Golang FP

Functional Programming (FP: Beginer)

(Integrative-Generative AI Edition)

Author: Student Price Book Center

คำนำ

การเขียนโปรแกรมเชิงฟังก์ชัน (Functional Programming – FP) ได้รับความนิยมเพิ่มมากขึ้นในโลกของซอฟต์แวร์สมัยใหม่ เนื่องจากช่วยให้การพัฒนาโปรแกรมมีความคงที่ ทดสอบง่าย และลดความซับซ้อนในการจัดการ state หรือ side effects แม้ภาษา Go จะถูกออกแบบให้เป็นภาษา imperative และ concurrent-first แต่ Go ก็มีคุณสมบัติและเครื่องมือพื้นฐานหลายอย่างที่ช่วยให้นักพัฒนาสามารถนำแนวคิดของ FP มาใช้ได้อย่างมีประสิทธิภาพ หนังสือเล่มนี้ถูกออกแบบขึ้นเพื่อเป็นคู่มือ **Beginner-Friendly** สำหรับผู้ที่ต้องการเรียนรู้ FP ผ่านภาษา Go โดยเน้นความเข้าใจเชิงลึกควบคู่ไปกับตัวอย่างปฏิบัติจริง

หนังสือเล่มนี้แบ่งเนื้อหาออกเป็น 5 บทหลัก โดยเริ่มตั้งแต่แนวคิดพื้นฐานของ Functional Programming ใน Go จนถึงเทคนิคเชิงปฏิบัติและโครงสร้างข้อมูลแบบ functional ในแต่ละบท เราจะพาผู้อ่านผ่านการทำความเข้าใจ **ความหมายของ FP, ความแตกต่างระหว่าง Imperative และ Declarative, ฟังก์ชันบริสุทธิ์ (Pure Functions) และ Side Effects** รวมถึงข้อดีและข้อจำกัดของ FP ในบริบทของ Go การเรียนรู้แบบค่อยเป็นค่อยไปนี้ช่วยให้ผู้อ่านสามารถจับแนวคิดพื้นฐานและประยุกต์ใช้กับโค้ดจริงได้ทันที

บทต่อมาเน้นการเรียนรู้เกี่ยวกับ **First-Class Functions** ซึ่งเป็นคุณสมบัติสำคัญของภาษา Go ที่รองรับแนวคิดเชิงฟังก์ชัน ฟังก์ชันใน Go สามารถถูกเก็บในตัวแปร ส่งเป็น argument ให้กับฟังก์ชันอื่น หรือคืนค่าจากฟังก์ชันได้ ทำให้โค้ดมีความยืดหยุ่นและ modular มากขึ้น ผู้อ่านจะได้ฝึกสร้างฟังก์ชันที่ reusable และเรียนรู้การออกแบบโค้ดในรูปแบบ Higher-Order Functions (HOFs) ซึ่งเป็นพื้นฐานสำคัญของการเขียน FP ที่ซับซ้อนขึ้น

จากนั้นเราจะเข้าสู่หัวข้อ **Anonymous Functions / Lambda** ซึ่งช่วยให้โค้ดกระชับและชัดเจนยิ่งขึ้น ฟังก์ชันที่ไม่มีชื่อสามารถประกาศและใช้งานได้ทันที ทำให้เหมาะกับการใช้งานแบบ inline และสามารถส่งไปยัง higher-order functions เพื่อปรับพฤติกรรมของโปรแกรมตามบริบทโดยไม่ต้องสร้างฟังก์ชันแยก ผู้อ่านจะได้เรียนรู้การสร้าง lambda expressions, inline functions และการใช้ anonymous functions ใน Go พร้อมตัวอย่างที่สามารถนำไปประยุกต์ใช้ได้ทันที

Immutability เป็นอีกหนึ่งหัวข้อสำคัญที่หนังสือเล่มนี้จะพาผู้อ่านไปทำความเข้าใจ แม้ Go จะไม่มี immutability built-in แบบบางภาษา แต่เราสามารถประยุกต์ใช้ const, struct แบบ read-only หรือแนวทาง copy-on-write เพื่อหลีกเลี่ยงการเปลี่ยนแปลง state ของ slice, map และ struct การรักษาความคงที่ของข้อมูลช่วยลด side effects เพิ่มความปลอดภัย และทำให้โค้ดคาดเดาได้ง่ายขึ้น ผู้อ่านจะได้เรียนรู้ทั้งแนวคิดและเทคนิคปฏิบัติที่จะนำไปสร้างโปรแกรม functional ที่ปลอดภัยและ maintainable บทสุดท้ายจะพาผู้อ่านไปรู้จัก **Built-in Functional Tools** ของ Go เช่น การใช้ range ใน loops แบบ declarative และการสร้างฟังก์ชัน map, filter, reduce แบบ custom functions นอกจากนี้ยังครอบคลุมการใช้งาน slices และ arrays ในลักษณะ functional การรวมเทคนิคเหล่านี้ช่วยให้การประมวลผลข้อมูล

มีความยืดหยุ่น ชัดเจน และสื่อความหมายเชิงฟังก์ชันได้ครบถ้วน พร้อมตัวอย่าง Ultimate Project และ Integration Examples ที่สามารถรันได้จริง

หนังสือเล่มนี้ออกแบบให้ผู้อ่านสามารถ **เรียนรู้ควบคู่กับการปฏิบัติจริง** ด้วยตัวอย่างบูรณาการ Ultimate Project และ Super Ultimate Project ในแต่ละบท เพื่อให้เห็นภาพรวมของ FP ใน Go ตั้งแต่แนวคิดพื้นฐานจนถึงการสร้างระบบที่ซับซ้อน การเรียนรู้แบบลงมือทำนี้จะช่วยให้ผู้เริ่มต้นสามารถเข้าใจ FP อย่างเป็นระบบ พร้อมทั้งสามารถประยุกต์แนวคิดนี้ในงานจริงได้อย่างมั่นใจ

ท้ายที่สุด หนังสือเล่มนี้ไม่ได้มุ่งเพียงแค่สอน syntax ของ Go แต่ยังเน้นการพัฒนาทักษะ **การคิดแบบเชิงฟังก์ชัน** (Functional Thinking) ซึ่งเป็นคุณสมบัติสำคัญของนักพัฒนาซอฟต์แวร์สมัยใหม่ ผู้ที่อ่านหนังสือเล่มนี้จนจบจะสามารถเขียนโค้ด Go แบบ functional ที่อ่านง่าย maintainable และพร้อมรับมือกับโปรแกรมซับซ้อน พร้อมทั้งมีพื้นฐานที่มั่นคงสำหรับการเรียนรู้หัวข้อระดับกลางและขั้นสูงในอนาคต

ด้วยความปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 1

Introduction to Functional Programming in Go

(Introduction to Functional Programming in Go)

