor)
- Observe that a^2 is a multiple of 4:
  $a^2 = a * a = 2j * 2j = 4j^2$
- This means that $2b^2$ is a multiple of 4. We observe that `b` must be even given that if it were odd `b` could equal 3 which means $2b^2$ is not a multiple of 4 (i.e., $2(3)^2 = 18$ which is not a multiple of 4)
- It is impossible, however, for `b` to be both odd and even
- Our assumption cannot be true and we may conclude that $\sqrt{2}$ is an irrational number.

Types of Proofs

Proofs by Contradiction

- HOMEWORK: 17–18

Types of Proofs

Diagonalization Proofs

- A binary relation on a set `A` may be visualized as a matrix whose rows and columns are labeled with the elements of `A`
- If `i` is related to `j` then the entry in row `i` and column `j` contains an `x`

Types of Proofs

Diagonalization Proofs

- A binary relation on a set `A` may be visualized as a matrix whose rows and columns are labeled with the elements of `A`
- If `i` is related to `j` then the entry in row `i` and column `j` contains an `x`
-

	a	b	c	d	e	f	g
a	x		x				
b				x		x	x
c		x	x				x
d	x					x	
e				x			
f	x	x			x		
g					x		x

Types of Proofs

Diagonalization Proofs

-

	a	b	c	d	e	f	g
a	x		x				
b				x		x	x
c		x	x				x
d	x					x	
e				x			
f	x	x			x		
g					x		x

- The main diagonal of the visualization of binary relation defines two sets: those elements that are related to themselves and those elements that are not related to themselves

Types of Proofs

Diagonalization Proofs

-

	a	b	c	d	e	f	g
a	x		x				
b				x		x	x
c		x	x				x
d	x					x	
e				x			
f	x	x			x		
g					x		x

- The main diagonal of the visualization of binary relation defines two sets: those elements that are related to themselves and those elements that are not related to themselves
- Formally:

$$\mathtt{D} = \{\mathtt{a} \mid (\mathtt{a},\ \mathtt{a}) \in \mathtt{R}\} \qquad \hat{\mathtt{D}} = \{\mathtt{a} \mid (\mathtt{a},\ \mathtt{a}) \notin \mathtt{R}\}$$

Types of Proofs

Diagonalization Proofs

	a	b	c	d	e	f	g
a	x		x				
b				x		x	x
c		x	x				x
d	x					x	
e				x			
f	x	x			x		
g					x		x

- The main diagonal of the visualization of binary relation defines two sets: those elements that are related to themselves and those elements that are not related to themselves
- Formally:

  $$\mathtt{D} = \{\mathtt{a} \mid (\mathtt{a},\ \mathtt{a}) \in \mathtt{R}\} \qquad \hat{\mathtt{D}} = \{\mathtt{a} \mid (\mathtt{a},\ \mathtt{a}) \notin \mathtt{R}\}$$

- The *diagonalization principle* states that $\hat{\mathtt{D}}$ is not equal to any row in the visualization: $\hat{\mathtt{D}} \neq \mathtt{R}_{\mathtt{a}}$

Types of Proofs

Diagonalization Proofs

-

	a	b	c	d	e	f	g
a	x		x				
b				x		x	x
c		x	x				x
d	x					x	
e				x			
f	x	x			x		
g					x		x

- The main diagonal of the visualization of binary relation defines two sets: those elements that are related to themselves and those elements that are not related to themselves
- Formally:

$$\mathtt{D} = \{\mathtt{a} \mid (\mathtt{a},\ \mathtt{a}) \in \mathtt{R}\} \qquad \hat{\mathtt{D}} = \{\mathtt{a} \mid (\mathtt{a},\ \mathtt{a}) \notin \mathtt{R}\}$$

- The *diagonalization principle* states that $\hat{\mathtt{D}}$ is not equal to any row in the visualization: $\hat{\mathtt{D}} \neq \mathtt{R}_{\mathtt{a}}$
- The diagonalization principle is used as part of a proof by contradiction
- A statement is assumed to be true and diagonalization is used to develop a contradiction

Types of Proofs

Diagonalization Proofs

- Prove that the real numbers in `(0..1)` are uncountable
- We shall use a well-known fact for computer scientists: every real number in `(0..1)` may be written as a binary number of infinite length

Types of Proofs

Diagonalization Proofs

Theorem

The set of real numbers in `(0..1)` *is uncountable.*

Types of Proofs

Diagonalization Proofs

Theorem

The set of real numbers in `(0..1)` *is uncountable.*

Proof.

Assume the set of real numbers in `(0..1)` is countable.
This means there exists a program to print these numbers that eventually prints any arbitrary binary digit of any real number in `(0..1)`. The printing of these real numbers looks as follows:

```
1. .0 0 0 1 0 1 1 ...
2. .1 1 0 1 0 0 0 ...
3. .0 1 0 0 0 0 0 ...
4. .0 1 1 0 0 0 1 ...
5. .0 0 0 0 1 1 0 ...
6. .1 1 1 1 1 1 1 ...
7. .1 1 0 0 0 0 0 ...
      ⋮
```

Observe that the binary digits form a matrix. Consider the real number represented by $\hat{\mathtt{D}}$. $\hat{\mathtt{D}}$ cannot ever equal an arbitrary row `i` in the matrix of printed numbers because their $\mathtt{i}^{\mathtt{th}}$ bits differ. This means that the real number represented by $\hat{\mathtt{D}}$ is never be printed. This contradicts the assumption made that the set of real numbers in `(0..1)` are countable and, therefore, the set of real numbers in `(0..1)` is uncountable. □

Types of Proofs

Diagonalization Proofs

- Recall that in the previous chapter we tabled the discussion on whether or not $\mathbb{R}$, and, $\mathbb{Z}$ are equinumerous

Types of Proofs

Diagonalization Proofs

- Recall that in the previous chapter we tabled the discussion on whether or not $\mathbb{R}$, and, $\mathbb{Z}$ are equinumerous
- Observe that the previous proof means that there is no bijection between $\mathbb{N}$ and $\mathbb{R}$
- That is, $\mathbb{N}$ and $\mathbb{R}$ are not equinumerous

Types of Proofs

Diagonalization Proofs

- Recall that in the previous chapter we tabled the discussion on whether or not $\mathbb{R}$, and, $\mathbb{Z}$ are equinumerous
- Observe that the previous proof means that there is no bijection between $\mathbb{N}$ and $\mathbb{R}$
- That is, $\mathbb{N}$ and $\mathbb{R}$ are not equinumerous
- We know from the previous chapter that $\mathbb{N}$ and $\mathbb{Z}$ are equinumerous

Types of Proofs

Diagonalization Proofs

- Recall that in the previous chapter we tabled the discussion on whether or not $\mathbb{R}$, and, $\mathbb{Z}$ are equinumerous
- Observe that the previous proof means that there is no bijection between $\mathbb{N}$ and $\mathbb{R}$
- That is, $\mathbb{N}$ and $\mathbb{R}$ are not equinumerous
- We know from the previous chapter that $\mathbb{N}$ and $\mathbb{Z}$ are equinumerous
- Therefore, their does not exist a bijection between $\mathbb{R}$ and $\mathbb{Z}$
- These sets are not equinumerous

Types of Proofs

Diagonalization Proofs

- HOMEWORK: 19–20
- QUIZ: 21 (due in 1 week)