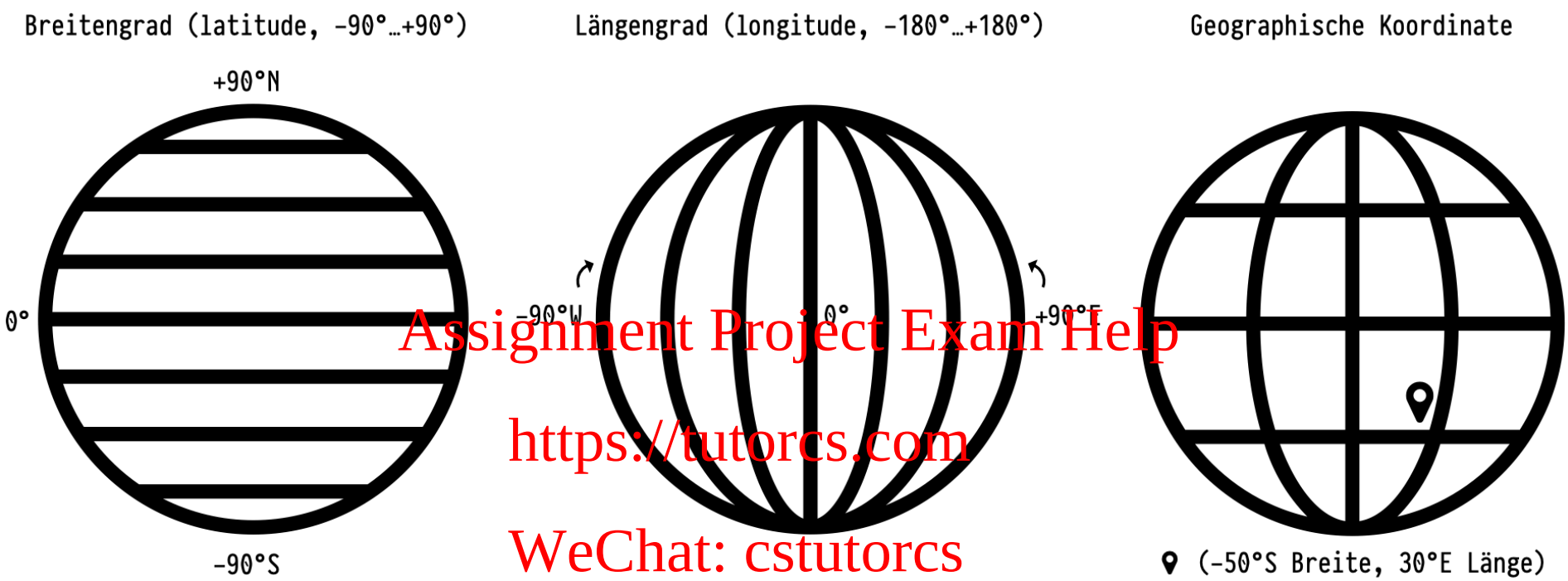
Informatik 1

05 – Prädikat-Signaturen, Benannte Signaturen, Gemischte Daten, Pattern Matching, Lokale Definitionen

Winter 2020/21

Torsten Grust
Universität Tübingen, Germany

1 Geographische Koordinaten (Breiten-/Längengrade)



Bestimmung der geographischen Koordinaten des Ortes

- Zum **Geocoding** siehe auch nominatim.openstreetmap.org.

2 Prädikat-Signaturen

Sei p ein Prädikat mit Signatur $(t \to \text{boolean})$. Eine **Prädikat-Signatur** der Form

```
(predicate p)
```

gilt für jeden Wert x, der

1. Signatur t besitzt **und**
2. zusätzlich $(p\ x) \rightsquigarrow$ erfüllt.

Signatur $(\text{predicate}\ p)$ ist damit **spezifischer** (**restriktiver**) als Signatur t.