เนื้อหา

บทที่ 1: Introduction to Functional Programming in Go

ในโลกของการเขียนโปรแกรมสมัยใหม่ แนวคิดของ “Functional Programming” หรือ **FP** ได้รับความสนใจเพิ่มขึ้นอย่างต่อเนื่อง เนื่องจากสามารถช่วยให้นักพัฒนาสร้างซอฟต์แวร์ที่มีความยืดหยุ่นสูง ทดสอบง่าย และลดความซับซ้อนของโค้ดได้อย่างมีประสิทธิภาพ ภาษา Go ซึ่งเป็นภาษาที่เน้นความเรียบง่ายและประสิทธิภาพ ก็สามารถประยุกต์แนวคิดของ FP เข้ามาใช้ได้เช่นกัน แม้ว่าจะไม่ใช่ภาษาฟังก์ชันนิยมโดยกำเนิด แต่ Go ก็เปิดโอกาสให้ผู้พัฒนาสามารถใช้แนวคิดเหล่านี้เพื่อปรับปรุงรูปแบบการเขียนโปรแกรมให้ชัดเจนและเป็นระบบมากขึ้น

ก่อนอื่น เราจำเป็นต้องทำความเข้าใจว่า **Functional Programming คืออะไร** FP เป็นแนวคิดในการเขียนโปรแกรมที่มุ่งเน้นการใช้ “ฟังก์ชันบริสุทธิ์” (pure functions) และการหลีกเลี่ยงการเปลี่ยนแปลงสถานะ (state mutation) ภายนอก โดยให้ความสำคัญกับการประกาศ “สิ่งที่ต้องการให้โปรแกรมทำ” มากกว่าการกำหนด “วิธีการทีละขั้นตอน” เพื่อให้ได้ผลลัพธ์นั้น ๆ แนวคิดนี้ช่วยให้โค้ดมีความคงที่ คาดเดาได้ และสามารถนำฟังก์ชันกลับมาใช้ซ้ำได้โดยไม่เกิดผลข้างเคียงที่ไม่พึงประสงค์

เมื่อเปรียบเทียบกับแนวทางแบบ **Imperative** ซึ่งเป็นแนวทางที่ Go ใช้โดยทั่วไป ความแตกต่างสำคัญจะอยู่ที่มุมมองของนักพัฒนา โปรแกรมแบบ Imperative จะบอกคอมพิวเตอร์ว่า “ทำอะไร” ทีละขั้น เช่น การวนลูป การแก้ไขตัวแปร และการจัดการโครงสร้างข้อมูล ในขณะที่โปรแกรมแบบ **Declarative** จะมุ่งเน้นการบอกว่า “ต้องการให้เกิดผลลัพธ์แบบใด” โดยปล่อยให้รายละเอียดการ

ดำเนินการเป็นหน้าที่ของระบบหรือกลไกภายใน วิธี Declarative จึงมักทำให้โค้ดมีความกระชับและอ่านง่ายกว่า ซึ่ง FP เป็นแนวทางหนึ่งที่อยู่ในกลุ่มของ Declarative programming

หัวใจสำคัญของ FP คือ **Pure Functions** หรือฟังก์ชันบริสุทธิ์ ซึ่งเป็นฟังก์ชันที่ให้ผลลัพธ์เหมือนเดิมเสมอเมื่อป้อนค่าพารามิเตอร์ชุดเดียวกัน และไม่มีการเปลี่ยนแปลงข้อมูลภายนอกหรือ state ภายนอกใด ๆ สิ่งนี้ตรงข้ามกับฟังก์ชันที่มี **Side Effects** เช่น การเขียนไฟล์ การเปลี่ยนค่าตัวแปร global หรือการพิมพ์ออกหน้าจอ การควบคุม side effects ให้อยู่ในขอบเขตที่จำกัดเป็นหัวใจของการทำให้โค้ดสามารถทดสอบและบำรุงรักษาได้ง่าย

ในภาษา Go เราสามารถนิยามและใช้งาน pure functions ได้อย่างง่ายดายผ่านการประกาศฟังก์ชันทั่วไปและการคืนค่าที่ชัดเจน นอกจากนี้ Go ยังสนับสนุนการส่งฟังก์ชันเป็นพารามิเตอร์หรือการคืนค่าฟังก์ชัน (higher-order functions) ซึ่งเป็นอีกหนึ่งคุณสมบัติสำคัญของ FP ถึงแม้ Go จะไม่มีฟีเจอร์บางอย่างเช่น pattern matching หรือ lazy evaluation ที่พบในภาษาฟังก์ชันนิยมแท้ ๆ แต่ก็มีเครื่องมือพื้นฐานเพียงพอสำหรับการสร้างโค้ดเชิงฟังก์ชันอย่างมีประสิทธิภาพ

ข้อดีของการนำ FP มาใช้ใน Go ได้แก่ การทำให้โค้ดอ่านง่ายและบำรุงรักษาได้ดีขึ้น ลดจำนวนบั๊กที่เกิดจากการเปลี่ยนแปลงสถานะร่วม (shared state) เพิ่มความสามารถในการทดสอบหน่วย (unit testing) และเอื้อต่อการทำงานแบบขนาน (concurrency) ได้เป็นอย่างดี นอกจากนี้ยังช่วยให้การแยกส่วนประกอบของโปรแกรม (modularization) เป็นไปอย่างชัดเจนมากขึ้น ทำให้สามารถนำโค้ดกลับมาใช้ซ้ำได้ง่าย

อย่างไรก็ตาม FP ใน Go ก็มีข้อจำกัด เช่น การขาดเครื่องมือและ syntax บางอย่างที่สนับสนุนการเขียนโค้ดเชิงฟังก์ชันอย่างลึกซึ้ง เช่น generics แบบยืดหยุ่นก่อนเวอร์ชัน 1.18 การขาด immutability แบบ built-in และไม่มีฟีเจอร์บางอย่างที่พบใน Haskell หรือ Scala ดังนั้น นักพัฒนาจึงต้องประยุกต์แนวคิด FP อย่างมีสติและรู้ขอบเขตของภาษาด้วย บทนี้จะพาคุณไปสำรวจแนวคิดพื้นฐานทั้งหมดเหล่านี้ และเรียนรู้วิธีนำมาประยุกต์ใช้กับภาษา Go ได้อย่างเหมาะสม

Introduction to Functional Programming in Go

- ความหมาย FP
- Imperative vs Declarative
- Pure functions และ Side Effects
- ข้อดีและข้อจำกัดของ FP ใน Go

ความหมายของ FP (Functional Programming)

Functional Programming (FP) คือแนวทางเขียนโปรแกรมที่เน้นการคำนวณโดยใช้ *ฟังก์ชันเป็นหน่วยพื้นฐาน* ของการทำงาน หลักการสำคัญได้แก่:

- **ฟังก์ชันเป็น first-class values** — ฟังก์ชันถูกส่งเป็นพารามิเตอร์ คืนค่า หรือตีเก็บในตัวแปรได้

