Go und Elixir

funktional

Gliederung

01
Algorithmus
Shunting-Yard und RPN-Calculator

02
Architektur
Programmarchitektur und Datenfluss

03
Demo

04
Vergleich
Umsetzung in Go und Elixir

05
Takeaways

Worum geht es?

Wie berechnen Maschinen mathematische Ausdrücke?

„Normale“ Infix-Notation:

- Programmatisch schwer zu verarbeiten
- Klammern und Operatorpräzedenz

Infix

```
2 * (1 - 3)
```

Reverse Polish Notation (RPN):

- Keine Klammern & Präzedenzregeln
- Lineare, Stackbasierte Auswertung

RPN

```
2 1 3 - *
```

Ziel: Umformen mit Shunting-Yard Algorithmus & Ergebnis ausrechnen

Shunting-Yard Algorithmus

→ Start mit einer Liste an Eingabetokens

- Verarbeitung von links nach rechts
- Zwei Datenstrukturen:
  - Output-Queue
  - Operator-Stack
- Grundregeln:
  - Zahlen → direkt zur Ausgabe
  - Operatoren → Warten im Stack (nach Präzedenz)
  - Klammern → Bestimmen, wann Operatoren in die Queue kommen

→ nächster Schritt: RPN-Berechner

Beispiel „2 * (1 - 3)“

Input (Infix):
2 * (1 - 3)

Aktuelles Token:

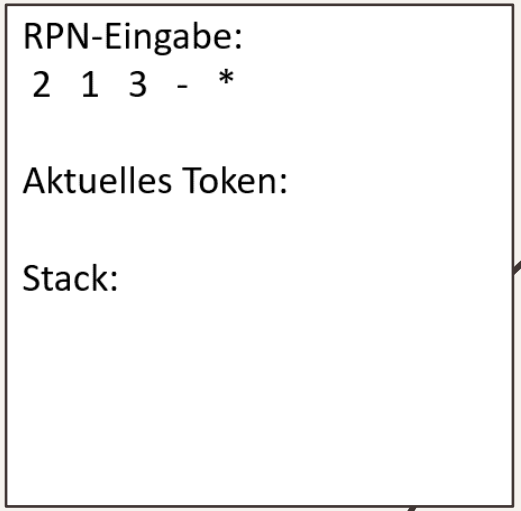
Stack:

Output (RPN):

RPN-Calculator

- Verarbeitung von links nach rechts
- Stackbasierte Auswertung
- Grundregeln:
  - Zahlen → Stack
  - Operatoren → 2 Werte reduzieren

Beispiel „2 1 3 - *“



Architektur & Datenfluss

04
Vergleich

Umsetzung in Go und Elixir

Tokenmodellierung

Elixir: Tupel, Atome

```
{:num, 3.0}
{:op, :+}
:lpar
:rpar
```

- Bedeutung ist der Wert

→ Pattern Matching auf Struktur

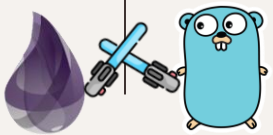
Go: Structs, Enums, explizite Typen

```
type Token struct {
    Kind TokenKind
    Num float64
    Op Op
}
```

- Bedeutung nicht direkt im Wert kodiert

→ explizite Fallunterscheidung

→ passendes Feld manuell auslesen



Kontrollfluss

Elixir

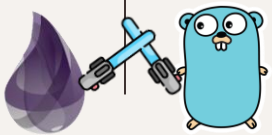
```
def step({:num, x}, state), do: ...
def step({:op, op}, state), do: ...
def step(:lpar, state), do: ...
```

- Fallunterscheidung über Pattern Matching
- Kontrollfluss folgt der Datenstruktur

Go

```
switch tok.Kind {
case TokNum:
    ...
case TokOp:
    ...
}
```

- Fallunterscheidung über `switch case`
- Kontrollfluss folgt der Programmlogik



Umgang mit Zustand

Elixir

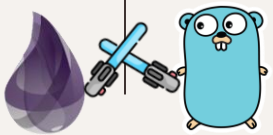
```elixir
Enum.reduce(tokens, init, fn tok, state ->
  step(tok, state)
end)
```

- Keine Mutation
- Zustand wird weitergereicht
- Deklarativ: beschreibt Zustandsübergänge

Go

```go
for _, tok := range tokens {
    state = step(tok, state)
}
```

- Mutation erlaubt → lokal begrenzt
- Nach außen hin „pure“ Funktion
- Imperativ: beschreibt Ablaufschritte



Error-Handling

Elixir

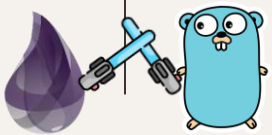
```elixir
{:ok, result}
{:error, reason}
```

- Fehler sind „normale“ Rückgabewerte
- Fließen im Datenstrom mit
- Entscheidung oft am Ende der Pipeline

Go

```go
result, err := f(...)
if err != nil {
    return err
}
```

- Fehler sind Sonderfälle
- Explizite Prüfung
- Entscheidung bei jedem Schritt



Testing

Elixir

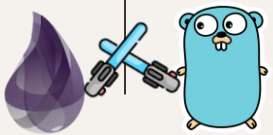
```elixir
assert {:ok, [_ | _]} = result
```

- Tests kompakt und deklarativ
- Pattern Matching in Assertions

Go

```go
if err != nil {
    t.Fatal(err)
}
```

- Tests explizit und schrittweise
- Helper-Funktionen für Wiederverwendung



Vergleich: Funktionale Umsetzung

	Elixir	Go
Tokenmodellierung	Bedeutung strukturell im Wert	Unterscheidung über Feld
Kontrollfluss	Pattern Matching	switch case
Umgang mit Zustand	wird weitergereicht	lokal verändert
Error-Handling	Teil des Datenflusses	explizit geprüft
Testing	Kompakt und deklarativ	explizit

Takeaways

Elixir

- Ungewohnt, vorallem am Anfang
- Klar und kompakt

Go

- Gut nachvollziehbar, expliziter Kontrollfluss
- Funktionale Konzepte: was ist idiomatisch und was bewusst gewählt?
- Mehr Boilerplate (≈ 350 LoC vs 220 in Elixir)