3 Benannte Signaturen

Für eine (komplexe) Signatur *t* kann mittels

```
(define new-t (signature t))
```

ein **neuer Signaturname** *new-t* als selbstdokumentierende Abkürzung (**Abstraktion!**) definiert werden.

Beispiele:

```
(define farbe
  (signature (one-of "Karo" "Herz" "Pik" "Kreuz")))

(define latitude
  (signature (predicate latitude?)))
```

4 ¦ Geocoding API (Teachpack geocoder.rkt)

Übersetze eine Ortsbeschreibung mittels des *OpenStreetMaps* **Geocoding** API in eine Position auf der Erdkugel:

```
(: geocoder (string -> geocode))
```


Ein geocode Record besteht aus folgenden Komponenten:

Komponente	Name	Signatur
Adresse	address	string
Ortsangabe	loc	location
Nordostecke	northeast	location
Südwestecke	southwest	location
Kategorie	class	string


```
(: geocode-address (geocode -> string))  ; Selektor
                   ⋮
```

5 Gemischte Daten

Signatur `mixed` beschreibt **gemischte Daten:**

```
(mixed t₁ ... tₙ)
```

ist gültig für jeden Wert, der *mindestens eine* der Signaturen t_1, ..., t_n erfüllt.

Beispiel:

- `(: string->number (string -> (mixed number (one-of #f))))`
  konvertiert Strings in Zahlen (falls möglich):

```
(string->number "42")     ⇝  42
(string->number "five")   ⇝  #f  ; Konversion gescheitert
```

Geocoding API (vervollständigt mit Fehlerbehandlung)

```
(: geocoder (string -> (mixed geocode geocode-error)))
```

Komponenten von `geocode`:

Komponente	Name	Signatur
Adresse	`address`	`string`
Ursprung	`loc`	`location`
Nordostecke	`northeast`	`location`
Südwestecke	`southwest`	`location`
Kategorie	`class`	`string`

Komponenten von `geocode-error`:

Komponente	Name	Signatur
Fehlerart	`level`	`(one-of "TCP" "HTTP" "JSON" "API")`
Fehlermeldung	`message`	`string`

Prädikate, die Signaturen von Werten erkennen

- Das **Prädikat** *t?* zu einer Signatur *t* unterscheidet Werte der Signatur *t* von **allen anderen** Werten:

```
; (t? x) ≡ hat x die Signatur t ?
(: t? (any -> boolean))
```


  - **NB:** Die Signatur `any` ist für *alle* Werte (*"any value"*) gültig und stellt damit keinerlei Restriktion dar.

- **Prädikate** existieren sowohl für eingebaute Signaturen als auch für Records:

```
(natural? 42)          ⇝ #t  ┆  (string? 42)          ⇝ #f
(natural? "fortytwo")  ⇝ #f  ┆  (string? "fortytwo")  ⇝ #t
(location? (make-location -50.0 30.0)) ⇝ #t
(location? "Mos Eisley, Tatooine")     ⇝ #f
```

6 Konstruktionsanleitung: Funktionen über gemischen Daten

Konstruktionsanleitung für Funktion f, die **gemischte Daten** der Signaturen t_1,...,t_n **konsumiert:**

```
⋮                                  ; ← Kurzbeschreibung + Testfälle
(: f (... (mixed t₁ ... tₙ) ... -> ...))
(define f
  (lambda (... x ...)              hier hat x definitiv Signatur t₁
    ...
    (cond ((t₁? x) ...)  ; 1
          ⋮
          ((tₙ? x) ...))
    ...
  ))                               hier hat x definitiv Signatur tₙ
```


- In Branch 1 $(t_1?\ x) \rightsquigarrow$ `#t`: behandle x wie einen Wert der Signatur t_1.

7 ¦ Pattern Matching

Pattern Matching kombiniert ❶ *Fallunterscheidung* und ❷ *Dekonstruktion* zusammengesetzter Daten.