- **Pure functions** — ฟังก์ชันที่ให้ผลลัพธ์เดียวกันเสมอเมื่อรับอินพุตเดียวกัน และไม่ทำ side-effects (เช่น ไม่เปลี่ยนสถานะภายนอก, ไม่พิมพ์, ไม่อ่าน/เขียนไฟล์)
- **Immutability by discipline** — ข้อมูลมักไม่ถูกแก้ไข แต่สร้างอ็อบเจกต์/ค่าชุดใหม่เมื่อแปลงข้อมูล
- **Higher-order functions** — ฟังก์ชันที่รับหรือคืนฟังก์ชัน เช่น map, filter, reduce
- **Composition** — สร้างฟังก์ชันจากการรวมฟังก์ชันย่อย ๆ เพื่อสร้าง pipeline ของการประมวลผล

ในโลกของภาษา functional แบบ pure (เช่น Haskell) หลายอย่างถูกบังคับโดยภาษา แต่ใน Go เราทำ **แบบมีวินัย (by-discipline)** — Go ให้ความยืดหยุ่นมากกว่า แต่ไม่มีการบังคับ immutability หรือ pattern-matching แบบ native

---

Imperative vs Declarative — ต่างกันอย่างไร และตัวอย่างใน Go

Imperative (เชิงคำสั่ง) — บอกคอมพิวเตอร์ “จะทำอะไรเป็นขั้นตอน”

ตัวอย่าง: ใช้ for loop เพื่อวนและเปลี่ยนค่า

```
// Imperative: double and keep only even numbers
res := []int{}
for _, v := range nums {
    if v%2 == 0 {
        res = append(res, v*2)
    }
}
```

Declarative (เชิงประกาศ) — บอก “อะไรที่ต้องการให้ได้” มากกว่าจะบอกขั้นตอนละเอียด

ตัวอย่าง: ใช้ map + filter (implemented as functions) เพื่อประกาศการเปลี่ยนแปลง

```
doubled := Map(Filter(nums, func(x int) bool { return x%2==0 }), func(x int) int { return x*2 })
```

ข้อแตกต่างเชิงปฏิบัติ

- Imperative มักเข้าใจง่ายสำหรับคนที่คุ้นกับการจัดการ state และมักเร็วกว่าในบางกรณี (เพราะลดการ allocate)
- Declarative/functional เน้น readability, composability, และเหตุผลเชิงตรรกะ — เหมาะกับการทดสอบและ reasoning แต่บางครั้งมี overhead (allocate, copy) หากทำไม่ระวัง

Pure Functions และ Side Effects — ความหมายเชิงลึก + ตัวอย่างใน Go

Pure functions

คุณสมบัติ:

1. ผลลัพธ์ขึ้นอยู่กับพารามิเตอร์อินพุตเท่านั้น

2. ไม่มีผลข้างเคียงภายนอก (no side-effects)

ตัวอย่าง (pure):

```
func Add(a, b int) int {
    return a + b
}
```

ตัวอย่าง generic Map (pure):

```
func Map[T any, R any](s []T, f func(T) R) []R {
    out := make([]R, len(s))
    for i, v := range s {
        out[i] = f(v)
    }
    return out
}
```

Map ข้างบนเป็น *pure* ถ้า f เป็น pure — มันไม่แก้ไขสไลซ์ต้นฉบับ แต่คืนสไลซ์ใหม่

Side effects (ผลข้างเคียง)

ตัวอย่าง side-effects:

- การเขียน/อ่านไฟล์, network I/O, console printing
- การแก้ไขตัวแปร global หรือ data structure ที่ถูกแชร์
- การเรียก time.Now() หรือสุ่มที่ให้ค่าต่างกันเมื่อเรียกหลายครั้ง
- การ Panic — เปลี่ยน control flow ภายนอก

ตัวอย่าง impure (มี side-effect — แก้ไขสไลซ์ต้นฉบับ):

```
func DoubleInPlace(nums []int) {
    for i := range nums {
        nums[i] *= 2 // mutate the input slice
    }
}
```

หรือฟังก์ชันที่พิมพ์ออกหน้าจอ:

```
func PrintAndDouble(x int) int {
    fmt.Println("processing", x) // side effect
    return x * 2
}
```

ทำไม pure สำคัญ

- **Referential transparency**: ถ้า f เป็น pure, เราสามารถแทนการเรียก f(x) ด้วยผลลัพธ์คำนวณไว้ล่วงหน้าได้ — ช่วย reasoning & memoization

- **ทดสอบง่าย**: เพียงใส่ input เดียวก็ยืนยัน output ได้โดยไม่ต้อง mock IO หรือ global state
- **ปลอดภัยในการรันพร้อมกัน (concurrency-safe)**: ถ้าไม่มี shared mutable state จะปลอดภัยใน goroutine หลายตัว

ข้อดีของการใช้ FP ใน Go

1. **Modularity & Composability** — ฟังก์ชันเล็ก ๆ ที่รวมกันได้ง่าย ทำให้โค้ดอ่านเป็น flow ของการแปลงข้อมูล
2. **Testability** — Pure function ง่ายต่อการ unit-test (no mocks)
3. **Reasoning & Refactoring** — ลดขอบเขตของ state ทำให้หา bug ได้เร็วขึ้น
4. **Concurrency synergy** — การออกแบบเป็น pure transformations + channels/goroutines ทำให้สร้าง pipelines ที่ parallel ได้ง่ายและปลอดภัย
5. **Reusable utilities** — ด้วย generics (Go 1.18+) เราสามารถเขียน Map/Filter/Reduce แบบ reusable ได้สะดวก
6. **Separation of concerns** — แยก logic (pure) ออกจาก side effects (IO) — ดีต่อ maintainability

---

ข้อจำกัด / ข้อควรระวังของ FP ใน Go

1. **ไม่มี immutability บังคับ** — Go ไม่บังคับให้ข้อมูลเป็น immutable; การรักษา immutability ต้องทำด้วยวินัย (copying)
    - การ copy ข้อมูล (เช่น สไลซ์ขนาดใหญ่) อาจเพิ่ม overhead ทางหน่วยความจำและ GC
2. **ไม่ใช่ภาษา functional-first** — ขาด syntactic sugar ที่ FP languages มี เช่น pattern matching, currying, lazy evaluation, algebraic data types (ADTs)
3. **ไม่มี guarantee tail-call optimization** — recursion แบบลึกอาจทำให้ stack overflow; ใช้ loops หรือ explicit stacks ใน hot paths
4. **API/standard library เป็นแนว imperative** — หลาย API ใน ecosystem คาดหวังการแก้ไข state หรือใช้งาน callback แบบ imperative
5. **อาจไม่ idiomatic ในทีม** — โค้ด functional ที่ "เกินไป" อาจอ่านยากสำหรับทีมที่คุ้นกับ style แบบ imperative ของ Go
6. **ความซับซ้อนในการจัดการ error** — Go ใช้ pattern (T, error) ซึ่งทำให้การคอมโพสชันของฟังก์ชันที่ส่งกลับ error ต้องมี helper/adapter (หรือ monad-like patterns) เพื่อความสะดวก
7. **Overhead ทางประสิทธิภาพ** — การสร้างอ็อบเจกต์ใหม่/allocate มาก ๆ อาจแพ้การ mutate in-place ใน hot loops — ต้อง profile ก่อนตัดสินใจ