❶ **Fallunterscheidung:** Spezialform **match** vergleicht Wert e mit gegebenen **Patterns** $pat_1, pat_2, \ldots, pat_n$:


```
(match e
  (pat₁ e₁)
  (pat₂ e₂)
      ⋮
  (patₙ eₙ)
)
```


Patterns pat_i werden von oben nach unten gegen e *gematched*

- Falls pat_i das erste auf e *matchende* Pattern ist, ist Zweig e_i das Ergebnis des **match**. Sonst Programmabbruch ↯.

Pattern Matching — Wann *matched* ein Pattern?

2 **Dekonstruktion** (siehe ⚑) durch Patterns:

```
(match e ... (pat_i e_i) ...)
```

- Jedes **Pattern** pat_i kann eine von vier Formen annehmen:
  1. ℓ (**Literal**):
     e matched, falls $e \to \ell$.
  2. `_` (***Don't care***, Unterstrich):
     e matched immer.
  3. id (**Identifier**):
     e matched immer, im Zweig e_i ist id an e gebunden.
  4. $(\text{make-}t\ pat_{i1}\ \ldots\ pat_{ik})$ (**Record-Konstruktur**, $k \geqslant 0$):
     e matched, falls es via $(\text{make-}t\ x_1\ \ldots\ x_k)$ konstruiert wurde und ⚑ alle x_j auf pat_{ij} ($j = 1 \ldots k$) matchen.

8 Lokale Definition mittels let

Beispiel:

Abstand dist(•,•) zweier geographischer Positionen l_1 und l_2 auf der Erdkugel in km (lat, lng in Radian (180°~π)):

$$\begin{aligned}\text{dist}(l_1,l_2) = {} & \text{Erdradius in km} \times \\ & \text{acos}(\cos(l_1.\text{lat}) \times \cos(l_1.\text{lng}) \times \cos(l_2.\text{lat}) \times \cos(l_2.\text{lng}) + \\ & \quad \cos(l_1.\text{lat}) \times \sin(l_1.\text{lng}) \times \cos(l_2.\text{lat}) \times \sin(l_2.\text{lng}) + \\ & \quad \sin(l_1.\text{lat}) \times \sin(l_2.\text{lat}))\end{aligned}$$

- Berechnung von dist führt zur **wiederholten Auswertung identischer Teilausdrücke:**
  - l_1.lat (3×), l_1.lng (2×), l_2.lat (3×), l_2.lng (2×)

Lokale Definition mittels `let`

Mittels **let** lassen sich Werte e_i **lokal** an Namen id_i binden:

```
(let ((id₁ e₁)
      (id₂ e₂)
          ⋮
      (idₙ eₙ))
  ⌜e⌝)                    ; der Wert von e ist Wert
  ⌞ ⌟                     ; des gesamten let-Ausdrucks
```


- id_1, ..., id_n können in e (**nur hier** ⌷) verwendet werden:
  1. Außerhalb des **let** sind die id_i unbekannt.
  2. In den e_i sind die id_i unbekannt
     (alle e_i werden [potentiell] *parallel ausgewertet*).

⚠ **let** ab Sprachlevel *Die Macht der Abstraktion* verfügbar.

`let` ist syntaktischer Zucker

`let` ist **syntaktischer Zucker** (eine **derived form**) und lässt sich mit existierenden Konstrukten äquivalent ausdrücken:

$$\texttt{(let ((}id_1\ e_1\texttt{)}\ \vdots\ \texttt{(}id_n\ e_n\texttt{))}\ e\texttt{)} \equiv \texttt{((lambda (}id_1\ \ldots\ id_n\texttt{)}\ e\texttt{)}\ e_1\ \ldots\ e_n\texttt{)}$$

- Links und rechts vom $\equiv$:
  1. In e (und **nur hier**) ist id_i an e_i gebunden.
  2. Der Wert von e ist Ergebnis des gesamten Ausdrucks.

(Daher verbreitete Sprechweise: e ist der *Body* des `let`.)