หมายเหตุเชิงทันสมัย: การเพิ่ม generics ทำให้การเขียน utilities แบบ FP ง่ายขึ้น แต่ก็ยังไม่มี ADTs/pattern-matching ในตัว — ทำให้บาง pattern ต้องจำลองเองหรือใช้โครงสร้างข้อมูลเฉพาะ

แนวทางปฏิบัติ (Practical tips / Best practices)

- **แยก pure logic ออกจาก side-effects** — ให้ฟังก์ชัน core คืนค่าเท่านั้น แล้วเรียก IO ที่ edge ของระบบ
- **ใช้ copy เมื่อจำเป็น** — ถ้าต้องการ immutability ให้ชัดเจน ให้ทำ copy (เช็ก performance)
- **ใช้ generics สำหรับ utilities** — เขียน Map, Filter, Reduce, MapE (with error) แบบ generic เพื่อ reuse
- **ใช้ channels/goroutines สำหรับ streaming pipelines** — ให้ฟังก์ชันแปลงข้อมูลเป็น pure แล้วใช้ concurrency เพื่อ parallelism
- **ระวัง closures ที่ capture state** — closures สร้าง stateful behavior — ดีแต่ต้องระวัง race-condition และความยากในการทดสอบ
- **ตรวจวัด (profile) ใน hot-paths** — ถ้าข้อมูลขนาดใหญ่หรือความเร็วสำคัญ ให้พิจารณา in-place mutation ในส่วนที่จำเป็นเท่านั้น
- **สร้าง combinator สำหรับ error** — เช่น MapE หรือ Bind เพื่อจัดการ pattern (T, error) ให้คอมโพสได้สะดวกขึ้น

ตัวอย่างสั้น — Map, Filter generic และ MapE (with error)

```
package main

import "fmt"

// Generic Map
func Map[T any, R any](s []T, f func(T) R) []R {
    out := make([]R, len(s))
    for i, v := range s {
        out[i] = f(v)
    }
    return out
}

// Generic Filter
func Filter[T any](s []T, pred func(T) bool) []T {
```

```
    out := make([]T, 0, len(s))
    for _, v := range s {
        if pred(v) {
            out = append(out, v)
        }
    }
    return out
}

// Map with error propagation
func MapE[T any, R any](s []T, f func(T) (R, error)) ([]R, error) {
    out := make([]R, 0, len(s))
    for _, v := range s {
        r, err := f(v)
        if err != nil {
            return nil, err
        }
        out = append(out, r)
    }
    return out, nil
}

func main() {
    nums := []int{1, 2, 3, 4, 5}
    doubled := Map(nums, func(x int) int { return x * 2 })
    evens := Filter(doubled, func(x int) bool { return x%2 == 0 })
    fmt.Println(doubled, evens)
}
```

สรุปสั้น ๆ (TL;DR)

- FP คือแนวคิดที่เน้น pure functions, composition และ immutability (โดยวินัยใน Go)
- **Imperative** บอก *วิธีทำ* (ขั้นตอน) — **Declarative/FP** บอก *สิ่งที่ต้องการให้ได้* (แปลงข้อมูลเป็น flow)
- **Pure functions** ลด complexity และเพิ่ม testability; **side effects** ให้คุมไว้ที่ edge ของระบบ

- ใน Go FP ให้ประโยชน์จริง โดยเฉพาะกับ data transformation และ concurrent pipelines — แต่ต้องระวังเรื่อง performance, ไม่มี immutability บังคับ และ ecosystem ยังมี API แบบ imperative มาก

บทที่ 1 — รายละเอียดเชิงลึก: Introduction to Functional Programming in Go

---

1) ความหมายของ FP (เชิงลึก)

- **แก่นหลัก**: FP มองการคำนวณเป็นการ *แปลงค่า* (transformations) โดยใช้ฟังก์ชันเป็นหน่วยหลัก ฟังก์ชันควรเล็ก กระชับ และนำกลับมาใช้ใหม่ได้ (reusable)
- **คุณสมบัติสำคัญ**: first-class functions, higher-order functions, pure functions, composition, immutability (โดยวินัย), declarative style
- **แนวคิดออกแบบ**: แยก *logic* (pure, deterministic) ออกจาก *side effects* (IO, logging, network) — วาง side effects ไว้ที่ขอบของระบบ (the edge) เท่านั้น
- **ภาษา vs แนวคิด**: บางภาษาทำ FP อย่างเข้มข้น (Haskell — pure), Go ให้เครื่องมือบางส่วน — ต้องออกแบบด้วยวินัย (by discipline)

---

2) Imperative vs Declarative (เชิงลึก + ข้อเทคนิค)

- **Imperative**: บอก "ทำอย่างไร" (how) — ขั้นตอน, การเปลี่ยน state, loops, mutation
    - เหมาะกับ: hot loops, low-level stateful algorithms, when you need minimal allocations
- **Declarative (functional style)**: บอก "ต้องการอะไร" (what) — เน้นการประกาศ transformations เช่น map ⟶ filter ⟶ reduce
    - เหมาะกับ: readability, composing transformations, testability, parallelizable flows

ตัวอย่างเปรียบเทียบ (imperative vs functional)

Imperative:

```
res := []int{}
for _, v := range nums {
    if v%2 == 0 {
        res = append(res, v*2)
    }
}
```

Functional-style (with helpers):

doubled := Map(Filter(nums, func(x int) bool { return x%2==0 }), func(x int) int { return x*2 })

เชิงปฏิบัติ

- Declarative มักสร้าง allocations (new slices) — ต้องพิจารณา performance (preallocate capacity, reuse buffers)
- Imperative อาจอ่านง่ายกว่าในบางทีม และลด allocation ได้ แต่เพิ่ม risk ของ bugs จาก mutable state

---

3) Pure functions และ Side Effects (เชิงลึก)

Pure function: นิยามเชิงเทคนิค

- ผลลัพธ์ขึ้นกับพารามิเตอร์เท่านั้น (deterministic)
- ไม่มีการเปลี่ยนแปลง state ภายนอก และไม่มี IO/การเปลี่ยน control flow ภายนอก (no side effects)
- ทำให้มี **referential transparency** — สามารถแทนการเรียกด้วยค่าคงที่ได้

ตัวอย่าง pure:

func Add(a, b int) int { return a + b }

Side effects (ตัวอย่าง)

- การเขียนไฟล์ / network call / printing
- แก้ไขตัวแปร global / shared memory
- ใช้เวลา/สุ่ม (time.Now(), rand) ที่ทำให้ผลแตกต่างกัน
- การ panic / os.Exit (เปลี่ยน control flow)

การจัดการ side effects

- แยกเป็นชั้น: core logic (pure) ⟶ adapter layer ⟶ IO
- ให้ pure functions คืนค่า (data) เท่านั้น แล้วจึงให้ outer layer ทำ IO
- ใน context concurrency ให้ pure logic ทำงานบน data copy แล้วให้ outer layer handle synchronization/IO

ตัวอย่าง pattern: pure core + IO edge

```
// Pure core
func ProcessRecord(r Record) Result { /* deterministic */ }

// IO edge
func ProcessAndSave(r Record, store Store) error {
    res := ProcessRecord(r)        // pure
    return store.Save(res)         // side-effect at edge
}
```

4) การนำ FP มาประยุกต์ใน Go — โค้ดตัวอย่างเชิงลึก

Generic Map / Filter / Reduce (Go generics)

```
package fp

// Map
func Map[T any, R any](s []T, f func(T) R) []R {
    out := make([]R, len(s))
    for i, v := range s {
        out[i] = f(v)
    }
    return out
}

// Filter
func Filter[T any](s []T, pred func(T) bool) []T {
    out := make([]T, 0, len(s))
    for _, v := range s {
        if pred(v) {
            out = append(out, v)
        }
    }
    return out
}

// Reduce
func Reduce[T any, R any](s []T, f func(R, T) R, init R) R {
    acc := init
    for _, v := range s {
        acc = f(acc, v)
    }
    return acc
}
```

MapE — map ที่ propagate error (monad-like helper)

```
func MapE[T any, R any](s []T, f func(T) (R, error)) ([]R, error) {
    out := make([]R, 0, len(s))
    for _, v := range s {
        r, err := f(v)
        if err != nil {
            return nil, err
        }
        out = append(out, r)
    }
    return out, nil
}
```

Result / Either แบบง่าย (generic) — เพื่อ chain ฟังก์ชันที่อาจ error

```
type Result[T any] struct {
    Val T
    Err error
}

func Ok[T any](v T) Result[T] { return Result[T]{Val: v} }
func ErrResult[T any](e error) Result[T] { var zero T; return Result[T]{Val: zero, Err: e} }

func (r Result[T]) Map[R any](f func(T) R) Result[R] {
    if r.Err != nil {
        return ErrResult[R](r.Err)
    }
    return Ok(f(r.Val))
}

func (r Result[T]) Bind[R any](f func(T) Result[R]) Result[R] {
    if r.Err != nil {
        return ErrResult[R](r.Err)
    }
    return f(r.Val)
}
```

การใช้ Result ทำให้เราเขียน chain แบบ monadic (แต่ยังคงเป็น Go-style) ได้สะดวกขึ้น

5) Functional + Concurrency — Patterns (ลึก + ปฏิบัติจริง)

Go เหมาะกับการสร้าง **concurrent functional pipelines** โดยใช้ channels และ goroutines — แต่ต้องจัดการเรื่อง cancellation, errors, และ resource leaks

Generator / Map / Filter แบบ stream (ใช้ context)

```
func Generator(ctx context.Context, nums ...int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for _, n := range nums {
            select {
            case <-ctx.Done():
                return
            case out <- n:
            }
        }
    }()
    return out
}

func MapChan(ctx context.Context, in <-chan int, f func(int) int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for v := range in {
            select {
            case <-ctx.Done():
                return
            case out <- f(v):
            }
        }
    }()
    return out
}
```

```
func FilterChan(ctx context.Context, in <-chan int, pred func(int) bool) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for v := range in {
            if pred(v) {
                select {
                case <-ctx.Done():
                    return
                case out <- v:
                }
            }
        }
    }()
    return out
}
```

จุดสำคัญ: ใช้ context เพื่อหยุด pipeline และหลีกเลี่ยง goroutine leak

Fan-out / Fan-in worker pool

- Fan-out: สร้างหลาย worker อ่านจากช่องเดียวกัน ⟶ parallelize CPU work
- Fan-in: รวมผลจากหลาย worker เข้าช่องเดียว

ตัวอย่าง pattern: ใช้ sync.WaitGroup และ close เมื่อหมดงาน

Error handling in concurrent pipelines

- ใช้ errgroup.Group (package x/sync/errgroup) เพื่อรวม error และ cancel ทั้ง pipeline เมื่อเกิด error
- หรือส่ง Result{Val, Err} ผ่าน channel แล้ว aggregate ที่ consumer

---

6) Currying, Partial Application, Composition (จำลองใน Go)

Go ไม่มี syntax สำหรับ currying แต่เราสามารถใช้ closures เพื่อสร้าง partial functions

Currying-like:

```
func Add(a int) func(int) int {
    return func(b int) int { return a + b }
}
```

Compose / Pipe (สองฟังก์ชัน):

```
func Compose[A any, B any, C any](f func(B) C, g func(A) B) func(A) C {
    return func(a A) C { return f(g(a)) }
}
```

ข้อจำกัด: การสร้าง generic variadic compose ยากกว่า แต่สำหรับ 2–3 ฟังก์ชันใช้งานได้ดี

---

7) Immutability ใน Go — วิธีปฏิบัติและประเด็นประสิทธิภาพ

- Go ไม่บังคับ immutability — ทำได้โดย **ไม่แก้ไข input** และคืนค่าใหม่ (copies)
- **การ copy สไลซ์และ struct**:
  - สไลซ์ที่ append จะ allocate ถ้าความจุไม่พอ ⟶ พิจารณา make ด้วย capacity ที่เหมาะสม
  - ใช้ copy(dst, src) เพื่อคัดลอกเนื้อหา (ประสิทธิภาพดีกว่า append ในบางกรณี)
- **ใช้ pointer เมื่อ struct ใหญ่** — แต่ระวัง shared mutability และ race
- **การ reuse buffer**: sync.Pool เพื่อหลีกเลี่ยง allocation ใน hot paths

ตัวอย่าง preallocate:

```
out := make([]T, 0, len(s))
for _, v := range s { out = append(out, transform(v)) }
```

---

8) Pitfalls / Gotchas (ที่เจอบ่อยใน Go + FP)

- **closure capturing loop variable**: ต้องสำเนาตัวแปรภายใน loop

```
for i := range xs {
    v := xs[i] // capture v, not i
    go func() { fmt.Println(v) }()
}
```

- **goroutine leaks**: channels ไม่ถูกปิด / goroutine ถูกบล็อกเพราะ consumer หยุดทำงาน — ใช้ context และ defer close
- **race conditions**: closures ที่ access shared state; ต้องใช้ sync.Mutex หรือ avoid shared state
- **allocation-heavy pipelines**: chaining Map/Filter ที่สร้าง slice ใหม่หลายตัว ⟶ อาจแพ้ performance; ทดสอบและ profile
- **deep recursion**: ไม่มี TCO ⟶ อย่าใช้ recursion ลึก ๆ ใน hot path

---

9) การจัดการ error แบบ functional (pattern)

- Go ใช้ (T, error) pattern — ถ้าต้องการ composition-friendly ให้สร้าง combinators:
  - MapE สำหรับ map ที่ return error

    - Bind/FlatMap สำหรับ chaining (T, error) returning func
- หรือใช้ Result[T] type (ข้างต้น) เพื่อ chain แบบ monadic

ตัวอย่าง Bind:

```
func Bind[T any, R any](v T, f func(T) (R, error)) (R, error) {
    return f(v)
}

// หรือใช้งานกับ Result:
r := Ok(5).Bind(func(x int) Result[int] { return Ok(x*2) })
```

---

10) Testing & Benchmarking (เชิงลึก)

- **Unit tests**: เน้น test pure functions — ง่ายและ deterministic
- **Table-driven tests**: เหมาะกับฟังก์ชันทำนายผล
- **Integration tests**: ทดสอบ edge layer ที่รวม IO + pure core
- **Benchmarks**: ใช้ testing.B เพื่อเปรียบเทียบ in-place mutation vs immutable copy

ตัวอย่าง benchmark skeleton:

```
func BenchmarkMapImmutable(b *testing.B) {
    data := make([]int, 10000)
    for i := 0; i < b.N; i++ {
        _ = Map(data, func(x int) int { return x * 2 })
    }
}

func BenchmarkMapInPlace(b *testing.B) {
    data := make([]int, 10000)
    for i := 0; i < b.N; i++ {
        tmp := make([]int, len(data))
        copy(tmp, data)
        for j := range tmp { tmp[j] *= 2 }
    }
}
```

- **Profile**: ใช้ go test -bench . -benchmem และ pprof (go test -cpuprofile / pprof) เพื่อตัดสินใจ

---

11) ไกด์ไลน์ (Checklist) — เมื่อจะใช้ FP ในโปรเจกต์ Go

1. แยก core logic เป็น pure functions ให้มากที่สุด
2. วาง side effects ไว้ที่ boundary (handlers, adapters)
3. ใช้ generics สำหรับ utilities (Map/Filter/Reduce/MapE) เพื่อ reuse
4. ถ้าเป็น stream/continuous data ⟶ ใช้ channels + context + errgroup
5. ใน hot paths ให้พิจารณา in-place mutation (profile ก่อนเปลี่ยน)
6. ป้องกัน goroutine leaks ด้วย context, close channels, และ errgroup ถ้าเป็นไปได้
7. สร้าง unit tests สำหรับ pure functions — จะได้ความคุ้มค่าเร็วมาก
8. ถ้าทีมยังไม่คุ้น FP — เขียน docs / coding-style และตัวอย่าง เพื่อร่วมมาตรฐาน

---

12) ตัวอย่างโปรเจกต์ฝึกทำ (ไอเดียระดับประยุกต์)

- **Concurrent JSON Transformer** — อ่าน JSON stream ⟶ map/transform fields ⟶ write to output (ใช้ pipeline + MapE + errgroup)
- **Log processing pipeline** — filter, enrich, aggregate logs แบบ streaming (channels, windowing)
- **ETL batch processor** — read CSV ⟶ pure transforms ⟶ validate (MapE) ⟶ write DB with bulk insert
- **Parallel image processor** — fan-out image resizing tasks ⟶ fan-in results (worker pool + FP-style transforms)

---

13) สรุปเชิงปฏิบัติ

- FP ใน Go ให้ประโยชน์จริง: modularity, testability, readable transformation pipelines, synergy กับ concurrency
- ข้อจำกัดเป็นเรื่องของภาษา (no enforced immutability, no TCO, limited syntactic sugar) และ performance tradeoffs (allocation) — แต่ **ถ้ารู้วิธี** (preallocate, reuse buffers, profile, use context/errgroup) FP ใน Go จะเป็นเครื่องมือทรงพลังสำหรับงาน data transformation และ concurrent processing

---

ความหมาย Functional Programming (FP)

☐ **ความหมาย Functional Programming (FP)**

1. Functional Programming คืออะไร

Functional Programming (FP) คือ **paradigm** หรือ **แนวคิดในการเขียนโปรแกรม** ที่ถือว่าฟังก์ชัน (function) เป็น **หน่วยพื้นฐานของการคำนวณ** ทุกอย่างสามารถนิยามในเชิง *mathematical functions* ได้ โดยมุ่งเน้นไปที่:

- การเขียนโปรแกรมในรูปแบบ **การประกาศ (declarative)** มากกว่า **การสั่งงาน (imperative)**
- การหลีกเลี่ยง **mutable state** และ **side effects**
- การใช้ **pure functions** และ **higher-order functions**
- การเน้น **composition** คือการประกอบฟังก์ชันเข้าด้วยกัน เพื่อสร้าง flow ของการคำนวณ

พูดง่าย ๆ FP พยายามทำให้โค้ด:
เหมือนกับการเขียนสูตรทางคณิตศาสตร์ ที่ผลลัพธ์ขึ้นอยู่กับ input เท่านั้น ไม่ขึ้นกับ state ภายนอก

---

2. FP ในเชิงคณิตศาสตร์

แนวคิด FP มาจาก **λ-calculus (lambda calculus)** ซึ่งเป็นรากฐานทางทฤษฎีของการคำนวณ:

- ฟังก์ชันใน FP ≈ *การแมปค่าจาก input ⟶ output*
- เช่น f(x) = x^2 จะ **เสมอ** คืนค่าเลขยกกำลังสองของ input ที่รับมา
- ไม่มี state แอบซ่อน, ไม่มีผลกระทบข้างเคียง

---

3. FP ในเชิงการเขียนโค้ด Go

แม้ Go จะเป็น **imperative language** ที่ออกแบบให้เรียบง่ายและใกล้กับระบบ (system-level programming) แต่ก็มี **ความสามารถบางส่วน** ที่เอื้อให้ใช้แนวทาง FP ได้ เช่น:

- **First-class functions** – ฟังก์ชันเป็นค่าได้ (assign, pass, return)
- **Closures** – ฟังก์ชันสามารถจำค่าจาก scope ภายนอกได้
- **Anonymous functions (lambdas)** – เขียนฟังก์ชัน inline ได้
- **Higher-order functions** – ฟังก์ชันสามารถรับฟังก์ชันเป็น argument และส่งฟังก์ชันกลับมาได้

ตัวอย่าง Go แบบ FP:

```
package main

import "fmt"

// higher-order function
func applyTwice(f func(int) int, x int) int {
    return f(f(x))
```

```
}

func main() {
    square := func(n int) int { return n * n }
    result := applyTwice(square, 2) // (2^2)^2 = 16
    fmt.Println(result)
}
```

---

4. เปรียบเทียบแนวคิด FP กับแนวคิดทั่วไป (Imperative)

- Imperative (Go แบบปกติ):
- sum := 0
- for i := 1; i <= 5; i++ {
- sum += i
- }
- fmt.Println(sum) // 15
- Functional style (Go):
- package main
- import "fmt"
-
- func reduce(nums []int, f func(int, int) int, init int) int {
- acc := init
- for _, n := range nums {
- acc = f(acc, n)
- }
- return acc
- }
-
- func main() {
- nums := []int{1, 2, 3, 4, 5}
- sum := reduce(nums, func(a, b int) int { return a + b }, 0)
- fmt.Println(sum) // 15
- }

---

5. สรุปความหมาย FP ใน Go

- FP คือการเขียนโค้ดโดย **คิดในเชิงคณิตศาสตร์**: input ⟶ output
- Go ไม่ใช่ **pure functional language** แบบ Haskell แต่มีเครื่องมือที่ทำให้เขียนแนว FP ได้
- ใน Go, FP ใช้ได้ดีในงาน:
    - **Data transformation pipelines** (map, filter, reduce)
    - **Concurrency pipelines** ผ่าน channel + goroutines
    - **Testability**: โค้ดง่ายต่อการทดสอบเพราะไม่มี side effects

---

□ ดังนั้น ถ้าจะสรุป **"ความหมาย FP ใน Go"** แบบสั้นแต่ลึก:

FP ใน Go คือการใช้คุณสมบัติของภาษา (first-class functions, closures, higher-order functions) เพื่อเขียนโค้ดที่ **ปราศจาก side effects**, **อ่านเป็น declarative pipeline**, และ **เน้น composition ของฟังก์ชัน** แม้ Go จะไม่ใช่ functional language แท้ แต่สามารถนำแนวคิดนี้มาใช้เพื่อเพิ่มความชัดเจน, modularity, และความสามารถในการทดสอบของโปรแกรมได้

Imperative vs Declarative ในบริบทของ Functional Programming (FP) ใน Go

---

□ **Imperative vs Declarative**

1. Imperative Programming (แนวสั่งงาน)

Imperative คือแนวคิดการเขียนโปรแกรมที่เน้น **“How” ⟶ ต้องทำอย่างไรทีละขั้นตอน**

- โปรแกรมเมอร์บอก **ลำดับการทำงาน (step by step)**
- ใช้ **loops, assignments, state ที่เปลี่ยนแปลงได้ (mutable state)**
- คล้ายการเขียน **สูตรทำอาหาร** ว่าจะต้องทำอะไรบ้างเป็นลำดับ

ตัวอย่าง Imperative Style ใน Go:

```
package main

import "fmt"

func main() {
    nums := []int{1, 2, 3, 4, 5}
    sum := 0
    for i := 0; i < len(nums); i++ {
        sum += nums[i] // อัพเดท state ไปเรื่อย ๆ
    }
```

```
    fmt.Println("Sum:", sum)
}
```

ลักษณะเด่นของ Imperative:

- เน้น **state ที่เปลี่ยนได้** (เช่น sum เปลี่ยนค่าหลายครั้ง)
- ผู้อ่านโค้ดต้อง **ตามลำดับเหตุการณ์** ถึงจะเข้าใจ
- เหมาะกับงานที่ต้อง **ควบคุมรายละเอียดทุกขั้นตอน**

---

2. Declarative Programming (แนวประกาศ)

Declarative คือแนวคิดที่เน้น **"What" ⟶ ต้องการผลลัพธ์แบบไหน**

- โปรแกรมเมอร์บอกว่า **"อยากได้อะไร"** ไม่ใช่ **"ต้องทำยังไง"**
- ลดการจัดการ state โดยตรง
- ใช้ฟังก์ชันแบบ functional เช่น **map, filter, reduce**
- คล้ายกับการบอก **สิ่งที่อยากได้** แทนที่จะอธิบายทุกขั้นตอน

ตัวอย่าง Declarative Style ใน Go (Functional Style):

```
package main

import "fmt"

// reduce function
func reduce(nums []int, f func(int, int) int, init int) int {
    acc := init
    for _, n := range nums {
        acc = f(acc, n)
    }
    return acc
}

func main() {
    nums := []int{1, 2, 3, 4, 5}
    sum := reduce(nums, func(a, b int) int { return a + b }, 0)
    fmt.Println("Sum:", sum)
}
```

ลักษณะเด่นของ Declarative:

- เน้นการ **อธิบายผลลัพธ์ที่ต้องการ** มากกว่าลำดับวิธีทำงาน

- ลด **mutable state** ⟶ โค้ดอ่านง่ายขึ้น
- เหมาะกับ **data transformation**, **query-like operations**, และ **concurrent pipelines**

---

3. เปรียบเทียบ Imperative vs Declarative ใน Go

คุณสมบัติ	Imperative	Declarative
วิธีคิด	บอกวิธีทำงานทีละขั้นตอน	บอกผลลัพธ์ที่ต้องการ
การใช้ state	มีการเปลี่ยนค่า state บ่อย	พยายามลดหรือหลีกเลี่ยงการเปลี่ยน state
โครงสร้าง	loops, conditionals, variables	map, filter, reduce, composition
การอ่านโค้ด	ผู้อ่านต้องตามลำดับทีละบรรทัด	ผู้อ่านเห็น flow ของการแปลงข้อมูลทันที
เหมาะกับ	งาน low-level, ต้องควบคุมละเอียด	งาน data transformation, stream processing

---

4. ตัวอย่างที่เห็นภาพชัด

โจทย์: หาผลรวมของเลขคู่ใน slice [1,2,3,4,5,6]

Imperative Style:

```
nums := []int{1, 2, 3, 4, 5, 6}
sum := 0
for _, n := range nums {
    if n%2 == 0 {
        sum += n
    }
}
fmt.Println(sum) // 12
```

Declarative Style:

```
nums := []int{1, 2, 3, 4, 5, 6}

// filter evens
evens := []int{}
for _, n := range nums {
    if n%2 == 0 {
        evens = append(evens, n)
    }
}
```

```
// reduce sum
sum := 0
for _, e := range evens {
    sum += e
}
fmt.Println(sum) // 12
```

(ถ้า Go มี map/filter/reduce built-in แบบ Haskell/JS จะยิ่ง declarative กว่านี้ แต่เราจำเป็นต้อง implement เอง)

---

5. สรุป Imperative vs Declarative ใน FP for Go

- Imperative = “สั่งงานทีละขั้นตอน” ⟶ ควบคุมละเอียด แต่โค้ดอาจซับซ้อน
- Declarative = “บอกผลลัพธ์ที่อยากได้” ⟶ โค้ดอ่านง่าย เหมาะกับ data pipelines
- Go สนับสนุน **imperative เป็นหลัก** แต่เราสามารถ **เขียน declarative style** ได้ โดยใช้ **higher-order functions, closures, และ functional composition**

□ ถ้าให้เปรียบเทียบสั้น ๆ:

- **Imperative** = ทำอาหารเองทีละขั้นตอน (หั่น, ผัด, ต้ม)
- **Declarative** = บอกเชฟว่า "ฉันอยากกินผัดกะเพราไข่ดาว" แล้วปล่อยให้เชฟจัดการรายละเอียด

ตัวอย่างโปรแกรม Functional Programming ด้วย Go

- **พื้นฐาน (Basic)** ⟶ 5 โปรแกรม
- **แนวประยุกต์ (Applied)** ⟶ 5 โปรแกรม

□ **โครงสร้างการอธิบาย**

แต่ละโปรแกรมจะมีรูปแบบเดียวกัน:

1. □ **ไฟล์เต็ม**
2. □ **โครงสร้าง** (ไฟล์/ฟังก์ชันสำคัญ)
3. □ **คำอธิบายโค้ด**
4. ▶□ **ผลการรัน (Output)**

□ **ส่วนที่ 1: โปรแกรมพื้นฐาน (5 ตัว)**

□ โปรแกรมที่ 1: Pure Function (บวกเลข)

```
package main

import "fmt"

// pure function: ขึ้นกับ input เท่านั้น
func add(a, b int) int {
    return a + b
}

func main() {
    fmt.Println(add(3, 4))
    fmt.Println(add(10, 20))
}
```

โครงสร้าง

- add() ⟶ pure function
- main() ⟶ เรียกใช้งาน

คำอธิบาย

- add เป็น **pure function** เพราะผลลัพธ์ขึ้นกับ input เท่านั้น ไม่มี side effects

ผลการรัน

```
7
30
```

□ โปรแกรมที่ 2: Higher-Order Function

```
package main

import "fmt"

// higher-order: รับ function เป็น argument
func apply(f func(int) int, x int) int {
    return f(x)
}
```

```
func main() {
    square := func(n int) int { return n * n }
    cube := func(n int) int { return n * n * n }

    fmt.Println(apply(square, 3)) // 9
    fmt.Println(apply(cube, 2))   // 8
}
```

คำอธิบาย

- apply รับฟังก์ชัน f และค่าตัวเลข x ⟶ คืนค่าหลังจากเรียก f(x)

ผลการรัน

9

8

☐ **โปรแกรมที่ 3: Map Function (เขียนเอง)**

```
package main

import "fmt"

func mapFunc(nums []int, f func(int) int) []int {
    result := make([]int, len(nums))
    for i, v := range nums {
        result[i] = f(v)
    }
    return result
}

func main() {
    nums := []int{1, 2, 3, 4, 5}
    squared := mapFunc(nums, func(x int) int { return x * x })
    fmt.Println(squared)
}
```

คำอธิบาย

- mapFunc ⟶ ทำหน้าที่คล้าย map ของ FP ในภาษาอื่น
- ส่ง function เข้ามาเพื่อเปลี่ยนทุกค่าใน slice

ผลการรัน

[1 4 9 16 25]

□ โปรแกรมที่ 4: Filter Function

```
package main

import "fmt"

func filterFunc(nums []int, f func(int) bool) []int {
    result := []int{}
    for _, v := range nums {
        if f(v) {
            result = append(result, v)
        }
    }
    return result
}

func main() {
    nums := []int{1, 2, 3, 4, 5, 6}
    evens := filterFunc(nums, func(x int) bool { return x%2 == 0 })
    fmt.Println(evens)
}
```

ผลการรัน

[2 4 6]

□ โปรแกรมที่ 5: Reduce Function

```
package main

import "fmt"

func reduceFunc(nums []int, f func(int, int) int, init int) int {
    acc := init
    for _, v := range nums {
```

```
        acc = f(acc, v)
    }
    return acc
}

func main() {
    nums := []int{1, 2, 3, 4, 5}
    sum := reduceFunc(nums, func(a, b int) int { return a + b }, 0)
    fmt.Println(sum)
}
```

ผลการรัน

```
15
```

☐ ส่วนที่ 2: โปรแกรมแนวประยุกต์ (5 ตัว)

☐ โปรแกรมที่ 1: Sum of Odd Squares (Map ⟶ Filter ⟶ Reduce)

```
package main

import "fmt"

func mapFunc(nums []int, f func(int) int) []int {
    result := make([]int, len(nums))
    for i, v := range nums {
        result[i] = f(v)
    }
    return result
}

func filterFunc(nums []int, f func(int) bool) []int {
    result := []int{}
    for _, v := range nums {
        if f(v) {
            result = append(result, v)
        }
```

```
    }
    return result
}

func reduceFunc(nums []int, f func(int, int) int, init int) int {
    acc := init
    for _, v := range nums {
        acc = f(acc, v)
    }
    return acc
}

func main() {
    nums := []int{1,2,3,4,5,6,7,8,9,10}

    squared := mapFunc(nums, func(x int) int { return x * x })
    odds := filterFunc(squared, func(x int) bool { return x%2 != 0 })
    sum := reduceFunc(odds, func(a, b int) int { return a + b }, 0)

    fmt.Println("Sum of odd squares:", sum)
}
```

ผลการรัน

```
Sum of odd squares: 165
```

□ โปรแกรมที่ 2: Functional String Processing

```
package main

import (
    "fmt"
    "strings"
)

func mapFunc(strs []string, f func(string) string) []string {
    result := make([]string, len(strs))
```

```
    for i, v := range strs {
        result[i] = f(v)
    }
    return result
}

func filterFunc(strs []string, f func(string) bool) []string {
    result := []string{}
    for _, v := range strs {
        if f(v) {
            result = append(result, v)
        }
    }
    return result
}

func main() {
    names := []string{"Alice", "Bob", "Charlie", "adam", "alex"}

    upper := mapFunc(names, strings.ToUpper)
    aNames := filterFunc(upper, func(s string) bool { return strings.HasPrefix(s, "A") })
    lower := mapFunc(aNames, strings.ToLower)

    fmt.Println("Result:", lower)
}
```

ผลการรัน

```
Result: [alice adam alex]
```

□ โปรแกรมที่ 3: Concurrent Pipeline (Channels)

```
package main

import "fmt"

func generator(nums ...int) <-chan int {
```

```
    out := make(chan int)
    go func() {
        for _, n := range nums {
            out <- n
        }
        close(out)
    }()
    return out
}

func mapFunc(in <-chan int, f func(int) int) <-chan int {
    out := make(chan int)
    go func() {
        for v := range in {
            out <- f(v)
        }
        close(out)
    }()
    return out
}

func filterFunc(in <-chan int, f func(int) bool) <-chan int {
    out := make(chan int)
    go func() {
        for v := range in {
            if f(v) {
                out <- v
            }
        }
        close(out)
    }()
    return out
}
```

```
func main() {
    in := generator(1,2,3,4,5,6,7,8,9,10)
    doubled := mapFunc(in, func(x int) int { return x * 2 })
    evens := filterFunc(doubled, func(x int) bool { return x%4 == 0 })

    for v := range evens {
        fmt.Println(v)
    }
}
```

ผลการรัน

```
4
8
12
16
20
```

□ โปรแกรมที่ 4: JSON Data Transformation

```
package main

import (
    "encoding/json"
    "fmt"
    "strings"
)

type User struct {
    Name string `json:"name"`
    Age  int    `json:"age"`
}

func mapUsers(users []User, f func(User) User) []User {
    result := make([]User, len(users))
    for i, u := range users {
        result[i] = f(u)
```

```
    }
    return result
}

func filterUsers(users []User, f func(User) bool) []User {
    result := []User{}
    for _, u := range users {
        if f(u) {
            result = append(result, u)
        }
    }
    return result
}

func main() {
    jsonData := `[{"name":"Alice","age":30},{"name":"Bob","age":20},{"name":"alex","age":25}]`
    var users []User
    json.Unmarshal([]byte(jsonData), &users)

    adults := filterUsers(users, func(u User) bool { return u.Age >= 21 })
    upper := mapUsers(adults, func(u User) User {
        u.Name = strings.ToUpper(u.Name)
        return u
    })

    result, _ := json.MarshalIndent(upper, "", "  ")
    fmt.Println(string(result))
}
```

ผลการรัน

```
[
  {
    "name": "ALICE",
    "age": 30
  },
```